SYBASE REPLICATION SERVER PERFORMANCE AND TUNING

Understanding and Achieving Optimal Performance with Sybase Replication Server
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents</td>
<td>2</td>
</tr>
<tr>
<td>Author’s Note</td>
<td>4</td>
</tr>
<tr>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>Overview and Review</td>
<td>6</td>
</tr>
<tr>
<td>Replication System Components</td>
<td>6</td>
</tr>
<tr>
<td>Replication Server Internal Processing</td>
<td>7</td>
</tr>
<tr>
<td>Analyzing Replication System Performance</td>
<td>10</td>
</tr>
<tr>
<td>Primary DataServer/Database</td>
<td>12</td>
</tr>
<tr>
<td>DataServer Configuration Parameters</td>
<td>12</td>
</tr>
<tr>
<td>Primary Database Transaction Log</td>
<td>13</td>
</tr>
<tr>
<td>Application/Database Design</td>
<td>13</td>
</tr>
<tr>
<td>Replication Agent Processing</td>
<td>20</td>
</tr>
<tr>
<td>Secondary Truncation Point Management</td>
<td>20</td>
</tr>
<tr>
<td>Rep Agent LTL Generation</td>
<td>21</td>
</tr>
<tr>
<td>Replication Agent Communications</td>
<td>25</td>
</tr>
<tr>
<td>Replication Agent Troubleshooting</td>
<td>25</td>
</tr>
<tr>
<td>Replication Server General Tuning</td>
<td>27</td>
</tr>
<tr>
<td>Replication Server/RSSD Hosting</td>
<td>27</td>
</tr>
<tr>
<td>RS Generic Tuning</td>
<td>29</td>
</tr>
<tr>
<td>RSSD Generic Tuning</td>
<td>33</td>
</tr>
<tr>
<td>RSM/SMS Monitoring</td>
<td>33</td>
</tr>
<tr>
<td>Inbound Processing</td>
<td>36</td>
</tr>
<tr>
<td>SQM Processing</td>
<td>36</td>
</tr>
<tr>
<td>SQT Processing</td>
<td>40</td>
</tr>
<tr>
<td>Distributor Processing</td>
<td>47</td>
</tr>
<tr>
<td>Minimal Column Replication</td>
<td>53</td>
</tr>
<tr>
<td>Outbound Queue Processing</td>
<td>58</td>
</tr>
<tr>
<td>DSI SQT Processing</td>
<td>58</td>
</tr>
<tr>
<td>DSI Transaction Grouping</td>
<td>60</td>
</tr>
<tr>
<td>DSI Function String Generation</td>
<td>64</td>
</tr>
<tr>
<td>DSI Command Batching</td>
<td>67</td>
</tr>
<tr>
<td>RSI Queue Processing</td>
<td>68</td>
</tr>
<tr>
<td>Replicate DataServer/Database</td>
<td>70</td>
</tr>
<tr>
<td>Performance Monitoring</td>
<td>70</td>
</tr>
<tr>
<td>Replicate DataServer Resources</td>
<td>72</td>
</tr>
<tr>
<td>Trigger/Procedure Execution Time</td>
<td>72</td>
</tr>
<tr>
<td>Concurrency Issues</td>
<td>77</td>
</tr>
<tr>
<td>Parallel DSI Performance</td>
<td>79</td>
</tr>
<tr>
<td>Need for Parallel DSI</td>
<td>79</td>
</tr>
<tr>
<td>Parallel DSI Internals</td>
<td>80</td>
</tr>
<tr>
<td>Serialization Methods</td>
<td>85</td>
</tr>
<tr>
<td>Large Transaction Processing</td>
<td>90</td>
</tr>
<tr>
<td>Maximizing Performance with Parallel DSI’s</td>
<td>96</td>
</tr>
<tr>
<td>Text/Image Replication</td>
<td>100</td>
</tr>
<tr>
<td>Text/Image Datatype Support</td>
<td>100</td>
</tr>
<tr>
<td>RS Implementation &amp; Internals</td>
<td>102</td>
</tr>
<tr>
<td>Performance Implications</td>
<td>108</td>
</tr>
<tr>
<td>Topic</td>
<td>Page</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>------</td>
</tr>
<tr>
<td>Asynchronous Request Functions</td>
<td>110</td>
</tr>
<tr>
<td>Purpose</td>
<td>110</td>
</tr>
<tr>
<td>Implementation &amp; Internals</td>
<td>112</td>
</tr>
<tr>
<td>Performance Implications</td>
<td>115</td>
</tr>
<tr>
<td>Multiple DSFs</td>
<td>116</td>
</tr>
<tr>
<td>Concepts &amp; Terminology</td>
<td>116</td>
</tr>
<tr>
<td>Performance Benefits</td>
<td>116</td>
</tr>
<tr>
<td>Implementation</td>
<td>117</td>
</tr>
<tr>
<td>Business Cases</td>
<td>132</td>
</tr>
</tbody>
</table>
Author’s Note

Thinking is hard work – “Silver Bullets” are much easier.

Several years ago, when Replication Server 11.0 was fairly new, Replication Server Engineering (RSE) collaborated on a paper that was a help to us all. Since that time, Replication Server has gone through several releases and Replication Server Engineering has been too busy keeping up with the advances in Adaptive Server Enterprise and the future of Replication Server to maintain the document. However, the requests for a paper such as this have been a frequent occurrence, both internally as well as from customers. Hopefully, this paper will satisfy those requests. But as the above comment suggests, reading this paper will require extensive thinking (and considerable time). Anyone hoping for a “silver bullet” does not belong in the IT industry.

This paper was written for and addresses the functionality in Replication Server 12.0. As the Replication Server product continues to be developed and improved, it is likely that later improvements to the server, especially anticipated improvements scheduled for the 12.1 release, may supercede the recommendations contained in this paper.

It is assumed that the reader is familiar with Replication Server terminology, internal processing and in general the contents of the Replication Server System Administration Guide. In addition, basic Adaptive Server Enterprise performance and tuning knowledge is considered critical to the success of any Replication System’s performance analysis.
Introduction

"Just How Fast Is It?"

This question gets asked constantly. Unfortunately, there are no standard benchmarks such as TPC-C for replication technologies and RSE does not have the bandwidth nor resources to do benchmarking. Consequently, the stock RSE reply used to be 5MB/min (or 300MB/hr) based on their limited testing on development machines (small ones at that). However, Replication Server has been clocked at 2.4GB/hr sustained in a 1.2TB database and more than 40GB has been replicated in a single day into the same 1.2TB database (RS 12.0 and ASE 11.9.3 on Compaq Alpha GS140’s for the curious). Additionally, some customers have claimed that by using multiple DSI’s, they have achieved 10,000,000 transactions an hour!! Although this sounds unrealistic, a monitored benchmark in 1995 using Replication Server 10.5 achieved 4,000,000 transactions (each with 10 write operations) a day from the source replicating to three destinations (each with only 5 DSI’s) for a total delivery of 12,000,000 transactions per day (containing 120,000,000 write operations). As usual, your results may vary. Significantly. It all depends. And every other trite caveat muttered by a tuning guru/educator/consultant. The goal of this paper is to educate so that the reader understands why they may be seeing the performance they are and suggest possible avenues to explore with the goal of improved performance without resorting to the old tried-and-true trial-and-error stumble-fumble.

Before we begin, however, it is best to lay some ground rules about what to expect or not to expect from this paper. Focusing on the latter:

- This paper will not discuss database server performance and tuning (although it frequently is the cause of poor replication performance) except as required for replication processing.
- This paper will not discuss non-ASE RepAgent performance (perhaps it will in a future version) except where such statements can be made generically about RepAgents.
- This paper will not discuss Replication Server Manager.
- This paper will not discuss how to “benchmark” a replicated system.
- This paper will not discuss Replication Server system administration.

Now that we know what we are going to skip, what we will cover:

- This paper will discuss all of the components in a replication system and how each impacts performance.
- This paper will discuss the internal processing of the Replication Server, ASE’s Replication Agent and the corresponding tuning parameters that are specific for performance.

It is expected that the reader is already familiar with Replication Server internal processing and basic replication terminology as described in the product manuals. This paper focuses heavily on Replication Server in an Adaptive Server Enterprise environment.

In the future, it is expected that this paper will be expanded to cover several topics only lightly addressed in this version or not addressed at all. These topics include:

- Heterogeneous replication performance
- Replication Routes (Direct and Indirect)
- Replication Server 12.1+ Monitors and Counters
Overview and Review

Where Do We Start?

Unfortunately, this is the same question that is asked by someone faced with the task of finding and resolving throughput performance problems in a distributed system. The last words of that sentence hold the key…it’s a distributed system. That means that there are lots of pieces and parts that contribute to Replication Server performance – most of which are outside of the Replication Server. After the system has been in operation, there are several RS commands that will help isolate where to begin. However, if just designing the system and you wish to take performance into consideration during the design phase (always a must for scalable systems), then the easiest place to begin is the beginning. Accordingly, this paper will attempt to trace a data bit being replicated through the system. Along the way, the various threads, processes, etc. will be described to help the reader understand what is happening (or should happen?) at each stage of data movement. After getting the data to the replicate site, a number of topics will be discussed in greater detail. These topics include text/image replication, parallel DSI’s, etc. A quick review of the components in a replication system and the internal processing within Replication Server are illustrated in the next sections.

Replication System Components

The components in a basic replication system are illustrated below. For clarity, the same abbreviations used in product manuals as well as educational materials are used. The only addition to this over pictures in the product manuals is the inclusion of SMS – in particular, Replication Server Manager (RSM) and the inclusion of the host for the RS/RSSD.

Of course, the above is extremely simple – the basic single direction primary to replicate distributed system, one example of which is the typical Warm-Standby configuration.

Whether for performance reasons or due to architectural requirements, often the system design involves more than one RS. A quick illustration is included below:
The above is still fairly basic. Today, some customers have progressed to multi-level tree-like structures or virtual networks exploiting high-speed bandwidth backbones to form information buses.

**Replication Server Internal Processing**

When hearing the terms “internal processing”, most Replication Server administrators immediately picture the internal threads. While understanding the internal threads is an important fundamental concept, it is strictly the starting point to beginning to understand how Sybase Replication Server processes transactions. Unfortunately, many Replication Server administrators stop there, and as a result never really understand how Replication Server is processing their workload. Consequently, this leaves the administrator ill equipped to resolve issues and in particular to analyze performance bottlenecks within the distributed system. Details about what is happening within each thread as data is replicated will be discussed in later chapters.

**Replication Server Threads**

There are several different diagrams that depict the Replication Server internal processing threads. Most of these are extremely similar and only differ in the relationships between the SQM, SQT and dAIO threads. For the sake of this paper, we will be using the following diagram, which is slightly more accurate than those documented in the Replication Server Administration Guide:
Replicated transactions flow through the system as follows:

1. Replication Agent forwards logged changes scanned from the transaction log to the Replication Server.
2. Replication Agent User thread functions as a connection manager for the Replication Agent and passes the changes to the SQM.
3. The Stable Queue Manager (SQM) writes the logged changes to disk via the Asynch I/O daemon. Once written to disk, the Replication Agent can safely move the secondary truncation point forward (based on scan_batch_size setting).
4. Transactions from source systems are stored in the inbound queue until a copy has been distributed to all subscribers (outbound queue).
5. The Stable Queue Transaction (SQT) thread requests the next disk block from the SQM and sorts the transactions into commit order.
6. Once the commit record for a transaction has been seen, the SQT alerts the Distributor thread that a transaction is available. The Distributor reads the transaction and determines who is subscribing to it, whether subscription migration is necessary, etc.
7. Once all of the subscribers have been identified, the Distributor thread forwards the transaction to the SQM for the outbound queue for the destination connection.
8. The SQM writes the transaction out to the outbound queue via the Asynch I/O daemon.
9. Transactions are stored in the outbound queue until delivered to the destination.
10. The DSI Scheduler uses the SQT library functions to retrieve transactions from the outbound queue it transaction form and determines delivery strategy (batching, grouping, parallelism, etc.)
11. Once the delivery strategy is determined, the DSI Scheduler then passes the transaction to a DSI Executor.
12. The DSI Executor translates the replicated transaction functions into the destination command language (i.e. Transact SQL) and applies the transaction to the replicated database.

Again, the only difference here vs. those in the product manuals is the inclusion of the System Table Services (STS), Asynchronous I/O daemon (dAIO), SQT/SQM and queue data flow and the lack of a SQT thread reading from the outbound queue (instead, the DSI-S is illustrated making SQT library calls). While the difference is slight, it is illustrated here for future discussion. Keeping these differences in mind, the reader is referred to the Replication Server System Administration Guide for details of internal processing for replication systems involving routing or Warm Standby.
Inter-Thread Messaging

Additionally, inter-thread communications is not done through a strict synchronous API call. Instead, each thread simply writes a message into the target thread’s OpenServer message queue (standard OpenServer in memory message structures for communicating between OpenServer threads). Once the target thread has processed each message, it can use standard callback routines or put a response message back into a message queue for the sending thread. This resembles the following:

![OpenServer Message Queues Diagram](image)

Those familiar with multi-threaded programming or OpenServer programming will recognize this as a common technique for communication between threads – especially when multiple threads are trying to communicate with the same destination thread. Accordingly, callbacks are used primarily between threads in which one thread spawned the other and the child thread needs to communicate to the parent thread. An example of this in Replication Server is the DIST and SQT threads. The SQT thread for any primary database is started by the DIST thread. Consequently, in addition to using message queues, the SQT and DIST threads can communicate using Callback routines.

Note that the message queues are not really tied to a specific thread - but rather to a specific message. As a result, a single thread may be putting/retrieving messages from multiple message queues. Consequently, it is possible to have more message queues than threads, although the current design for Replication Server doesn’t require such. By now, those familiar with many of the Replication Server configuration parameters will have realized the relationship between several fairly crucial configuration parameters: num_threads, num_msgqueues and num_msgs (especially why this number could be a large multiple of num_msgqueues). Since this section was strictly intended to give you a background in Replication Server internals, the specifics of this relationship will be discussed later in the section discussion Replication Server tuning.

OQID Processing

One of the more central concepts behind replication server recovery is the OQID – Origin Queue Identifier. The OQID is used for duplicate and loss detection as well as determining where to restart applying transactions during recovery. The OQID composition depends largely on the source system. For Sybase ASE, the OQID is a 36 byte binary value composed of the following elements:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2</td>
<td>Database generation id (from dbcc gettrunc())</td>
</tr>
<tr>
<td>3-8</td>
<td>Log page timestamp</td>
</tr>
<tr>
<td>9-14</td>
<td>Log page rowid (rid)</td>
</tr>
<tr>
<td>15-20</td>
<td>Log page rid for the oldest transaction</td>
</tr>
<tr>
<td>21-28</td>
<td>Datetime for oldest transaction</td>
</tr>
<tr>
<td>29-30</td>
<td>Used by RepAgent to delete orphaned transactions</td>
</tr>
<tr>
<td>31-32</td>
<td>Unused</td>
</tr>
<tr>
<td>33-34</td>
<td>Appended by TD for uniqueness</td>
</tr>
</tbody>
</table>
Through the use of the database generation id, log page timestamp and log record row id (rid), ASE guarantees that the
OQID is always increasing sequentially. As a result, any time the RS detects a OQID lower than the last one, it can
somewhat safely assume that it is a duplicate. Similarly at the replicate, when the DSI compares the ODID in the
rs_lastcommit table with the one current in the active segment, it can detect if the transaction has already been applied.

Why would there be duplicates?? Simply because the Replication Server isn’t updating the RSSD or the rs_lastcommit
table with every replicated row. Instead, it is updating every so often after a batch of transactions has been applied.

Should the system be halted mid-batch and then restarted, it is possible that the first several have already been applied.
At the replicate, a similar situation occurs in that the Replication Server begins by looking at the oldest active segment
in the queue – which may contain transactions already applied.

For heterogeneous systems, the database generation (bytes 1-2) and the RS managed bytes (33-36) are the same,
however the other components depend on what may be available to the replication agent to construct the OQID. This
may include system transaction id’s or other system generated information that uniquely identifies each transaction to
the Replication Agent.

An important aspect of the OQID is the fact that each replicated row from a source system is associated with only one
OQID and vice versa. This is key to not only identifying duplicates for recovery after a failure (i.e. network outage),
but also in replication routing. From this aspect, the OQID ensures that only a single copy of a message is delivered in
the event that the routing topology changes. Those familiar with creating intermediate replication routes and concept of
logical network topology provided by the intermediate routing capability will recognize the benefit of this behavior.

The danger is that some people have attempted to use the OQID or origin commit time in the rs_lastcommit table for
timing. This is extremely inaccurate. First, the origin commit time comes from the timestamp in the commit record (a
specific record in the transaction log) on the primary. This time is derived from the dataserver’s clock, which is
synched with the system clock about once per minute. There can be drift obviously, but not more than a minute as it is
re-synched each minute. The dest_commit time in the rs_lastcommit table, on the other hand, comes from the getdate()
function call in rs_update_lastcommit. The getdate() function is a direct poll of the system clock on the replicate. The
resulting difference between the two could be quite large in one sense or even negative if the replicate’s clock was
slow.

The best mechanism to determining latency is to simply run a batch of 1,000 atomic inserts into the primary and
monitor the end time at the primary and replicate. For large sets of transactions, obviously a stop watch is not even
necessary. If the Replication Server is keeping the system up to the point a stop watch would be necessary, then you
don’t have a latency problem. If, however, it finishes at the primary in 1 minute and at the replicate in 5 minutes – then
you have a problem.

Analyzing Replication System Performance

Having set the stage, the rest of this document will be divided into sections detailing how these components work in
relation to possible performance issues. The major sections will be:

- Primary DataServer/Database
- Replication Agent Processing
- Replication Server and RSSD General Tuning
- Inbound Processing
- Outbound Queue Processing
- Replicate DataServer/Database
After these sections have been covered in some detail, this document will then cover several special topics related to DSI processing in more detail. This includes:

- Parallel DSI Performance
- Text/Image Replication
- Asynchronous Request Functions
- Multiple DSI’s
Primary DataServer/Database

It is Not Possible to Tune a Bad Design

The above comment is the ninth principal of the “Principals of OLTP Processing” as stated by Nancy Mullen of Andersen Consulting in her paper OLTP Program Design in OLTP Processing Handbook (McGraw-Hill). A truer statement has never been written. Not only can you not fix it by replication, but in most cases, a bad design will also cause replication performance to suffer. In many cases when replication performance is bad, we tend to focus quickly at the replicate. While it is true that many replication performance problems can be resolved there, the primary database often also plays a significant role. In this section, we will begin with basic configuration issues and then move into some of the more problematic design issues that affect replication performance.

DataServer Configuration Parameters

While Sybase has striven (with some success) to make replication transparent to the application, it is not transparent to the database server. In addition to the Replication Agent Thread (even though significantly better than the older LTM’s as far as impact on the dataserver), replication can impact system administration in many ways. One of those ways is proper tuning of the database engine’s system configuration settings. Several settings that would not normally be associated with replication, nonetheless, have a direct impact on the performance of the Replication Agent or in processing transactions within the Replication Server.

Procedure Cache Sizing

A common misconception is that procedure cache is strictly used for caching procedure query plans. However, in recent years, this has changed. The reason is than in most large production systems, the procedure cache was grossly oversized, consequently under utilized and contributed to the lack of resources for data cache. For example, in a system with 2GB of memory dedicated to the database engine, the default of 20% often meant that ~400MB of memory was being reserved for procedure cache. Often, real procedure cache used by stored procedure plans is less than 10MB. ASE engineers began tapping in to this resource by caching subquery results, etc. in procedure cache. When the Replication Agent thread was internalized within the ASE engine (ASE 11.5), it was no different. It also used procedure cache.

The Replication Agent uses procedure cache for several critical functions:

- Schema Cache - Caching for database object structures, such as table, column names, text/image replication states, used in the construction of LTL.
- Transaction Cache - Caching LTL statements pending transfer to the Replication Server

As a result, system administrators who have tuned the procedure cache to the minimal levels prior to implementing replication may need to increase it to accommodate Replication Agent usage. You can see how much memory a Replication Agent is using via the 9204 trace flag (additional information on enabling/disabling Replication Agent trace flags is located in the Replication Agent section).

```
sp_config_rep_agent <db_name>, "trace_log_file", "<filepathname>"
sp_config_rep_agent <db_name>, "traceon", "9204"
-- monitor for a few minutes
sp_config_rep_agent <db_name>, "traceoff", "9204"
```

Generally speaking, the Replication Agent’s memory requirements will be less than normal server’s metadata cache requirements for system objects (sysobjects, syscolumns, etc.). A rule of thumb if sizing a new system for replication might be to use the metadata cache requirements as a starting point.

Metadata Cache

The metadata cache itself is important to replication performance. As will be discussed later, as the Replication Agent reads a row from the transaction log, it needs access to the object’s metadata structures. If forced to read this from disk, the Replication Agent processing will be slowed while waiting for the disk I/O to complete. Careful monitoring of the metadata cache via sp_sysmon during periods of peak performance will allow system administrators to size the metadata cache configurations appropriately.
User Log Cache (ULC)

User (or Private) Log Cache was implemented in Sybase SQL Server 11.0 as a means of reducing transaction log semaphore contention and the number of times that the same log page was written to disk. In theory, a properly sized ULC would mean that only when a transaction was committed, would the records be written to the physical transaction log. One aspect of this that could have had a large impact on the performance of replication server was that this would mean that a single transaction’s log records would be contiguous on disk vs. interspersed with other user’s transactions. This would significantly reduce the amount of sorting that the SQT thread would have to do within the Replication Server.

However, in order to ensure low latency, a decision was made in the design of SQL Server 11.x, that if the OSTAT_REPLICATED flag was on, the ULC would be flushed much more frequently than normal. In fact, in some cases, the system behaves as if it did not have any ULC. As one would suspect, this can lead to higher transaction log contention as well as negating the potential benefit to the SQT thread.

Primary Database Transaction Log

As you would assume, the primary transaction log plays an integral role in replication performance, particularly the speed at which the Replication Agent can read and forward transactions to the Replication Server.

Physical Location

The physical location of the transaction log plays a part in both the database performance as well as replication performance. The faster the device, the quicker Replication Agent will be able to scan the transaction log on startup, recovery and during processing when physical i/o is required. Some installations have opted to use Solid State Disks (SSD’s) as transaction log devices to reduce user transaction times, etc. While such devices would help the Replication Agent, if resources are limited, a good RAID-based log device will be sufficient to enable the SSD to be used as a stable device or other requirement for general server performance (tempdb).

Named Cache Usage

Along with log I/O sizing, binding the transaction log to a named cache can have significant performance benefits. The reason stems from the fact that the Replication Agent cannot read a log page until it has been flushed to disk. While this does happen immediately after the page is full due to recovery reasons, if a named cache is available, the probability is much higher that the Replication Agent can read the log from memory vs. disk. If forced to read from disk, the Replication Agent performance may drop to as low as 1GB/hr.

Application/Database Design

While the above configuration settings can help reduce performance degradation, undoubtedly the best way to improve replication performance from the primary database perspective is the application or primary database design itself.

Multiple Physical Databases

One of the most frequent complaints is that the Replication Agent is not reading the transaction log fast enough, prompting calls for the ability to have more than one Replication Agent per log or a multi-threaded Replication Agent vs. the current threading model. Although some times this can be alleviated by properly tuning the Replication Agent thread, adjusting the above configuration settings, etc., there is a point where the Replication Agent is simply not able to keep up with the logged activity. A classic case of this can be witnessed during large bcp operations (100,000 or more rows) in which the overhead of constructing LTL for each row is significant enough to cause the Replication Agent to begin to lag behind. With the exception of bulk operations, when ever normal OLTP processing causes the Replication Agent to lag behind, the most frequent cause is the failure on the part of the database designers to consider splitting the logical database into two or more physical databases based on logical data groups.

Consider for example, the mythical pubs2 application. Purportedly, it is a database meant to track the sales of books to stores from a warehouse. Let’s assume that 80% of the transactions are store orders. That means the other 20% of the transactions are administering the lists of authors, books, book prices, etc. If maintained in the same database, this extra 20% of the transactions could be just enough to cause a single Replication Agent to lag behind the transaction.
logging. And yet, what would be lost by separating the database into two physical databases – one containing the authors, books, stores and other fairly static information, while the other functions strictly as the sales order processing database? The answer is not much. While some would say that it would involve cross-database write operations, the real answer is not really. Appropriately designed, new authors, books and even stores would be entered into the system outside the scope of the transaction recording book sales. Cross-database referential integrity would be required (for which a trigger vs. declarative integrity may be more appropriate), but even this does not pose a recovery issue except to academics. The real crux of the matter is, is it more important to have a record of a sale to a store in the dependent database even if the parent store record is lost due to recovery, or is it more important to enforce referential integrity at all points and force recovery of both systems?? Obviously, the former is better.

As a result, it makes sense to separate a logical database into several physical databases for the following types of data groupings:

- Application object metadata such as menu lists, control states, etc.
- Application driven security implementations (screen navigation permissions, etc.)
- Static information such as tangible business objects including part lists, suppliers, etc.
- Business event data such as sales records, shipment tracking events, etc.

Not only does this naturally lend itself to the beginnings of shareable data segments reusable by many applications, by doing so, you also will increase the degree of parallelism on the inbound side of Replication Server processing.

**Transaction Processing**

After the physical database design itself, the next largest contributor is how the application processes transactions. An inefficient application not only increases the I/O requirements of the primary database, it also can significantly degrade replication performance. Several of the more common inefficiencies are discussed below.

**Avoid Repeated Row Re-Writes**

One of the more common problems brought about by forms-based computing is that the same row of data may be inserted and then repeatedly updated by the same user during the same session. A classic scenario is the scenario of filling out an application for loans or other multi-part application process. A second common scenario is one in which fields in the “record” are filled out by database triggers, including user auditing information (last_update_user), order totals, etc. While some of this is unavoidable to insure business requirements are met, it may add extra work to the replication process. Consider the following mortgage application scenario:

User inserts basic loan applicant name, address information
As user transitions to next screen for property info, the info is saved to the database.
User adds the property information (stored in same database table).
As user transitions to the next screen, the property information is saved to the database.
User adds dependent information (store in same table in denormalized form)
User hits save before asking credit info (not stored in same table)

Just considering the above scenario, the following database write operations would be initiated by the application:

```
insert loan_application (name, address)
update loan_application (property info)
update loan_application (dependent info)
```

Now, consider the actual I/O costs if the database table had a trigger that recorded the last user and datetime that the record was last updated.

```
insert loan_application (name, address)
update loan_application (lastuser, lastdate)
update loan_application (property info)
update loan_application (lastuser, lastdate)
update loan_application (dependent info)
update loan_application (lastuser, lastdate)
```

As a result, instead of a single record, the Replication Agent must process 6 records – each of which will incur the same LTL translation, Replication Server normalization/distribution/subscription processing, etc. On top of which, consider what happens at the replicate (if triggers are not turned off for the connection) – local trigger firings at the replicate are bolded.
Some may question the reality of such an example. It is real. While remaining unnamed, one of Sybase’s mortgage banking customers had a table containing 65 columns requiring 8-10 application screens before completely filled out. After each screen, rather than filling out a structure/object in memory, each screen saved the data to the database. During normal database processing, this led to an extremely high amount of contention within the table made worse by the continual page splitting to accommodate the increasing row size. Replication was enabled in a Warm-Standby configuration for availability purposes. Although successful, you can guess the performance implications within Replication Server from such a design.

**Batch Process/Bulkcopy Concurrency**

In some cases, the lack of concurrency at the primary translates directly into replication performance problems at the replicate. Consider for example, the ever-common bulkcopy problem. “Net gouge” for years has stated that during slow bcp, the bcp utility translates the rows of data into individual insert statements. Consequently, people find it surprising that Replication Server has difficulty keeping up. In the first place, the premise is false. While slow bcp is several orders of magnitude slower than fast bcp, it is still a bulk operation and consequently does not validate user-defined datatypes, declarative referential integrity, check constraints nor fire triggers. As a result, of course, it is still several orders of magnitude faster than individual insert statements that Replication Server will use at the replicate.

Now, consider the scenario of a nightly batch load of three tables. If bcp’d sequentially using slow bcp, it may take 4-6 hours to load the data. Unfortunately, when replication is implemented, the batch process at the replicate requires 8-10 hours to complete, exceeding the time requirements and possibly encroaching on the business day. Checking the replicated database during this time shows extremely little CPU or I/O utilization and the maintenance user process busy only a fraction of the time. All the normal “things” are tried and even parallel DSI’s are implemented – all to no avail. Customer decides that Replication Server just can’t keep up.

The reality of the above scenario is that several problems contributed to the poor performance:

- The bcp probably did not use batching (-b option) and as a result was loaded in a single transaction. As a result, the Replication Server could only ever use a single DSI, no matter how many were configured, as it had to apply it as a single transaction.
- Further, it would be held in the inbound queue until the commit record was seen by the SQT thread – as a large transaction, this may incur multiple scans of the inbound queue to recreate the transaction records due to filling the SQT cache.
- Lack of batch size in the bcp (-b option) more than likely drove Replication Server to use large transaction threads – while this may have reduced the overall latency in one area due to not having to wait for the DSI to see the commit record, it also meant that Replication Server only considered a small number of threads preserved for large transactions.
- Replication Agent probably was not tuned (batching and ltl_batch_size) as will be discussed in the next section.
- Even if bcp batching were enabled, by sequentially loading the tables, concurrent DSI threads would suffer a high probability of contention, especially on heap tables or indexes – due to working on a single table. If attempting to use parallel DSI’s, this will force the use of the less efficient default serialization method of “wait_for_commit”.

Some of the above will be addressed in the section specific to Parallel DSI tuning, however, it should be easy to see how the Replication Server lagged behind. It also illustrates a very key concept:
Key Concept #1: The key to understanding Replication Server performance is understanding how the entire Replication System is processing your transaction.

Now, consider what would likely happen if the following scenario was followed for the three tables:

- All three tables were bcp’d concurrently using a batch size of 100.
- Replication Server was tuned to recognize 1,000 statements as a large transaction vs. 100.
- Replication Agent was tuned appropriately.
- DOL/RLL locking at the replicate database.
- DSI serialization was set to “none” (see Parallel DSI tuning section).
- Optionally, tables partitioned (although not necessary for performance gains – if partitioned, DOL/RLL is a must).

Would the SQT cache size fill? Probably not. Would the Parallel DSI’s be used/effective? Most assuredly. Would Replication Server keep up? It probably would still lag, but not as much. At the primary, it now may take only 2 hours to load the data (arguably less if not batching) and 3 hours at the replicate.

The same scenario is evident in purge operations. Typically, a single purge script begins by deleting masses of records using SQL joins to determine which rows can be removed. The problem is of course that this is identical from a replication perspective as a bcp operation – a large transaction with no concurrency. An alternative approach in which a delete list is generated and then used to cursor through the main tables using concurrent processes may be more recoverable, cause less concurrency problems at the primary and improve replication throughput. This leads to a second key concept:

Key Concept #2: The optimal primary transaction profile for replication is concurrent users updating/inserting/deleting small numbers of rows per transaction spread throughout different tables.

That does not mean low volume! It can be extremely high volume. It just means it is better from a replication standpoint for 10 processes to delete 1,000 rows each in batches of 100 than for a single process to delete 100,000 rows in a single transaction. Accordingly, the best way to improve replication performance of large batch operations is to alter the batch operation to use concurrent smaller transactions vs. a single large transaction.

Procedure vs. Table Replication

First, there is a common misconception that you cannot replicate both procedures and tables modified by replicated procedures. This is partially based on the following paragraph:

“If you use function replication definitions, do not attempt to replicate affected data using table replication definitions and subscriptions. If the stored procedures are identical, they will make identical changes to each database. If the affected tables are also replicated, duplicate updates would result.”

- page 9-3 in Replication Administration/11.5

However, consider the following paragraphs:

In replicating stored procedures via applied functions, it may be advisable to create table replication definitions and subscriptions for the same tables that the replicated stored procedures will affect. By doing this you can ensure that any normal transactions that affect the tables will be replicated as well as the stored procedure executions.

However, DML inside stored procedures marked as replicated is not replicated. Thus, in this case, you must subscribe to the stored procedure even if you also subscribe to the table.

- page 3-145 in Replication Reference/11.5
Confused?? A lot of people are. What it really refers to is if you replicate a procedure, the DML changes within the procedure will not be replicated, no matter what. The way this is achieved is that normally, as a DML statement is logged, if the object’s OSTAT_REPLICATE flag is set, then the ASE logger sets the transaction log record’s LSTAT_REPLICATED flag. For a stored procedure, this means that the stored procedure receives the LSTAT_REPLICATED flag, and the ASE logger does not mark any DML records for replication until after that procedure execution has completed. Attempting to force both to be replicated (i.e. executing a replicated procedure in one database with replicated DML modifications in another) could lead to database inconsistencies. The only way to force this replication is to a) replicate a procedure call in one database and b) that procedure modify data in a table that is also replicated in another database. This would allow both to be replicated as two independent log threads would be involved. The one that would be evaluating the DML for replication would not be aware that the DML was even inside a procedure that was also replicated.

Which brings us to the point the second reference was making. The second reference stated that it “may be advisable to create table replication definitions and subscriptions for the same tables…”. The reason for this is exactly the fact that DML within a procedure is NOT replicated – and needs reverse logic to understand the impact. Consider the scenario of New York, London Tokyo, San Francisco and Chicago all sharing trade data. A procedure at New York is executed at the close of the day to update the value of mutual funds based on the closing market position of the funds stock contents. All the other sites subscribe to the mutual fund portfolio table. Now, consider what would happen if only San Francisco and Chicago subscribed to the procedure execution. Neither London nor Tokyo would ever receive the update mutual fund values!!! Why?? Since the DML within the replicated procedure is not marked for replication, the Replication Agent would only forward the procedure execution log records and NOT the logged mutual fund table modifications. Since neither subscribed to the procedure, they would not receive anything. Which brings us to the following concept:

**Key Concept #3:** If replicating a procedure as well as the tables modified by the procedure, any replicate that subscribes to one should also subscribe to the other to avoid data inconsistency.

A notable exception to that is that if replicating to a data warehouse, the data warehouse may not want to subscribe to a purge or archive procedure executed on the OLTP system.

Now that we have cleared that matter up and we understand that we **can** replicate procedures and tables they affect simultaneously, the question is how does this affect performance. The answer – as in all performance questions – is: “It depends”. Replicating procedures can both improve replication performance as well as degrade replication performance. The former is often referenced in replication design documents, and consequently, will be discussed first.

Consider a normal retail bank. At a certain part of the month, the bank updates all of the savings accounts with interest calculated on the average daily balance during that month. This literally can be tens of thousands to hundreds of thousands of records. If replicating the savings account table to regional offices, failover sites, or elsewhere, this would mean the following:

1. The Replication Agent would have to process and send to the Replication Server every individual account record.
2. The account records would have to be saved to the stable device.
3. Each and every account record would be compared to subscriptions for possible distribution.
4. The account records would have to be saved again to the stable device – once for each destination.
5. Each account record would have to update as individual updates at each of the replicates

The impact would be enormous. First, beyond a doubt, the Replication Agent would lag significantly. Secondly, the space requirements and the disk I/O processing time would be nearly insurmountable. Third, the CPU resources required for tens to hundreds of thousands of comparisons are enormous. And lastly, the time it would take to process that many individual updates would probably exceed the required window.

How would replicating stored procedures help?? That’s easy to see. Rather than updating the records via a static SQL statement at the primary, a stored procedure containing the update would be executed instead. If this procedure were replicated, then the Replication Agent would only have to read/transfer a single log record to the Replication Server, which in turn would only have to save/process that single record. The difference could be hours of processing saved –
and the difference between a successful replication implementation or one that fails due to the fact the replicate can never catch up due to latency caused by excessive replication processing requirements.

Key Concept #4: Any business transaction that impacts a large number of rows is a good candidate for procedure replication, along with very frequent transactions that affect a small set of rows.

So, if stored procedures are can reduce the disk I/O and Replication Server processing, how can replicating a stored procedure negatively affect replication? The answer is two reasons: 1) the latency between begin at the primary and commit at the replicate; and 2) extreme difficulty in achieving concurrency in delivering replicated transactions to the replicate once the replicated procedure begins to be applied.

Let’s discuss #1. Remember, Replication Server only replicates committed transactions. Now, using our earlier scenario of our savings account interest procedure, let’s assume that the procedure takes 4 hours to execute. We would see the following behavior:

1. Procedure begins execution at 8:00pm and implicitly begins a transaction.
2. Replication Agent forwards procedure execution to RS nearly immediately.
3. RS SQT thread caches execution record until the procedure completes execution and the completion record is received via the implicit commit.
4. At midnight the procedure completes execution.
5. Within seconds, the Replication Agent has forwarded the commit record to RS and RS has moved the replicated procedure to the Data Server Interface (DSI).
6. The DSI begins executing the procedure at the replicate shortly after midnight
7. Assuming all things being equal, the procedure will complete at the replicate at 4:00am

Consequently, we have a total of 8 hours from when the process begins until it completes at the replicate, and 4 hours from when it completes at the primary until it completes at the replicate. This timeframe might be acceptable to some businesses. However, what if the procedure took 8 hours to execute? Basically, the replicate would not be caught up for several hours after the business day began – which may not be acceptable for some systems such as stock trading systems with more real time requirements.

That explains the latency issue – what of the concurrency? This requires a bit of thinking, but consider this: while the procedure is executing at the primary, concurrent transactions by customers (i.e. ATM withdrawals) may also be executing in parallel. Since they would commit far ahead of the interest calculation procedure, they would show up at the replicate within a reasonable amount of time. Assuming this pattern continues even after the procedure completes (i.e. checks clearing from business retailers), the following would happen:

1. Procedure completes at primary. It is followed by a steady stream of other transactions – possibly even a batch job requiring 3 hours to run.
2. Since RS guarantees commit order at the replicate, RS processes the transactions in commit order and internally forwards them to the DSI thread for execution at the replicate.
3. If only using a single DSI, the follow-up transactions would not even begin until the interest procedure had committed – some 4 hours later. If multiple DSI’s and no contention, the DSI would have to ensure that the follow-up transactions did not commit first and would do so by not sending the commit record for the follow-up transactions until the procedure had finished.
4. Due to contention, the replicated batch process may not even begin execution via a parallel DSI until the replicated interest procedure committed.

In addition to the fact that transactions committed shortly after the interest procedure suddenly have a 4 hour latency attached, the question that should come up is “Can the Replication Server catch up?” Let’s arbitrarily assume that due to database physical design that the DSI threads at maximum efficiency are only able to apply 500 transactions/hour. Now, let’s assume that the bank experiences a steady stream of 350 transactions/hour. Normally, this is not a problem – RS is able to keep up as it has a comfortable surge capability of 150 transactions/hour. However, now consider bullet
#3 above. Due to transaction grouping rules and command batching configuration settings, etc., a full 4 hours (or 1,400) of transactions would be backed up in the RS stable queue. It will take the RS 3 hours simply to apply them and during those 3 hours, another 1050 transactions would arrive. Using the reserve capacity of 150 transactions/hour, it would take nearly 10 hours for the RS to be fully caught up. So, …

Key Concept #5: Replicated procedures with long execution times may increase latency by delaying transactions from being applied at the replicate. The CPU and disk I/O savings with RS need to be balanced against this before deciding to replicate any particular procedure.

As a result, it may be advisable to actually replicate the row modifications. This could be done by not replicating the procedure but have the procedure cursor through each account. This would be the same as atomic updates, each a separate transaction (after all, there is no reason why Annie Aunt’s interest calculation needs to be part of the same transaction as Wally the Walrus – but whether or not that is how it is done at the primary, at the replicate they would be all part of the same transaction due to the fact the entire procedure would be replicated and applied within the scope of a single transaction.). While it may take RS several hours to catch up, entirely on the replicate – it just might be less than the latency incurred due to replicating the procedure.

Is there a way around this problem without replicating the individual row updates? Possibly. In this particular example, assuming the average daily balance is stored on a daily basis (or other form so that changes committed out of order do not affect the final result), a multiple DSI approach could be used to the replicate system, in which the replicated procedure could use it’s own dedicated connection to the replicates. Consequently, the Replication Server would be able to keep up with the ongoing stream of transactions, while concurrently executing the procedure. However, this would only work in such places where having a transaction that committed at the primary after the interest calculation but commits before it at the replicate does not cause a disparity in the balance. More will be discussed about this approach in a later section after the discussion about Parallel DSI’s.
Replication Agent Processing

Why is the Replication Agent so slow???

Frequently, comments will be made that the ASE Rep Agent is not able to keep up with logging in the ASE. For most normal user processing, a properly tuned Rep Agent on a properly tuned transaction log/system will have no trouble keeping up. This is especially true if the bulk of the transactions originate from GUI-base user screens since such applications naturally tend to have an order of magnitude more reads than writes. However, for systems with large direct electronic feeds or sustained bulk loading, Replication Agent performance is crucial. At this writing, a complete replication system based on Replication Server 12.0 is capable of maintaining over 2GB/Hr from a single database in ASE 11.9.3 using normal RAID devices (vs. SSD’s). In this section we will be examining how the Replication Agent works – and in particular, two bottlenecks quite easily overcome by adjusting configuration parameters. As mentioned earlier, since this paper does not yet address many of the aspects of heterogeneous replication, this section should be read in the context of the ASE Replication Agent thread. However, the discussions on Log Transfer Language and the general Rep Agent communications are common to all replication agents as all are based on the replication agent protocol supported by Sybase.

Secondary Truncation Point Management

Every one knows that the ASE Replication Agent maintains the ASE secondary truncation point, however, there are a lot of misconceptions about the secondary truncation point and the Replication Agent, including:

- The Replication Agent looks for the secondary truncation point at startup and begins re-reading the transaction log from that point.
- The Replication Agent cannot read past the primary truncation point.
- “Zero-ing the LTM” resets the secondary truncation point back to the beginning of the transaction log.

As you would guess from the previous sentence, these are not necessarily accurate. In reality, there is a lot more communication and control from the Replication Server in this process than realized. The sequence of events is more along the lines of:

1. The Replication Agent logs in to the Replication Server and requests to “connect” the source database (via the “connect source” command) and provides a requested LTL version. Replication Server responds with the negotiated LTL version and upgrade information.
2. The Rep Agent asks the Replication Server who the maintenance user is for that database. The Replication Server looks the maintenance user up in the rs_maintusers table in the RSSD database and replies to the Rep Agent.
3. The Rep Agent asks the Replication Server where the secondary truncation point should be. The Replication Server looks up the locater in the rs_locaters table in the RSSD database and replies to the Rep Agent.
4. The Rep Agent starts scanning from the location provided by the Replication Server
5. The Replication Agent scans for a configurable number (scan_batch_size) log records.
6. After reaching scan_batch_size log records, the Replication Agent requests a new secondary truncation point for the transaction log. When this request is received, the Replication Server responds with the cached locater which contains the log page containing the oldest open transaction received from the Replication Agent. In addition, the Replication Server writes this cached locater to the rs_locaters table in the RSSD.
7. The Rep Agent moves the secondary truncation point to the log page containing the oldest open transaction received by Replication Server.
8. Repeat step 5.

An interaction diagram for this might look like the following:
The key elements to get out of this are fairly simple:

- Keep the RSSD as close as possible to the RS
- Every scan\_batch\_size rows, the Rep Agent stops forwarding rows to move secondary truncation point.
- The secondary truncation point is set to the oldest open transaction received by Replication Server – which may be the same as the oldest transaction in ASE (syslogshold) or it may be an earlier transaction as the Rep Agent has not yet read the commit record from the transaction log.

Regarding the first, if you notice, most of the time that the Rep Agent asks the RS for something, the RS has to check with the RSSD – or update the RSSD (i.e. the locater). So, don’t put the RSSD too far (network wise) from the RS. The best place is on the same box and have the primary network listener for the RSSD ASE be the TCP loopback port (127.0.0.1)

The second can be overcome with a willingness to absorb more log utilization. The default scan\_batch\_size is 1,000 records. As anyone who has read the transaction log will tell you, 1,000 log records happen pretty quickly. The result is that the Rep Agent is frequently moving the secondary truncation point. Benchmarks have show that raising scan\_batch\_size can increase replication throughput significantly. For example, at an early Replication Server customer, setting it to 20,000 improved overall RS throughput by 30%. Of course, the tradeoff to this is that the secondary truncation point stays at a single location in the log – translates to a higher degree of space used in the transaction log. In addition, database recovery time as well as replication agent recovery time will be lengthened as the portion of the transaction log that will be rescanned at database server and replication agent startup will be longer.

**Rep Agent LTL Generation**

The protocol used by sources to replication server is called Log Transfer Language (LTL). Any agent that wishes to replicate data via Replication Server must use this protocol, much the same way that RS must use SQL to send transactions to ASE. Fortunately, this is a very simple protocol with very few commands. The basic commands are listed in the table below.
<table>
<thead>
<tr>
<th>LTL Command</th>
<th>Subcommand</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>connect source</td>
<td></td>
<td>request to connect a source database to the replication system in order to start forwarding transactions.</td>
</tr>
<tr>
<td>get maintenance user</td>
<td></td>
<td>request to retrieve maintenance user name to filter transactions applied by the replication system.</td>
</tr>
<tr>
<td>get truncation</td>
<td></td>
<td>request to retrieve a log pointer to the last transaction received by the Replication Server.</td>
</tr>
<tr>
<td>distribute</td>
<td>begin transaction</td>
<td>Used to distribute begin transaction statements</td>
</tr>
<tr>
<td></td>
<td>commit/rollback</td>
<td>Used to distribute commit/rollback statements</td>
</tr>
<tr>
<td></td>
<td>applied</td>
<td>Used to distribute insert/update/delete SQL statements</td>
</tr>
<tr>
<td></td>
<td>execute</td>
<td>Used to distribute both replicated procedures as well as request functions</td>
</tr>
<tr>
<td></td>
<td>sqlddl append</td>
<td>Used to distribute DDL to WS systems</td>
</tr>
<tr>
<td></td>
<td>dump</td>
<td>Used to distribute the dump database/transaction log SQL commands</td>
</tr>
<tr>
<td></td>
<td>purge</td>
<td>Used during recovery to notify Replication Server that previously uncommitted transactions have been rolled back.</td>
</tr>
</tbody>
</table>

A sample of what LTL looks like is as follows:

distribute @origin_time="Apr 15 1988 10:23:23.001PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000001, @tran_id=0x000000000000000000000001 begin transaction 'Full LTL Test' -- added for clarity
distribute @origin_time="Apr 15 1988 10:23:23.002PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000002, @tran_id=0x000000000000000000000001 applied 'ltltest'.rs_insert yielding after @intcol=1, @smallintcol=1, @tinyintcol=1, @decimalcol=.12, @numericcol=2.1, @identitycol=1, @floatcol=3.2, @realcol=2.3, @charcol='first insert', @varcharcol='first insert', @text_col=hastext always_rep, @moneycol=$0.56, @smallmoneycol=$0.56, @datetimecol='14-15-1988 10:23:23:001PM', @smalldatetimecol='Apr 15 1988 10:23:23.002PM', @imagecol=0xaabbccddeeff, @varbinarycol=0xa0112233445566778899, @imagecol=hastext rep_if_changed, @bitcol=1 -- added for clarity
distribute @origin_time="Apr 15 1988 10:23:23.003PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000003, @tran_id=0x000000000000000000000001 applied 'ltltest'.rs_writetext append first last changed with log textlen=30 @text_col=-- added for clarity
distribute @origin_time="Apr 15 1988 10:23:23.004PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000004, @tran_id=0x000000000000000000000001 applied 'ltltest'.rs_writetext append first last changed with log textlen=119 @imagecol=-- added for clarity
distribute @origin_time="Apr 15 1988 10:23:23.005PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000005, @tran_id=0x000000000000000000000001 applied 'ltltest'.rs_writetext append last @imagecol=-- added for clarity
distribute @origin_time="Apr 15 1988 10:23:23.006PM", @origin_qid=0x0000000000000000000000000000000000000000000000000000000000000006, @tran_id=0x000000000000000000000001 applied 'ltltest'.rs_update yielding before @intcol=1, @smallintcol=1, @tinyintcol=1, @decimalcol=.12, @numericcol=2.1, @identitycol=1, @floatcol=3.2, @realcol=2.3, @charcol='first insert', @varcharcol='first insert', @text_col=hastext notrep always_rep, @moneycol=$1.56, @smallmoneycol=$0.56, @datetimecol='Apr 15 1988 10:23:23:002PM',
Although it looks complicated, the above is fairly simple – all of the above are distribute commands for a part of a transaction comprised of multiple SQL statements. The basic syntax for a distribute command for a DML operation is as follows:

```
distribute <commit time> <OQID> <tran id> applied <table>.<function>
yielding [before <col name>=<value> [, <col name>=<value>, ...]]
[after <col name>=<value> [, <col name>=<value>, ...]]
```

As you could guess, the distribute command will make up most of the communication between the Rep Agent and the Rep Server. Looking closely at what is being sent, you will notice several things:

- The appropriate replicated function (rs_update, rs_insert, etc.) is part of the LTL
- The column names are part of the LTL

The latter is not always the case as some heterogeneous Replication Agents can cheat and not send the column names (assuming Replication Definition was defined with columns in same order or through a technique called “structured tokens”. Although currently beyond the scope of this paper, this is achieved by the Replication Agent directly accessing the RSSD to determine replication definition column ordering. This improves Replication Agent performance by reducing the size of the LTL to be transmitted and allowing the Replication Agent to drop columns not included in the replication definition. This information, once retrieved, can be cached for subsequent records. Currently, the ASE Replication Agent does not support this interface. However, in general, the LTL distribute command illustrated above does leave us with another key concept:

**Key Concept #6: Ignoring subscription migration, the appropriate replication function rs_insert, rs_update, etc., is determined by the replication agent from the transaction log. The DIST/SRE determines which functions are sent according to migration rules, while the DSI determines the SQL language commands for that function.**

Having determined what the Replication Agent is going to send to the Replication Server, the obvious question is how does it get to that point. The answer is based on two separate processes – the normal ASE Transaction Log Service (XLS) and the Rep Agent. The process is similar to the following:

1. (XLS) The XLS receives a log record to be written from the ASE engine
2. (XLS) The XLS checks object catalog to see if logged object’s OSTAT_REPLICATED bit is set.
3. (XLS) If not, the XLS simply skips to writing the log record. If it is set, then the XLS checks to see if the DML logged event is nested inside a stored procedure that is also replicated.
4. (XLS) If so, the XLS simply skips to writing the log record. If not, then the XLS sets the log record’s LSTAT_REPLICATE flag bit
5. (XLS) The XLS writes the record to the transaction log
6. (RA) Some arbitrary time later, the Rep Agent reads the log record
7. (RA) The Rep Agent checks to see if the log record’s LSTAT_REPLICATE bit is set.
8. (RA) If so, Rep Agent proceeds to LTL generation. If not, the Rep Agent determines if the log record is a “special log record” such as begin/commit pairs, dump records, etc.
9. (RA) If not, the Rep Agent can simply skip to the next record. If it was, the Rep Agent proceeds with constructing LTL.
10. (RA) The Rep Agent checks to see if the operation was an update. If so, it also reads the next record to construct the before/after images.
11. (RA) The Rep Agent checks to see if the logged row was a text chain allocation. If so, it reads the text chain to find the TIPSA. This TIPSA is then used to find the data row for the text modification. The data row for writetext is then constructed in LTL. Then the text chain is read and constructed into LTL chunks of text/image append functions.

12. (RA) LTL Generation begins. Rep Agent checks it’s own schema cache (part of proc cache) to see if the logged object’s metadata is in cache. If not, it reads the objects metadata from system tables (syscolumns).

13. (RA) Rep Agent constructs LTL statement for the logged operation

14. (RA) If ‘batch_ltl’ parameter is false (default), the Rep Agent passes the LTL row to the Rep Server using the distribute command. If ‘batch_ltl’ is true, the Rep Agent waits until the LTL buffer is full prior to sending the records to the Rep Server.

This process is illustrated below. The two services are shown side-by-side due to the fact that they are independent threads within the ASE engine and execute in parallel on different log regions. This latter is due to the fact that the Rep Agent can only read flushed log pages (flushed to disk), consequently, it will always be working on a different log page than the XLS service.

The following list summarizes this into key elements how this affects replication performance and tuning.

- Replication Agent uses ASE procedure cache to maintain object metadata (schema cache) for constructing LTL as well as tracking transactions (transaction cache). As a result, more procedure cache may be necessary on systems with a lot of activity on large numbers of tables. In addition, careful monitoring of the system metadata cache to ensure that physical reads to system tables are not necessary.
- LTL batching can significantly improve Rep Agent processing as it can scan more records prior to sending the rows to the Rep Server (effectively a synch point in Rep Agent processing).
- Replicating text/image columns can slow down Rep Agent processing of the log due to reading the text/image chain.
- Marking objects for replication that are not distributed (i.e. for which no subscriptions or Warm Standby exists) has a negative impact on Rep Agent performance as it must perform LTL generation needlessly. In
addition, these “extra” rows will consume space in the inbound stable queue and valuable CPU time for the distributor thread.

- Procedure replication can improve Rep Agent throughput by reducing the number of rows for which LTL generation is required. For example, if a procedure modifies 1,000 rows, replicating the table will require 1,000 LTL statements to be generated (and compared in the distributor thread). By replicating the procedure only a single LTL statement will need to be generated and processed by Replication Server.

Key Concept #7 - In addition to Rep Agent tuning, the best way to improve Rep Agent performance is to minimize its workload. This can be achieved by not replicating text/image columns where not necessary and ensuring only objects for which subscriptions exist are marked for replication. In addition, replicating procedures for large impact transactions could improve performance significantly.

Replication Agent Communications

The Rep Agent connects to the Replication Server in “PASSTHRU” mode. A common question is “What does it mean by passthru mode?” The answer lies in how the server responds to packets. In passthru mode, a client can send multiple packets to the server without having to wait for the receiver to process them fully. However, they do have to synchronize periodically for the client to receive error messages and statuses. A way to think of it is that the client can simply start sending packets to the server and as soon as it receives packet acknowledgement from the TDS network listener, it can send the next packet. Asynchronously, the server can begin parsing the message. When the client is done, it sends an End-Of-Message (EOM) packet that tells the server to process the message and respond with status information. By contrast, typical client connections to Adaptive Server Enterprise are not passthru connections, consequently, the ASE server processes the commands immediately on receipt and passes the status information back to the client.

This technique provides the Rep Agent/Rep Server communication with a couple of benefits:

- Rep Agent doesn’t have to worry if the LTL command spans multiple packets.
- The destination server can begin parsing the messages (but not executing) as received, achieving greater parallelism between the two processes

If the Rep Agent configuration batch_ltl is true, Rep Agent will batch LTL to optimize network bandwidth (although the TDS packet size is not yet configurable). If not, as each LTL row is created, it is sent to the Rep Server. In either case, the messages are sent via passthru mode to the Rep Server. Every 16K, the Rep Agent synchs with the Rep Server by sending an EOM (at an even command boundary – EOM can not be placed in the middle of an LTL command).

Replication Agent Troubleshooting

There are several commands for troubleshooting the Rep Agent. At a basic level, sp_help_rep_agent can help track where in the log and how much of the log the Rep Agent is processing. However, for tougher problems, several trace flags exist.

<table>
<thead>
<tr>
<th>Trace Flag</th>
<th>Trace Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>9201</td>
<td>Traces LTL generated and sent to RS</td>
</tr>
<tr>
<td>9202</td>
<td>Traces the secondary truncation point position</td>
</tr>
<tr>
<td>9203</td>
<td>Traces the log scan</td>
</tr>
<tr>
<td>9204</td>
<td>Traces memory usage</td>
</tr>
</tbody>
</table>

Output from the trace flags is to the specified output file. The trace flags and output file are specified using the normal sp_config_rep_agent procedure as in the following:

```
sp_config_rep_agent <db_name>, "trace_log_file", "<filepathname>"
```
However, tracing the Rep Agent has a considerable performance impact as the Rep Agent must also write to the file. In the case of LTL tracing (9201), this can be considerable. As a result, Rep Agent trace flags should only be used when absolutely necessary.
Replication Server General Tuning

How much resources will Replication Server require?

The above is a favorite question – and a valid one – of nearly every system administrator tasked with installing a Replication Server. The answer, of course, is all depends – it depends on the transaction volume of the primary sites, how many replicate databases involved and how much latency the business is willing to tolerate.

The object of this section is to cover basic Replication Server tuning issues. It should be noted that these are general recommendations that apply to many situations, however, your specific business or technology requirements may prevent you from implementing the suggestions completely. Additionally, due to environment specific requirements, you may achieve better performance with different configurations than those mentioned here.

The recommendations in this section are based on the assumption of an enterprise production system environment and consequently are significantly higher than the software defaults.

Replication Server/RSSD Hosting

A common mistake is placing the RSSD database in one of the production systems being replicated to/from. While this in itself has other issues, one of the main problems stemming from this is that this frequently places the RSSD across the network from the Replication Server host. As you saw earlier in the Rep Agent discussion on secondary truncation point management, the volume of interaction between the Replication Server and RSSD can be substantial – just in processing the LTL. Add the queue processing, catalog lookups and other RSSD accesses increase this load considerably. This leads a critical key performance concept for RSSD hosting:

Key Concept #8: Always place the RSSD database in an ASE database engine on the same physical machine host as the Replication Server. In addition, make sure that the first network addresses in the interfaces file for that ASE database engine are 'localhost' (127.0.0.1) entries.

The latter part of the concept may take a bit of explaining. If you took in the host file on any platform (/etc/hosts for Unix; %systemroot%\system32\drivers\etc\hosts for WindowsNT), you should see an entry similar to:

127.0.0.1 localhost #loopback on IBM RS6000/AIX

In addition the Network Interface Card (NIC) IP addresses, the localhost IP address refers to the host machine itself. The difference is in how communication is handled when addressing the machine via the NIC IP address or the localhost IP address. If using the NIC IP address, packets destined for the machine name may not only have to hit the NIC card, but may also require NIS lookup access or other network activity (routing) that result in minimally the NIC hardware being involved. On the other hand, when using the localhost entry, the TCP/IP protocol stack knows that no network access is really required. As a result, the protocol stack implements a “TCP loopback” in which the packets are essentially routed between the two applications only using the TCP stack. An illustration of this is shown below:
As you could guess, this has substantial performance and network reliability improvements over using the network interface. Typically, this can be implemented by modifying the Sybase interfaces file to include listeners at the localhost address. However, these must be the first addresses listed in the interfaces file in order for this to work. For example:

NYPROD

master tcp /dev/tcp localhost 5000
master tcp /dev/tcp nymachine 5010
query tcp /dev/tcp localhost 5000
query tcp /dev/tcp nymachine 5010

NYPROD_RS

master tcp /dev/tcp localhost 5500
master tcp /dev/tcp nymachine 5510
query tcp /dev/tcp localhost 5500
query tcp /dev/tcp nymachine 5510

Note that many of today’s vendors have added the ability for the TCP stack to automatically recognize the machine’s IP address(es) and provide similar functionality without specifically having to use the localhost address. Even so, there may be a benefit to using the localhost address on machines in which the RSSD is co-hosted with application databases and the RS would have to contend with application users for the network listener with ASE. By using the localhost address, RS queries to the RSSD may by-pass the “traffic jam” on the network listener used by all the other clients. A word of warning. On some systems, implementing multiple network listeners – especially one on localhost – could result in severe performance degradation (especially when attempting large packet sizes). One such is AIX 4.3 with ASE 11.9.2.

Additionally, the machine should have the following minimal specifications (NOTE: The following specifications are not the bare minimums, but probably are the minimum a decent production system should consider to avoid resource contention or swapping):

<table>
<thead>
<tr>
<th>Resources</th>
<th>Recommendation</th>
</tr>
</thead>
<tbody>
<tr>
<td># of CPU’s</td>
<td>1 for each RS and ASE installed on box plus 1 for OS and monitoring (RSM). (min 2, 3 preferred)</td>
</tr>
<tr>
<td>Memory</td>
<td>64-128MB for ASE plus memory for each RS (64MB min) and operating system (32-64MB). Min of 256MB</td>
</tr>
<tr>
<td>Disk Space</td>
<td>ASE requirements plus RAID 0+1 device for stable queues – separate controllers/disk for stable queues – separate controllers/disk for ASE and RS</td>
</tr>
<tr>
<td>Network</td>
<td>Switched fast Ethernet, ATM or better (gigabit Ethernet)</td>
</tr>
</tbody>
</table>
The rationale behind these recommendations will be addressed in the discussions in the following sections.

Author’s Note: As of this writing, there should be no licensing concern to restricting the use of an ASE for the RSSD.

Each Replication Server license includes the ability to implement a “limited use” ASE solely for the purpose of hosting the RSSD ("limited use" means that the ASE server could only be used the RSSD – no application data, etc. permitted). Consequently, each RS implemented at a site could have its own ASE specifically for the RSSD.

RS Generic Tuning

Generally speaking, the faster the disk I/O subsystem and the more memory, the faster Replication Server will be. In the following sections Replication Server resource usage and tuning will be discussed in detail.

Replication Server Memory Utilization

A common question is how much memory is necessary for Replication Server performance to achieve desired levels. The answer really depends on several factors:

1. Transaction volume from primary systems
2. Number of primary and replicate systems
3. Number of parallel DSI threads
4. Number of replicated objects (repdef’s, subscriptions, etc.)

Of course, life isn’t that simple. Based on the above considerations, you have to adjust several configuration settings within the Replication Server for optimal settings with certain minimums required based on your configuration. Some of these are documented below:

<table>
<thead>
<tr>
<th>Parameter (Default)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_threads (50)</td>
<td>The number of internal processing threads, client connection threads, daemons, etc. The old formula for calculating this was ($PDB \times 7) + ($RDB \times 3) + 4 + (num_client_connections) + (parallel DSI's) + (subscriptions) + ... A simpler and more accurate formula would be ($Connections + 7) + num_client_connections + (parallel DSI's) + num_concurrent_subs</td>
</tr>
<tr>
<td>num_msgqueues (178)</td>
<td>Specifies the number of OpenServer message queues that will be available for the internal RS threads to use. The old formula for calculating this was: 2 + ($PDB \times 4) + ($RDB \times 2) + ($Direct_Routes) However, given that this number must always be larger than num_threads, a simpler formula would be: num_threads*2</td>
</tr>
<tr>
<td>num_msgs (45,568)</td>
<td>The number of messages that can be enqueued at any given time between RS threads. The default settings suggest a 1:256 ratio, although a 1:512 may be more advisable.</td>
</tr>
<tr>
<td>num_stable_queues (32)</td>
<td>Minimum number of stable queues. This should be at least twice the number of database connections + num_concurrent_subs.</td>
</tr>
<tr>
<td>num_client_connections (30)</td>
<td>Number of isql, RSM and other client connections (non-Rep Agent or DSI connections). The default of 30 is probably a little high for most systems - 20 may be a more reasonable starting point</td>
</tr>
<tr>
<td>num_mutexes (128)</td>
<td>Used to control access to connection and other internal resources. The old formula for calculating this was: 12 + ($PDB \times 2) + ($RDB) However, since each thread uses a mutex, this number must be</td>
</tr>
<tr>
<td>Parameter (Default)</td>
<td>Explanation</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>sqt_max_cache_size</td>
<td>Maximum SQT (Stable Queue Transaction) interface cache memory (in bytes) for each connection (Primary and Replicate).</td>
</tr>
<tr>
<td>(131,072)</td>
<td></td>
</tr>
<tr>
<td>dsi_sqt_max_cache</td>
<td>Maximum SQT interface cache memory for a specific database connection, in bytes. The default, 0, means the current setting of the sqt_max_cache_size parameter is used as the maximum cache size for the connection.</td>
</tr>
<tr>
<td>(0)</td>
<td></td>
</tr>
</tbody>
</table>

Each of these resources consumes some memory. However, once the number of databases and routes are known for each Replication Server, the memory requirements can be quickly determined. For the sake of discussion, let’s assume we are trying to scope a Replication Server that will manage the following:

- 20 databases (10 primary, 10 replicate) along with 5 routes
- 2 of the 10 replicate databases have Warm Standby configurations as well.
- 4 of the replicate databases have had the dsi_sqt_max_cache_size set to 3MB
- The RSSD contains about 5MB of raw data due to the large number of tables involved.
- md_source_memory_pool is maxed at 983,040, sqt_max_cache_size to 1MB
- num_threads is set to 250 for good measure (system requires nearly 200)

The memory requirement would be:

<table>
<thead>
<tr>
<th>Configuration value/formula</th>
<th>Memory</th>
<th>Example (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_msgqueues * 205 bytes each</td>
<td>36KB default</td>
<td>100</td>
</tr>
<tr>
<td>num_msgs * 57 bytes each</td>
<td>2.5MB default</td>
<td>7,125</td>
</tr>
<tr>
<td>num_mutexes * 205 bytes each</td>
<td>140KB default</td>
<td>140</td>
</tr>
<tr>
<td>num_threads * 2800 bytes each</td>
<td>140KB default</td>
<td>684</td>
</tr>
<tr>
<td># databases * 64K + (16K * # WarmStandby)</td>
<td>1 MB min</td>
<td>1,312</td>
</tr>
<tr>
<td># databases * 2 * sqt_max_cache_size</td>
<td></td>
<td>40,960</td>
</tr>
<tr>
<td>dsi_sqt_max_cache_size – sqt_max_cache_size</td>
<td></td>
<td>8,192</td>
</tr>
<tr>
<td>md_source_memory_pool * # databases</td>
<td></td>
<td>19,200</td>
</tr>
<tr>
<td>size of raw data in RSSD (STS cache)</td>
<td></td>
<td>5,120</td>
</tr>
</tbody>
</table>

| Minimum Memory Requirement (MB)                                   | 83MB           |

Of course, the easy way is to just use the values below as starting points (assumes a normal number of databases ~10 or less - if more/less adjust memory by same ratio):

<table>
<thead>
<tr>
<th></th>
<th>Normal</th>
<th>Mid Range</th>
<th>OLTP</th>
<th>High OLTP</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqt_max_cache</td>
<td>512K</td>
<td>1MB</td>
<td>2MB</td>
<td>8MB</td>
</tr>
<tr>
<td>dsi_sqt_max_cache_size</td>
<td>0 (default)</td>
<td>512KB</td>
<td>1MB</td>
<td>4MB</td>
</tr>
<tr>
<td>memory_limit</td>
<td>32MB</td>
<td>64MB</td>
<td>128MB</td>
<td>256MB</td>
</tr>
</tbody>
</table>

The definitions of each of these are as follows:
Normal – thousands to tens of thousands of transactions per day

Mid Range – tens to hundreds of thousands of transactions per day

OLTP – hundreds of thousands to millions of transactions per day

High OLTP – millions to tens of millions of transactions per day

Stable Device Tuning

Stable Queue I/O

As you are well aware, Replication Server uses the stable device(s) for storing the stable queues. The space on each stable device is allocated in 1MB chunks divided into 64 16K blocks. While individual replication messages are stored in “rows” in the block, from an I/O perspective, all I/O is done at the block (16K) level, while allocation is done strictly at the 1MB level. As each block is read and messages processed, the block is marked for deletion as a unit. Only when all of the blocks within a 1MB allocation unit have been marked for deletion and their respective save intervals expired, will the 1MB be deallocated from queue. Often, this is a source of frustration with novice Replication Server Administrators who vainly try to drop a partition and expect it to go away immediately.

The reason for this discussion is that administrators need to understand that the RS will essentially be performing 16K I/O’s using sequential I/O unless the queue space gets fragmented with several different queues on the same device (see below). Ordinarily, this would lend itself extremely well to UFS (file system) storage as UFS drivers and buffer cache are tuned to sequential i/o – especially using “read-ahead” logic to reduce time waiting for physical I/O. The problem with using UFS is two fold:

- Replication Server uses asynchronous I/O (dAIO daemon) to ensure I/O concurrency with different SQM threads for the different queues. Still today, some UFS systems such as HP-UX 11 and Digital Unix 4.0f do not allow asynchronous I/O to file systems. The net result is that writing to a UFS effectively single threads the process as the operating system performs a synchronous write and blocks the process.

- While most vendors (HP included) have enabled the ability to specify raw I/O (unbuffered) to UFS devices, the I/O routines with RS have not been updated to take advantage of this fact. As a result, using UFS devices could cause a loss of replicated data should there be a file system error.

With the exception of the SQM tuning parameters discussed later, there is not much manual tuning you can do to improve I/O performance.

Stable Queue Placement

One often requested feature that nearly made it into the 12.0 release, was the concept of “partition affinity” (now slated for a later release). Partition affinity refers to the ability to physically control the placement of individual database stable queues on separate stable device partitions. This will help alleviate the I/O contention between inbound/outbound queues (hopefully, if implemented in such a way that control is enabled to that level of discretion) and particularly between two different database connections.

Currently, you can get similar behavior through an undocumented behavior. If all of the stable device partitions are added prior to creating the database connections, the Replication Server will round-robin placement of the database connections on the individual partitions. The difference between this and adding the database connections prior to adding the extra partitions is illustrated below.
connections created prior to stable devices part2 and part3

connections created after stable devices part2 and part3

Figure 8 – Stable Device Partition Assignment & Database Connection Creation

Obviously, the situation on the right is more preferable. However, even though it may start this way, due to much higher transaction volume to/from one connection vs. another or longer save interval, one queue may end up “migrating” onto another connection’s partitions.

Stable Partition Devices

Another common mistake that system administrators make, is placing the Replication Server on a workstation with only a single device (or a server but only allowing the Rep Server to address a single large disk). First, this causes a problem in that while a Rep Server can manage large numbers of stable partition devices, each one is limited to 2040MB (less than 2GB). This has nothing to do with 32-bit addressing or the Rep Server could address a full 2GB (2048MB). The real reason is that the limit in the RSSD system table rs_diskpartitions which tracks space allocations.

create table rs_diskpartitions
(
    name varchar(255),
    logical_name varchar(30),
    id int,
    num_segs int,
    allocated_segs int,
    status int,
    allocation_map binary(255),
    vstart int
)
go

In the above DDL for rs_diskpartitions, the column allocation_map is highlighted. As each 1MB allocation is allocated/deallocated within the device a bit is set/cleared within this column. Those quick with math realize that 255 bytes*8 bits/byte=2040 bits – and hence the reason for the partition sizing limits. Consequently, try as one might, without volume management software, Rep Server will never be able to use all of the space in a 40GB drive. The reason is that the 7 partition limit in Unix would restrict it to ~14GB of space. Those who are familiar with vstart would be quick to claim this could be overcome simply by specifying a ‘large’ vstart and allowing 2-3 stable devices per disk partition. Well, it doesn’t quite work that way with Replication Server. For example, consider the following sample of code:

add partition part2 on '/dev/rdsk/c0t0d1s1' with size=2040 starting at 2041

The above command will fail. The reason is the vstart is subtracted from the size parameter to designate how far in to the device the partition will start. Consequently, as documented in the Replication Server Reference Manual, the following command only creates a 19MB device instead of a 20MB device 1MB inside the specified partition (and the above command would have attempted a partition of –1MB!!).

add partition part2 on '/dev/rdsk/c0t0d1s1' with size=20 starting at 1

Now that we understand the good, bad, and ugly of Replication Server physical storage, you will understand the reason for the next concept:
Key Concept #9: Replication Server Stable Partitions should be placed on RAID subsystems with significant amounts of NVRAM. While RAID 0+1 is preferable, RAID 5 can be used if there is sufficient cache. Logical volumes should be created in such a way that I/O contention can be controlled through queue placement.

RSSD Generic Tuning

You knew this was coming. Or, at least you should have after all the discussions on the number and frequency of calls between the Replication Server and the RSSD.

Key Concept #10: Normal good database and server tuning should also be performed on the RSSD database and host database server.

What does this mean? Consider the following points:

- Place the RSSD database in a separate server from production systems. This provides the best situation for maintaining flexibility should a reboot of the production database server or RSSD database server be required. However, the main reason is that it reduces or eliminates CPU contention that the RSSD primary user might have with production system long running queries (don’t let the parallel table scans hold your replication system hostage).
- Raise the priority for the RSSD primary user.
- Place the tempdb in a named cache.
- Place the RSSD catalog tables in a named cache separate from the exceptions log (although rs_systext presents a problem – put it in with the system catalog tables) and also use a different cache for the queue/space management tables. This is as much to decrease spinlock contention as much as it is to ensure that repeated hits on one RS system table don’t flush another from cache.
- Dedicate a CPU to the RSSD database server. If more than one RSSD is contained in the same ASE server, monitor CPU utilization.
- Set the log I/O size (i.e. bind the log to a log cache with 4K pool) for the RSSD. There are a few triggers in the RSSD, including one on the rs_lastcommit table (fortunately not there in primary or replicate systems) that is used to ensure users don’t accidentally delete the wrong rows from the RSSD.

RSM/SMS Monitoring

Installing Replication Server Manager (RSM) is an often neglected part of the installation. Any site that is using Replication Server in a production system without using RSM or equivalent third-party tool (such as BMC patrol) has made a grave error that they will pay for within the first 3 months of operation. Why is this true?? Simply because most sites don’t test their applications today and as a consequence the transaction testing which is crucial to any distributed database implementation is missed. This virtually guarantees that a transaction, such as a nightly batch job, will fail at the replicate due to a replicate database/ASE issue. For example, the classic “ran out of locks” error from the replicate ASE during batch processing.

RSM Implementation

Having established the need for it, the next question is “How is it best implemented?” The answer, of course, is it depends. However, consider the following guidelines:

1. Configure one RSM Server on each host where a Replication Server or ASE resides. These RSM Servers will function as the monitoring “agents”.

33
2. Configure one RSM Server on primary SMS monitoring workstation per replication domain. This RSM will function as the RSM “domain manager”. All interaction to the RSM monitoring “agents” will be done through the SMS RSM “domain manager”.

3. Configure RSM Client (Sybase Central Plug-In) or other monitoring scripts to connect to the SMS RSM “domain managers”.

4. If Backup Server, OpenSwitch or other OpenServer process is critical to operation, consider having one of the RSM “monitoring agents” on that host also monitor the process if no other monitoring capability has been implemented.

5. RSM load ratios: 1 RS = 3 ASE = 20 OpenServers. If more than one RS is on a host, consider adding multiple RSM monitoring agents every 3-5 RS’s (depending on RS load).

6. **Do NOT allow changes to the replication system be implemented through RSM.** The main reason for this is that it is a GUI. You will have no record of these changes and it is too easy to make mistakes. Have developers create scripts that can be thoroughly tested and run with high assurance that “fat fingers” won’t crash the system.

The last bullet is important. Similar to not keeping database scripts, if you don’t have a record of your replication system, you will after the first time you have to recreate them. Following the above, a sample environment might look like the following:

![Diagram of example replication system monitoring](image)

**Figure 9 – Example Replication System Monitoring**

**RSM vs. Performance**

RSM or other SMS software monitoring can impact Replication performance in several ways:

- Unlike ASE’s shared memory access to monitor counters, the Replication Server and RSSD must be “polled” to determine system status information. If the polling cycle is set too small – or too many individual administrators are independently attempting to monitor the system, this polling could degrade RS and RSSD performance.

- Excessive use of the heartbeat feature can interfere with normal replication.

On one production system with a straight Warm Standby implementation, between the RS accesses and the RSM accesses to the RSSD, replication increased tempdb utilization by 10% (100,000 inserts out of 1,000,000) during a single day of monitoring. Because the way RSM “re-uses” many of the same users, it was impossible to differentiate
between RS and RSM activity. However, it is clearly enough of a load to consider the separate RSSD server vs. using an existing ASE in high volume environments.

All of this is leading up to one point:

*Key Concept #11 - Monitoring is critical - but make the heart beat, not race!*
Inbound Processing

What comes in...

Earlier we took a look at the internal Replication Server threads in a drawing similar to the following:

In the above copy of the diagram, note that the threads have been divided into inbound and outbound processing. An important distinction – and one than many do not understand – is that the inbound threads used for a replication from a source to a primary belong to a different connection than the outbound group of threads. Consequently, as multiple destinations are added, the same set of inbound threads are used to deliver the data to all of the various sets of outbound threads for each connection.

In the sections below, we will be addressing the three main threads in the inbound processing within Replication Server. The Rep Agent User thread will not be discussed as it’s sole purpose is to provide a connection mechanism for the Rep Agent.

SQM Processing

The Stable Queue Manager (SQM) is the only thread that interacts with the stable queue – despite illustrations in the Replication Server System Administration Guide to the contrary. As a result, it performs all logical I/O to the stable queue (physical i/o actually performed by the dAIO daemon) and as one would suspect is then one of the focus points for performance discussions. True, but, it is best to get a better understanding of the SQM thread to better see that in itself, the SQM thread may not be contributing to slow downs in inbound queue processing.

The SQM is responsible for the following:

- “Queue I/O”. All reads, writes, deletes and queue dumps from the stable queue.
- “Duplicate Detection”. Compares OQID’s from LTL to determine if LTL log row is a duplicate of one already received.

Features of the SQM thread include support for:

- Multiple Writers. While not as apparent in inbound processing, if the SQM is handling outbound processing, multiple sources could be replicating to the same destination (i.e. a corporate rollup).
• Multiple Readers. More a function of inbound processing, a SQM can support multiple threads reading from the inbound queue. This includes user connections, Warm Standby DSI threads along with normal data distribution.

SQM Performance Analysis

One of the best and most frequent commands for SQM analysis is the admin who, SQM command (sample output below extracted from Replication Server Reference Guide).

<table>
<thead>
<tr>
<th>Spid</th>
<th>State</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Awaiting</td>
<td>101:0 TOKYO_DS.TOKO_RSSD</td>
</tr>
<tr>
<td>15</td>
<td>Awaiting</td>
<td>101:1 TOKYO_DS.TOKO_RSSD</td>
</tr>
<tr>
<td>52</td>
<td>Awaiting</td>
<td>16777318:0 SYDNEY_RS</td>
</tr>
<tr>
<td>68</td>
<td>Awaiting</td>
<td>103:0 LDS.pubs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duplicates</th>
<th>Writes</th>
<th>Reads</th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>8867</td>
<td>9058</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0.1</td>
<td>2037</td>
<td>2037</td>
</tr>
<tr>
<td>0</td>
<td>0.1.0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B Writes</th>
<th>B Filled</th>
<th>B Reads</th>
<th>B Cache</th>
<th>Save_Int:Seg</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0:0</td>
</tr>
<tr>
<td>0</td>
<td>34</td>
<td>44</td>
<td>2132</td>
<td>0:33</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>54</td>
<td>268</td>
<td>0:4</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>23</td>
<td>0</td>
<td>strict:0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>First Seg.Block</th>
<th>Last Seg.Block</th>
<th>Next Read</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.0</td>
<td>0.1.0</td>
</tr>
<tr>
<td>33.10</td>
<td>33.10</td>
<td>33.11.0</td>
</tr>
<tr>
<td>4.12</td>
<td>4.12</td>
<td>4.13.0</td>
</tr>
<tr>
<td>0.1</td>
<td>0.0</td>
<td>0.1.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Readers</th>
<th>Truncs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Now that we understand how Replication Server allocates space (1MB allocations) and performs I/O (16K blocks – 64 blocks per 1MB), the above starts to make a bit more sense. Although a more detailed discussion is in the Reference Guide, a quick summary of the output is listed here for easy reference.

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spid</td>
<td>RS internal thread process id – equivalent to ASE’s spid</td>
</tr>
<tr>
<td>State</td>
<td>Current state of SQM – Awaiting message, it is caught up and not necessarily part of the problem. However, if state shows “Active” or “Awaiting I/O”, the SQM is busy writing data to/from disk.</td>
</tr>
<tr>
<td>Info</td>
<td>Queue id and database connection for queue</td>
</tr>
<tr>
<td>Duplicates</td>
<td>Number of LTL records judged as already received – can increase at Rep Agent startup, but if continues to increase, it is a sign of someone recovering the primary database without adjusting the generation id.</td>
</tr>
<tr>
<td>Writes</td>
<td>Number of messages (LTL rows) written to the queue</td>
</tr>
<tr>
<td>Reads</td>
<td>Number of messages read from queue. May surge high at startup due to finding the next row. However, after startup, if this number starts outpacing writes by any significant number, messages are being reread from the queue due to large transactions or SQT cache too small.</td>
</tr>
</tbody>
</table>
### Column Meanings

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bytes</td>
<td>Number of actual bytes written to queue. The efficiency of the block usage can be calculated by dividing “Bytes” by “B Writes”. Obviously if the blocks were always full, the result would be close to 16K. However, in normal processing, this is often not the case as transactions tend to be more sporadic in nature. The most useful uses of this column are to track bytes/min throughput and to explain why the queue usage may be different than estimated (i.e. low block density).</td>
</tr>
<tr>
<td>B Writes</td>
<td>Number of 16K blocks written to queue</td>
</tr>
<tr>
<td>B Filled</td>
<td>Number of 16K blocks written to queue that were full</td>
</tr>
<tr>
<td>B Reads</td>
<td>Number of 16K blocks read from queue</td>
</tr>
<tr>
<td>B Cache</td>
<td>Number of 16K blocks read from queue that are cached</td>
</tr>
<tr>
<td>Save Int:Seg</td>
<td>Save interval in minutes (left of colon) and oldest segment (1MB allocation) for which save interval has not yet expired.</td>
</tr>
<tr>
<td>First Seg:Block</td>
<td>First undeleted segment and block in the queue.</td>
</tr>
<tr>
<td>Last Seg:Block</td>
<td>Last segment and block written to the queue. As a result, the size of the queue can be quickly calculated via Last Seg – First Seg (answer in MB)</td>
</tr>
<tr>
<td>Next Read</td>
<td>The next segment, block and row to be read. If it points to the next block after Last Seg:Block, then the queue is quiesced (caught up). If continually behind, then reading is not keeping up with writes. If Replication Server is behind, the amount of queue to be applied is ~ Last Seg – Next Read (answer in MB)</td>
</tr>
<tr>
<td>Readers</td>
<td>Number of readers</td>
</tr>
<tr>
<td>Trunc</td>
<td>Number of truncation points</td>
</tr>
</tbody>
</table>

In the above table, performance indicators were highlighted. As such, these are indications – further commands will be necessary to determine exactly what the problem is. A frequent command for inbound queue determination is admin who, sqt, while for outbound queues, it most likely will be a look at the replicate database.

### SQM Tuning

To control the behavior of the SQM, there are a couple of configuration parameters available:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| init_sqm_write_delay | Write delay for the Stable Queue Manager if queue is being read.  
Default: 1000 milliseconds  
Init_sqm_write_delay should be less than init_sqm_max_write_delay |
| init_sqm_max_write_delay | The maximum write delay for the Stable Queue Manager if the queue is not being read.  
Default: 10,000 milliseconds |
| sqm_warning_thr1   | Percent of partition segments (stable queue space) to generate a first warning. The range is 1 to 100.  
Default: 75 |
| sqm_warning_thr2   | Percent of partition segments used to generate a second warning. The range is 1 to 100.  
Default: 90 |
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqm_warning_thr_ind</td>
<td>Percent of total partition space that a single stable queue uses to generate a warning. The range is 51 to 100. Default: 70</td>
</tr>
</tbody>
</table>

The first two take a bit of explaining. The stable queue manager waits for at least `init_sqm_write_delay` milliseconds for a block to fill before it writes the block to the correct queue on the stable device. Of course, this is the initial wait time. When the delay time has expired, the SQM writer will check if there are actually readers waiting for this block. If there are no readers waiting for the block, then SQM will adjust this time and make it longer for the next wait time. To realize what this means, you have to remember that the reader for the block typically will be the SQT, DSI or RSI threads. If the reader is caught up, then it is in fact waiting for the disk block, and the SQM needs to close the block so that the reader can access it immediately. However, if the reader is behind and is still processing previous blocks, then they will not be waiting for this block and consequently, the SQM can wait a bit longer to see if the block can be filled before flushing it to disk.

The key here is that this is how long the SQM will wait before writing to the queue if the DSI, RSI or SQT threads are active to ensure full blocks.

On the other hand, `init_sqm_write_max_delay` is how long a block will be held due to the fact that the DSI, RSI or SQT threads are suspended and not reading from the queue or the reader was not waiting for the block so the SQM delayed past `init_sqm_write_delay`. A flush to the queue is guaranteed to happen after waiting for `init_sqm_write_max_delay`. This parameter has to do more with when the block will be flushed from memory. If the RS is fully caught up, the SQM readers (when up) may be requesting to read the same disk block as was just written. The SQM cheats and simply reads the block from cache. However, if the SQM reader is not up or is lagging, this parameter controls how long the SQM will keep the block in cache waiting for the reader to resume or catch up.

These seem confusing, but consider the following scenario:

1. SQM begins receiving LTL rows and begins to build a 16K block. Assuming the DSI, RSI or SQT are up, it waits `init_sqm_write_delay` before writing the current block to disk.
2. `init_sqm_write_delay` expires, so block is written to disk. However, the block is still cached in memory of the SQM. If the block was not full and the readers were not waiting for it, the next block will wait longer (to a maximum of `init_sqm_write_max_delay`).
3. DSI, RSI, or SQT reads the next block. If RS is fully caught up, the block it is requesting is the one just written. To avoid unnecessary disk I/O, the block is simply read from cache vs. the copy flushed to disk.

Now, a little bit different. Let’s kill the SQM reader (i.e. suspend the DSI…or may SQT dies due to not enough resources).

1. SQM begins receiving LTL rows and begins to build a 16K block.
2. `init_sqm_write_delay` expires, however, readers are not up, consequently block is not flushed to disk unless it is full.
3. If the reader comes back up within `init_sqm_max_write_delay`, it is able to retrieve the block from the SQM cache as discussed above if the next block to read is the current block.
4. If the reader does not come back up within init_sqm_max_write_delay, the block is flushed to disk regardless of full status. The reader will have to do a physical I/O to retrieve the disk block.

Increasing the init_sqm_max_write_delay probably is not useful. Currently (RS 12.0), Rep Server only caches a single block. If the SQM reader (DSI, RSI or SQT) is down for any length of time, the Rep Agent or DIST will still be supplying data to the SQM. As a result, the block will in all likelihood fill and get flushed to disk. Consequently, it is more probable that the queue will begin to back up if the SQM reader is down, necessitating a physical I/O. The only time increasing this may make sense is if increasing the init_sqm_write_delay to greater than 10,000ms – a very rare situation in which queue space may be at a premium and write activity is very low in the source system.

From a performance perspective, the most common cause of SQM contributing to performance issues is simply if the SQM can’t write to disk fast enough. Other than the “lucky” instances where you might see the state column in the admin who, sqm command stating “Awaiting I/O” this may be difficult to detect as the bytes written to the queue may be more than what was written to the transaction log. However, if you see that the transaction log’s rate exceeds the SQM rate – it may be an indication that the Rep Agent is not able to keep up. The SQM rarely is the culprit – typically it is that SQM reader (DSI, RSI or SQT) that is the root cause or the SQM writer (DIST, Rep Agent) that is the holdup.

**SQT Processing**

Unlike the SQM, which frequently is not the blame, the Stable Queue Transaction (SQT) thread is a common cause of problems – frequently simply addressable by adding cache. As mentioned earlier, the SQT thread is responsible for sorting the transactions into commit order. In order to better understand the performance implications of this (and the output of admin who, sqt), it is best to understand how the SQT works.

**SQT Sorting Process**

The SQT sorts transactions by using 4 linked lists, often referred to (confusingly enough) as “queues”. These lists are:

- **Open** – The first linked list that transactions are placed on, this queue is a list of transactions for which the commit record has not been processed or seen by the SQT thread yet.

- **Closed** – Once the commit record has been seen, the transaction is moved from the “Open” list to the closed list and a standard OpenServer callback is issued to the Distributor thread (or DSI, although this is internal to the DSI as will be discussed later in the outbound section).

- **Read** – Once the DIST or DSI threads have read the transaction from the SQT, the transaction is moved to the “Read” queue.

- **Truncate** – Along with the Open queue, when a transaction is first read in to the system, the transaction structure record is placed on the Truncate queue. Only after all of the transactions on a block have had the commit statements read and been processed by the DIST and placed on the read queue can the SQT request the SQM to delete the block.

To get a better idea how this works, consider the following example of three transactions committed in the following order at the primary database:
In this example, the transactions committed in the order 2-3-1. Due to the commit order, however, the transactions might have well have been applied similar to:

```
T17 T00 T16 T15 T14 T13 T12 T11 T10 T09 T08 T07 T06 T05 T04 T03 T02 T01 T00
CT1 BT1 D1 9 I1 8 U1 7 D1 6 U1 5 I1 4 I1 3 I1 2 I1 1 BT1
CT2 U2 7 I2 6 I2 5 I2 4 I2 3 I2 2 I2 1 BT2
CT3 D3 5 D3 4 I3 3 U3 2 U3 1 BT3
```

However, the transaction log is not that neat. In fact, it would probably look more like the following:

```
CT1 D1 9 I1 8 I1 7 U1 6 D1 5 I1 4 I1 3 I1 2 I1 1 BT1
CT2 U2 7 I2 6 I2 5 I2 4 I2 3 I2 2 I2 1 BT2
CT3 D3 5 D3 4 I3 3 U3 2 U3 1 BT3
```

After the Rep Agent has read the log into the RS, the transactions may be stored in the inbound queue in blocks similar to the following (assuming blocks were written due to timing and not due being full):

```
CT1 D1 9 I1 8 I1 7 CT3 D3 5 D3 4 CT2 I3 3 U3 2 U3 1 I2 6 BT3 I2 4 I2 3 I2 2 I2 1 BT2 I2 2 I2 1 BT1
```
The following diagrams illustrate the transactions being read from the SQM by the SQT, sorted via the Open, Closed, Read and Truncate queues within the SQT.

Note that the transaction is given a transaction structure record (TX1 in above) and statements read thus far along with the begin transaction record have been linked in a linked list to the Open queue. Note that immediately after reading the transaction from the SQM, the transaction id is recorded in the linked list for the Truncate queue. Continuing on and reading the next block from the SQM yields:

Having read the second block from the SQM, we encounter the second transaction. So, we begin a second linked list for its statements as well as continuing to build the first transactions list with statements belonging to it read from the second block. Additionally, we add that transaction to the Truncate queue. Continuing on and reading the next block from the SQM yields:
No new transactions were formed, so we are simply adding statements to the existing transaction linked lists.
Continuing on yields the following SQT organization:

```
Figure 18 – SQT Queues After Reading Inbound Queue Block 0.3
```

At this point, we have all three transactions in progress. Continuing with the next block read from the SQM yields the first commit transaction (for TX2). Since we now have a commit, the transaction’s linked list of statements is simply moved to the “Closed” queue and the DIST thread notified of the completed transaction. This yields an SQT organization similar to:

```
Figure 19 – SQT Queues After Reading Inbound Queue Block 0.4
```

Continuing with the next read from the SQM, the DIST is able to read TX2 which causes it to get moved to the “Read” queue and the commit record for TX3 is read, which moves it to the “Closed” queue. This yields an SQT organization similar to:
At this juncture, you might think that we could remove TX2 from the inbound queue. However, if you remember, all I/O is done at the block level. In addition, in order to free the space, the space must be freed contiguously from the front of the queue (block 0.0 in this case). Since the statements that make up TX2 are scatter among the blocks and statements for transactions for which the commit has not been seen yet, the deletion of TX2 must wait. Continuing on with the last block to be read, yields the following:

At this stage, all transactions have been closed, however, we still cannot remove them from the inbound queue. Remember, this is strictly memory sorting (SQT cache), consequently, if we removed them from the inbound queue now and a system failure occurred, we would lose TX1. Consequently, we have to wait until it has been read by the DIST. Once that is done, all three transactions would be in the “Read” queue and consequently a contiguous block of transactions could be removed since all of the transactions on the blocks have been read. If however, block 0.6 also contained a begin statement for TX4, then the deletes could still be done for blocks 0.0 through 0.5. How? The answer is that the SQM flags each row in the queue with a status flag that denotes whether it has been processed. Consequently on restart after recovery, the SQT doesn’t attempt to resort and resend transactions already processed. Instead, it simply starts with the first active segment/row and begins sorting from that point.

**SQT Performance Analysis**

Now that we see how the SQT works, this should help explain the output of the admin who, sqt command (example copied from Replication Server Reference Manual).
The observant will say that not all the SQT threads are listed as the ones for the inbound queues (designated with qid:1) are present, but the ones for outbound queues (designated qid:0) are missing. Well, the reality is that there is not a SQT thread for outbound queues. Instead, the DSI (Scheduler) calls SQT routines. Consequently, spids 10 & 0 above represent DSI threads performing SQT library calls. The output for the columns are described in the below table:

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spid</td>
<td>Process Id for each SQT thread</td>
</tr>
<tr>
<td>State</td>
<td>State of the processing for each SQT thread</td>
</tr>
<tr>
<td>Info</td>
<td>Queue being processed</td>
</tr>
<tr>
<td>Closed</td>
<td>Number of transactions in the “Closed” queue waiting to be read by DIST or DSI. If a large number of transactions are “Closed”, then the next thread (DIST or DSI-Exec) is the bottleneck as the SQT is simply waiting for the reader to read the transactions.</td>
</tr>
<tr>
<td>Read</td>
<td>Number of transactions in the “Read” queue. This essentially explains the number of transactions process not yet deleted from the queue. A high number in this block may point to a long transaction that is still “Open” at the very front of the queue (i.e. user went to lunch) as deleting queue space is fairly quick.</td>
</tr>
<tr>
<td>Open</td>
<td>Number of transactions in the “Open” queue for which commit has not been seen by SQT yet (although SQM may have written it to disk already)</td>
</tr>
<tr>
<td>Trunc</td>
<td>Number of transactions in the “Truncate” queue – essentially an ordered list of transactions to delete once processed in disk contiguous order. Trunc is the sum of the Closed, Read, and Open columns (due to reasons discussed above).</td>
</tr>
<tr>
<td>Removed</td>
<td>Number of transactions removed from cache. Transactions are removed if the cache becomes full or the transaction is a large transaction (discussion later)</td>
</tr>
<tr>
<td>Full</td>
<td>Denotes if the SQT cache is currently full.</td>
</tr>
<tr>
<td>SQM Blocked</td>
<td>1 if the SQT is waiting on SQM to read a message. This state should be transitory unless there are no closed transactions.</td>
</tr>
</tbody>
</table>
First Trans

This column contains information about the first transaction in the queue and can be used to determine if it is an unterminated transaction. The column has three pieces of information:

- ST: Followed by O (open), C (closed), R (read), or D (deleted)
- Cmds: Followed by the number of commands in the first transaction
- qid: Followed by the segment, block, and row of the first transaction

An example would be ST:O Cmds: 3245 qid: 103.5.23 – which basically tells you that at this stage, the first transaction in the queue is still “Open” (no commit read by SQT) and so far it has 3,245 commands in the transaction (probably a large one) and begins in the queue at segment 103 block 5 row 23. As we will see later, this is a very useful piece of information.

Parsed

The number of transactions that have been parsed. This is the total of transactions including those already deleted from the queue. Along with statistics, this field can give you an idea of the transaction volume over time.

SQM Reader

The index of the SQM reader handle. If multiple readers of an SQM, this designates which reader it is.

Change Oqids

Indicates that the origin queue ID has changed. Typically this only happens in Warm Standby after a switch active.

Detect Orphans

Indicates that it is doing orphan detection. This is largely only noticed on RSI queues. For normal database queues, if someone does not close their transaction when the system crashes, on recovery, the Rep Agent will see the recovery checkpoint and instruct the SQM to purge all the open transactions to that point.

Admin who, sqt is one of the key commands to determining problems on the inbound queue performance. In addition to helping you identify progress of transactions through the Open, Closed, Read and Truncate queues, it is extremely useful for determining when you have encountered a large transaction – or, one that is being held open for very long time. The column that assists in this is the “First Trans” column. Above we gave an example of one view of a large transaction (ST:O Cmds: 3245 qid: 103.5.23). Consider the following tips for this column:

<table>
<thead>
<tr>
<th>ST:</th>
<th>Cmds</th>
<th>Qid</th>
<th>Possible Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>increasing</td>
<td>same</td>
<td>Large transaction</td>
</tr>
<tr>
<td>O</td>
<td>same</td>
<td>same</td>
<td>Rep Agent down or uncommitted transaction at primary</td>
</tr>
<tr>
<td>O</td>
<td>changes</td>
<td>increasing</td>
<td>SQT processing normally</td>
</tr>
<tr>
<td>C</td>
<td>changes</td>
<td>slow</td>
<td>SQT reader not keeping up (if problems)</td>
</tr>
<tr>
<td>C</td>
<td>same</td>
<td>same</td>
<td>DIST down, outbound queue full</td>
</tr>
<tr>
<td>R</td>
<td>same</td>
<td>same</td>
<td>Transaction on same block/queue still active</td>
</tr>
</tbody>
</table>

Common Performance Problems

The most common problems with the SQT are associated with 1) large transactions; and 2) slow SQT readers (i.e. DIST). The first deals with the classic 10,000 row delete. If the SQT attempted to cache all of the statements for such a delete in it’s memory, it would quickly be exhausted. Consequently, when a large transaction is encountered, the SQT simply discards all of its statements and merely keeps the transaction structure in the appropriate list. However, this means that in order for the transaction to be passed to the SQT reader, the SQT must go back to the beginning of the transaction and physically rescan the disk. In addition to the slow down of simply doing the physical i/o, it effectively pauses the scanning where the SQT had gotten to until that transaction is fully read back off disk and sent to the DIST, etc.
The second problem is common as well. In cases where the DIST, or DSI threads cannot keep up, the Closed queue continues to grow until all available cache is used. Once this begins to happen, the SQT has a decision to make. If there are transactions in the Closed or Read queue, the SQT simply halts reading the SQM until the transaction is complete and queue can be truncated. If there are no transactions in the Closed or Read queue, the SQT finds the largest transaction in the Open queue, discards the statements (keeping the transaction structure – similar to a large transaction) and then keeps processing. Should this continue for very long, a large number of transactions in the SQT cache may have to be rescanned – further slowing down the overall process.

**SQT Performance Tuning**

There are two main ways of improving SQT performance. The first is rather obvious – increase the amount of memory that the SQT has by changing the value for sqt_max_cache_size. By default, the SQT has 128K for each inbound and outbound queue. So, for a total of 2 source and 5 destination databases we would have 14 (2 source inbound/outbound and 5 destination inbound/outbound) 128K memory segments for SQT cache. However, 128K is typically far, far too little. Most medium production systems need 512K – 1MB SQT caches with high volume OLTP systems using anywhere from 2MB+ of cache. Obviously, the more connections you have, the more this impacts overall Replication Server memory settings. With a 1MB sqt_max_cache_size setting, the earlier example of 2 source/5 destinations would require 14MB strictly for SQT cache.

This may be a bit excessive. If you think about it, if each source database is replicating to individual destination systems (1 to 2 and the other to 3), the outbound queue will contain “sorted” transactions provided that no other DIST thread is replicating into the destination. As a result, the SQT cache may not be fully utilized for the DSI – and it can be adjusted down on a connection-by-connection basis via the dsi_sqt_max_cache_size. However, if using Parallel DSI, the DSI may need more SQT cache to keep up with the multiple DSI’s parsing requirements.

The best way to detect either of these two situations is to watch the system during periods of peak activity via the admin who, sqt command. If the “Full” column is set to a 1, then it is a clear indication that SQT cache is undersized – particularly from the inbound processing side. In addition, another indication is the “Removed” column can give you a good idea even if the cache is not full. If the “Removed” column is growing and the transactions are not large, then it is probable that the cache was filled to capacity several times and multiple transactions normally not considered large were removed to make room.

**Distributor Processing**

Of all the processes in the Replication Server, the Distributor thread is probably the most CPU intensive. The reason for this is that the DIST thread is the “brains” behind the Rep Server – determining where the replicated data needs to go. In order to determine the routing of the messages, the DIST thread will call three library routines. These library routines are discussed below.

**Subscription Resolution Engine**

The Subscription Resolution Engine (SRE) looks at each statement independently and evaluates the distribution rules. In doing so, the SRE is responsible for determining two conditions:

- Which databases should receive the replicated insert, update, delete or procedure execution.
- Whether or not subscription migration is necessary at the destination.

For the most part, the SRE simply has to do a row-by-row comparison for each row in the transaction – it can ignore the begin/commit statements since they will be handled by the TD library calls later (discussed next). In checking each row, the SRE will be focusing on the subscriptions and their conditions for each replication definition defined on the object the row modification is for.

**Subscription Conditions**

To maintain performance, the SRE is a very lean/efficient set of library calls that only supports the following types of conditionals:

- Equality – for example col_name = constant. A special type of equality is permitted using rs_id columns is bit-wise comparisons with the logical AND (&) function.
• Range (unbounded and bounded) – for example col_name < constant or col_name > low_value and col_name < high_value
• Boolean AND conditionals

Note that several (sometimes disturbing to those new to Replication Server) forms of conditionals are not supported:

• Functions, formulas or operators (other than & with rs_address columns) are not supported
• Boolean OR, NOT, XOR conditionals. Boolean OR conditionals are easily accomplished via simply creating two subscriptions – one for each side of the OR clause.
• Not equals (!=, <>) comparators. However, this is easily bypassed by treating the situation like a non-inclusive range. For example (col_name != “New York”) becomes (col_name < “New York” OR col_name > “New York”) which is handled simply by using two subscriptions. For “not null” comparisons, a single subscription based on (col_name > NULL) is sufficient.
• Columns contained in the primary key can not have rs_address datatypes.

Many Replication System Administrators complain that the rs_address column isn’t as useful as it could be simply because it only supports 32 bits – restricting them to 32 sites in the organization. As a result, they feel that as their business grows, they have to add more rs_address columns causing considerable logic to be programmed in to the application or database triggers to support replication. While one rs_address column is easily managed, they are reluctant to add more. A valid complaint if you think of the bits one dimensionally with sites. Of course, using the rs_address column as an integer and subscribing with a normal equality (for example, subscribing where site_id = 123 vs. subscribing where site_id & 64) extends this near infinitely, however, if the data modification is projected for multiple sites, this could require multiple updates to the same rows and subscription migration issues. A better solution might be to think of the bits in the rs_address columns as components similar to class B & class C Internet addresses. High order bytes could be associated with countries or regions while the low order bits with specific sites within those regions. Consider the following examples of bit-wise addressing:

<table>
<thead>
<tr>
<th>Bit Addressing</th>
<th>Total Sites</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 – 28</td>
<td>112</td>
<td>Could be 4 World Regions – each with 28 locations</td>
</tr>
<tr>
<td>8 – 24</td>
<td>192</td>
<td>World Region – Location</td>
</tr>
<tr>
<td>16 – 16</td>
<td>256</td>
<td>Country/Region – Location</td>
</tr>
<tr>
<td>4 – 4 – 24</td>
<td>384</td>
<td>World Region – Country – Location</td>
</tr>
<tr>
<td>4 – 8 – 16</td>
<td>512</td>
<td>Hemisphere – Country/Region – Location</td>
</tr>
<tr>
<td>4 – 4 – 4 – 20</td>
<td>1280</td>
<td>Hemisphere – Country – Region – Location</td>
</tr>
<tr>
<td>4 – 4 – 8 – 16</td>
<td>2048</td>
<td>Hemisphere – Country – Region – Location</td>
</tr>
<tr>
<td>4 – 4 – 8 – 8 – 8</td>
<td>8192</td>
<td>Hemisphere – Country – Region – District – Office</td>
</tr>
</tbody>
</table>

While this does expand the number of conditions that must be checked, it logically fits with distribution rules the application may be trying to implement and therefore mentally easier to implement. Additionally, in the above, we treated each as separate individual locations. If the last bit address represented a region or “cell”, then the number of sites addressable with each scheme extends another order of magnitude.

As mentioned earlier, using multiple rs_address columns or “dimensioning” the rs_address column will result in more conditionals for the SRE to process. For multiple columns, the reason should be obvious – a separate condition would be necessary for each column. However, the same is true for rs_address columns that have been dimensioned – a separate condition would be necessary for each “dimension” at a minimum. This is simply due to the fact that the original intent of the rs_address column was a single dimension of bits. Consequently, when a condition such as (column & 64) returns a non-zero number, the row is replicated. Combining several bits as in (column & 71) could have some unexpected results. Since “71” is 64+4+2+1 (bits 6,2,1, and 0), you might think that this would achieve the goal. However, the way rs_address columns are treated, any column which has bits 6, 2, 1 or 0 on would get replicated to that site. This includes rs_address values of 3, 129, etc. The only way to ensure that exactly the desired value is met is to use multiple conditions as in:
-- my_rsaddr_col is an rs_address column
create subscription titles_sub
with replicate at SYDNEY_DS.pubsub
where my_rsaddr_col & 64
and my_rsaddr_col & 4
and my_rsaddr_col & 2
and my_rsaddr_col & 1

SRE Performance

Performance of the SRE depends on a number of issues that should be fairly obvious:

- Number of replication definitions per table.
- Number of subscriptions per replication definition
- Number of conditions per subscription

In order to reduce the number of physical RSSD lookups to retrieve replication definitions, subscriptions and where clauses, the SRE makes use of the System Table Services (STS) cache. Configurable through the replication server configuration sts_cachesize, the STS caches rows from each RSSD table/key combination in a separate hash table. The sts_cachesize parameter refers to the number of rows for each RSSD table. For most systems, the default sts_cachesize configuration of 100 is far too low. This would restrict the system to only retaining the most current 100 rows of subscription where clauses, etc. A better starting point might be to set sts_cachesize equal to the max of the number of repdefs managed by the current Rep Server or the number of subscriptions on the repdefs managed by the current Rep Server. A good way to determine how effective the STS cache is, is to turn on the cache statistics trace flag.

trace "on", STS, STS_CACHESTATS - Collects STS Statistics

As you can imagine, the largest impact that you can have is by increasing sts_cachesize to reduce the physical lookups.

Key Concept #12: The single largest tuning parameter to improve Distributor thread performance is increasing the sts_cachesize parameter in order to reduce physical RSSD lookups.

Transaction Delivery

The Transaction Delivery (TD) library is used to determine how the transactions will be delivered to the destinations. The best way to think of this is that while the SRE decides who gets which individual modifications, the TD is responsible for “packaging” these modifications into a transaction and requesting the writes to the outbound queue. For example, consider the following transaction:

begin transaction web_order
insert into orders (customer, order_num, ship_addr, ship_city, ship_state, ship_zip)
values (1122334, 123456789, "123 Main St", "Anytown", "NY", 21100)
insert into order_items (order_num, item_num, desc, qty, price, discount, total)
values (123456789, "31245Q", "Chamois Shirt", 250.00, 2, 0, 500.00)
insert into order_items (order_num, item_num, desc, qty, price, discount, total)
values (123456789, "987652W", "Leather Jacket", 250.00, 1, 0, 250.00)
insert into order_items (order_num, item_num, desc, qty, price, discount, total)
values (123456789, "54783L", "Welcome Mat", 12.00, 1, 0, 12.00)
insert into order_items (order_num, item_num, desc, qty, price, discount, total)
values (123456789, "732189H", "Volley Ball Set", 129.00, 1, 0, 129.00)
update orders set order_subtotal=$964.60, order_shipcost=$20, order_total=$984.60
commit transaction

Now, picture what happens in a normal replication environment if the source system was replicating to three destinations – each concerned with it’s own set of rules. For example, Replicate Database 1 (RDB1) might be concerned with clothing transactions (shipping warehouse for clothing), while RDB2 with transactions for household goods, and RDB3 focusing on sporting items. This would result in the following replicate database transactions:

begin transaction clothing_items
insert into orders (customer, order_num, ship_addr, ship_city, ship_state, ship_zip)
values (1122334, 123456789, "123 Main St", "Anytown", "NY", 21100)

-- replicate database 1 (clothing items)
begin transaction web_order
insert into orders (customer, order_num, ship_addr, ship_city, ship_state, ship_zip)
values (1122334, 123456789, "123 Main St", "Anytown", "NY", 21100)
The SRE physically determines what DML rows go to which of the replicates, however, it is the TD that “remembers” that each is within the scope of the outer transaction “web_order” and requests the rows to be written to each of the outbound queues.

Message Delivery

The Message Delivery (MD) module is called by the DIST thread to optimize routing of transactions to data servers or other Replication Servers. The DIST thread passes the transaction row and the destination ID to the MD module. Using this information and routing information in the RSSD, the module determines where to send the transaction:

- If the current Replication Server manages the destination connection, the message is written to the outbound queue via the SQM for the outbound connection.

- If the destination is managed by another Replication Server, the MD module checks to see if it is already sending the exact same message to another database via the same route. If so, the new destination is simply appended to the existing message. If not, the message is written to the outbound queue via the SQM for the RSI connection to the Replicate Replication Server.

This last point is crucial to understanding a major performance benefit to routing data – consider the following architecture

![Figure 22 – Example World Wide Topology](image)
multi-tiered aspects of the Pacific arena above, NY would only have to send a single message to cover Chicago, Dallas, Mexico City, San Francisco, Tokyo, Taiwan, Hong Kong, Peking, Sydney Australia, New Delhi. In the past, this has often been touted as a means to save expensive trans-oceanic bandwidth. While this may be true, from a technical perspective, the biggest savings is in the workload required of any one node – allowing unparalleled scalability.

In addition, this performance advantage gained by distributing the outbound workload may make it feasible to implement replication routing even to Replication Servers that may reside on the same host. Take, for example, the following scheme.

![Diagram](image)

**Figure 23 – Example Retail Sales Data Distribution**

In this scenario, if the Replication System begins to lag, the POS system may be impacted due to the affect the Replication Server could have on the primary transaction log if the Replication System’s stable devices are full. While none of the systems are very remote from the POS system, in this case, it may make sense to implement a MP Rep Server implementation by using multiple Replication Servers to balance the load.

![Diagram](image)

**Figure 24 – Retail Sales Data Distribution using Multiple Replication Servers**

Note that in the above example solution, the RS that manages the POS connection does not then manage any other database connections. Consequently, that RS can concentrate strictly on inbound queue processing and subscription resolution. The other three can concentrate strictly on applying the transactions at the replicates. With a 6-way SMP box, all four Replication Servers, along with a single ASE implementation for the RSSD databases could start making more effective use of larger server systems that they may be installed on.
Key Concept #13: While replication routes offer network bandwidth efficiency, they offer a tremendous performance benefit to Replication Server by reducing the workload on the primary Replication Server. This can be used to effectively create a MP Replication scenario for load balancing in local topologies.

An additional performance advantage in inconsistent network connectivity environments is that network problems that occur during Replication Server applying the transactions at the replicate can degrade performance due to frequent rollbacks/retry due to loss of connection.

Other than the sts_cachesize and replication routing, the other performance tuning parameter that directly affects the distributor thread is md_source_memory_pool. This is a memory pool specifically for the MD to cache the writes to the SQM for the outbound queues. While this is a connection scope tuning parameter, it is often misunderstood as it does not change destination connections, but rather the source connection. The reason for this is that we are still discussing the Distributor thread, which is part of the inbound side of replication server internal processing. By adjusting the md_source_memory_pool, you allow the source connection’s distributor thread to cache its writes when the outbound SQM is busy and to enable more efficient outbound queue space utilization. This is especially useful when a source system is replicating to multiple destinations without routing, when a replicate database has more than one source database (i.e., corporate rollup), or for the remote replication server when multiple destinations exist for the same source system. While it may help with outbound SQM efficiency, it really is not as useful in situations where a single database is replicating to another single database which only has that one source. The actual memory used and overall performance of the MD library can be monitored through the admin statistics command as follows:

```
admin statistics, md
```

<table>
<thead>
<tr>
<th>Source</th>
<th>Pending_Messages</th>
<th>Memory_Currently_Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYDNEY_DS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TOKYO_DS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>TOKYO_DS</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Messages_Delivered</th>
<th>SQM_Writes</th>
<th>Destinations_Delivered_To</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>34</td>
<td>34</td>
</tr>
<tr>
<td>551</td>
<td>551</td>
<td>551</td>
</tr>
<tr>
<td>1452</td>
<td>1452</td>
<td>1452</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Max_Memory_Hit</th>
<th>Is_RSI_Source?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Each of these values are described below:

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source</td>
<td>The Replication Server or data server where the message originated.</td>
</tr>
<tr>
<td>Pending_Messages</td>
<td>The number of messages sent to the SQM without acknowledgment. Usually, this occurs because Replication Server is processing the messages before writing them to disk.</td>
</tr>
<tr>
<td>Memory_Currently_Used</td>
<td>Memory used by pending messages.</td>
</tr>
<tr>
<td>Messages_Delivered</td>
<td>Number of messages delivered.</td>
</tr>
<tr>
<td>SQM_Writes</td>
<td>Number of messages received and processed.</td>
</tr>
<tr>
<td>Destinations_Delivered_To</td>
<td>Total number of destinations.</td>
</tr>
<tr>
<td>Max_Memory_Hit</td>
<td>Not yet implemented.</td>
</tr>
<tr>
<td>Is_RSI_Source?</td>
<td>Indicates whether the current Replication Server can send messages:</td>
</tr>
<tr>
<td></td>
<td>0 - This Replication Server cannot send messages 1 - This Replication Server can send messages</td>
</tr>
</tbody>
</table>
Minimal Column Replication

Unfortunately, appending the clause “replicate minimal columns” to replication definitions is often forgotten. While not reducing the workload of the DIST thread so much, it can dramatically reduce the workload of the DSI thread. To understand the impact of this, you first have to understand what happens normally.

Normal Replication Behavior

Under normal (non-minimal column) replication, the DIST thread does not perform any checking of what columns have been changed for an update statement. As a result, if an update of only 2 columns of a 10 column table occurs, Replication Server constructs a default functions string containing an update for all 10 columns of the table, setting the column values equal to the new values with a where clause of the primary key old values. For example, consider the following table (from pubs2 sample database shipped with Sybase ASE) and associated indexes.

```sql
create table titles
(title_id tid not null,
title varchar(80) not null,
type char(12) not null,
pub_id char(4) null,
price money null,
advance money null,
total_sales int null,
notes varchar(200) null,
pubdate datetime not null,
contract bit not null)
go
create unique clustered index titleidind
on titles (title_id)
go
create nonclustered index titleind
on titles (title)
go
```

For further fun, note that the salesdetail table has a trigger that updates the title.total_sales column:

```sql
create trigger totalsales_trig
on salesdetail
for insert, update, delete
as
/* Save processing: return if there are no rows affected */
if @@rowcount = 0
begin
return
end

/* add all the new values */
/* use isnull: a null value in the titles table means */
/* "no sales yet" not "sales unknown" */
update titles
set total_sales = isnull(total_sales, 0) + (select sum(qty)
from inserted
where titles.title_id = inserted.title_id)
where title_id in (select title_id from inserted)

/* remove all values being deleted or updated */
update titles
set total_sales = isnull(total_sales, 0) - (select sum(qty)
from deleted
where titles.title_id = deleted.title_id)
where title_id in (select title_id from deleted)
go
```

By now some of you may be already seeing the problem. As mentioned previously, for an update statement, RS will generate a full update of every column. Consider a mythical replication definition like:

```sql
create replication definition CHINOOK_titles_rd
with primary at CHINOOK.pubs2
with all tables named 'titles'
{
  "title_id" varchar(6),
  "title" varchar(80),
  "type" char(12),
  "pub_id" char(4),
  "price" money,
}
This means the function string (if you were to mimic it by altering the function string) would resemble:

```
alter function string CHINOOK_titles_rd.rs_update
for rs_sqlserver_function_class
output language
update titles
set title_id = ?title_id!new?,
title = ?title!new?,
type = ?type!new?,
pub_id = ?pub_id!new?,
price = ?price!new?,
advance = ?advance!new?,
total_sales = ?total_sales!new?,
notes = ?notes!new?,
pubdate = ?pubdate!new?,
contract = ?contract!new?
where title_id = ?title_id!old?
```

The result is rather drastic. The first problem, of course, is that the outbound queue will contain significantly more data than actually was updated - assuming the notes column was filled out. But this is minor compared to what really impacts DSI delivery speed.

For those of you familiar with database server performance issues, any time a row is updated, any index values that are updated automatically cause the index to be treated as “unsafe” and therefore also needing updated. In this example, every time a new order is inserted into the salesdetail table, the corresponding update at the replicate not only updates the entire row - it also performs index maintenance. Worse yet, if ANSI constraints were used, the related foreign key tables would have holdlocks placed on the related rows, increasing the probability of contention.

Clearly, this is not desirable behavior. Unfortunately, it occurs much more often than you would think. Consider:

- **Aggregate columns** – Such as the titles example.
- **Auditing columns** – this includes such columns as last_update_user, last_updated_date, etc. – similar to the trigger issue mentioned previously.
- **Status columns** – shipping/order status information for order entry or any workflow system.
- **Dynamic values** – product prices (sale prices, etc.). Consider a regional chain store that wants to replicate price changes to 60+ stores for 100’s of products. Now add in the overhead of changing every column and index maintenance – and the associated impact that could have on store operations.

Undoubtedly, there are others you could think of as well.

**Minimal Column Replication**

When the replication definition includes the “replicate minimal columns” phrase, the behavior is much different. With minimal column replication, only the columns whose before and after images do not match – as well as primary key values – are written to the outbound queue. Consequently, most of the updates to the titles table would be executing a function string similar to:

```
alter function string CHINOOK_titles_rd.rs_insert
for rs_sqlserver_function_class
output language
update titles
set total_sales = ?total_sales!new?
where title_id = ?title_id!old?
```

Which more than likely will execute much quicker in high volume environments. An interesting aspect to minimal column replication is what happens if the only columns updated were columns not included in the replication...
definition. Under normal replication rules, if a column is updated, the rs_update function is processed and set to the RS. The DIST thread simply strips out any columns not being replicated and generates the functions as appropriate. For example, in the above titles table, let’s assume that the contract column was excluded from the replication definition as in:

```sql
create replication definition CHINOOK_titles_rd
  with primary at CHINOOK.pubs2
  with all tables named 'titles'
  {
    "title_id" varchar(6),
    "title" varchar(80),
    "type" char(12),
    "pub_id" char(4),
    "price" money,
    "advance" money,
    "total_sales" int,
    "notes" varchar(200),
    "pubdate" datetime
  }
  -- Primary key determination based on: Primary Key Definition
  primary key ("title_id")
  searchable columns ("title_id")
```

Of course, the full update function string would now be:

```sql
alter function string CHINOOK_titles_rd.rs_update
for rs_sqlserver_function_class
output language
',
update titles
  set title_id = ?title_id!new?,
  title = ?title!new?,
  type = ?type!new?,
  pub_id = ?pub_id!new?,
  price = ?price!new?,
  advance = ?advance!new?,
  total_sales = ?total_sales!new?,
  notes = ?notes!new?,
  pubdate = ?pubdate!new?
where title_id = ?title_id!old?
',
```

Now, consider the following update statement:

```
Update titles set contract=1 where title_id="BU1234"
```

If this statement was executed at the primary, the replicate would receive a full update statement of all columns in the replication definition (excluding the contract column, of course), setting them to the same values they already are.

As you can guess, under minimal columns, this behaves differently. Obviously, if the only column(s) updated were columns excluded from the replication definition, the RS would otherwise attempt to generate an empty “set clause”. Consequently, if no columns in the replication definition were updated, the update itself is ignored by the Replication Server.

Keep in mind that this does impose a number of restrictions:

- Autocorrection can not be used while minimal column replication is enabled.
- Custom function strings containing columns other than the primary keys may not work properly or generate errors.

Regarding the first restriction, autocorrection should not normally be on. If left on, performance could be seriously degraded as each update translates into a delete/insert pair. Even if the values haven’t changed, this can have a greater penalty than not using minimal columns as the index maintenance load could be greater due to first removing the index keys (and any corresponding page shrinkage) and then re-adding them (which could cause splits). Consequently, **minimal column replication should be enabled by default**, and when autocorrection is necessary due to inconsistencies, the replication definition can be altered to remove minimal column replication (temporarily). Note that minimal column replication really only applies for updates. In the case of insert statements, each of the values are new and therefore need replication. For delete statements, this translates to only the primary key values being placed into the outbound queue (vs. the full before image as without minimal column replication) – which means any custom function strings (such as auditing) that is recording the values
being deleted in a history table will incur problems. Again, if not using custom function strings on the table, minimal column replication will not have a negative impact on RS functionality.

**Key Concept #14:** Unless custom function strings exist for update and delete functions for a specific table, minimal column replication should be considered. By using minimal columns, update operations at the replicate will proceed much quicker by avoiding unnecessary index maintenance and possibly avoiding updates altogether if the only columns updated at the primary are excluded from the replication definition.

**DIST Performance and Tuning**

Within each of the Distributor module discussions above, we covered tuning issues specific to that module. Overall, to monitor the performance or throughput of the Distributor thread, you can use the `admin who, dist` command.

```
admin who, dist
```

<table>
<thead>
<tr>
<th>Spid</th>
<th>State</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Active</td>
<td>102 SYDNEY_DS.SYDNEY_RSSD</td>
</tr>
<tr>
<td>22</td>
<td>Active</td>
<td>106 SYDNEY_DS.pubs2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PrimarySite</th>
<th>Type</th>
<th>Status</th>
<th>PendingCmds</th>
<th>SqtBlocked</th>
</tr>
</thead>
<tbody>
<tr>
<td>102</td>
<td>P</td>
<td>Normal</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>106</td>
<td>P</td>
<td>Normal</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Duplicates</th>
<th>TransProcessed</th>
<th>CmdsProcessed</th>
<th>MaintUserCmds</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>715</td>
<td>1430</td>
<td>0</td>
</tr>
<tr>
<td>290</td>
<td>1</td>
<td>293</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NoRepdefCmds</th>
<th>CmdsIgnored</th>
<th>CmdMarkers</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

The meaning for each of the columns is described below.

<table>
<thead>
<tr>
<th>Column</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>PrimarySite</td>
<td>The ID of the primary database for the SQT thread.</td>
</tr>
<tr>
<td>Type</td>
<td>The thread is a physical or logical connection.</td>
</tr>
<tr>
<td>Status</td>
<td>The thread has a status of “normal” or “ignoring.” You should only see “ignoring” during initial startup of the Replication Server.</td>
</tr>
<tr>
<td>PendingCmds</td>
<td>The number of commands that are pending for the thread. If the number of pending commands is high, then the DIST could be a bottleneck as it is not reading commands from the SQT in a timely manner.</td>
</tr>
<tr>
<td>SqtBlocked</td>
<td>Whether or not the thread is waiting for the SQT. This is the opposite of the above (PendingCmds). This essentially certifies that the DIST is not a cause for performance problems.</td>
</tr>
<tr>
<td>Duplicates</td>
<td>The number of duplicate commands the thread has seen and dropped. This should stop climbing once the Replication Server has fully recovered and the Status (above) changed from “ignoring” to “normal”.</td>
</tr>
<tr>
<td>TransProcessed</td>
<td>The number of transactions that have been processed by the thread.</td>
</tr>
<tr>
<td>CmdsProcessed</td>
<td>The number of commands that have been processed by the thread.</td>
</tr>
<tr>
<td>Column</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MaintUserCmds</td>
<td>The number of commands belonging to the maintenance user. This should be 0 unless the Rep Agent was started with the “send_maint_xacts_to_replicate” option.</td>
</tr>
<tr>
<td>NoRepdefCmds</td>
<td>The number of commands dropped because no corresponding replication definitions were defined. Usually an indication that a table/procedure is marked for replication but lacks a replication definition. If a procedure, this can be a key insight into why there may be database inconsistencies between a primary and replicate system.</td>
</tr>
<tr>
<td>CmdsIgnored</td>
<td>The number of commands dropped before the status became &quot;normal.&quot;</td>
</tr>
<tr>
<td>CmdMarkers</td>
<td>The number of special markers (rs_marker) that have been processed. Normally only noticed during replication system implementation such as adding a subscription or a new database.</td>
</tr>
</tbody>
</table>

As noted from the above command output, the DIST thread is responsible for matching LTL log rows against existing replication definitions to determine which columns should be ignored, etc. If the replication definition does not exist, it discards the log row at this stage. This is also when request functions are identified. The way this is detected is described in more detail later, however, if you remember from classes you have taken (or reading the manual), request functions have a replication definition specifying the real primary database which would not be the current connection processing the logged procedure execution. In any case, a large number of occurrences of NoRepdefCmds can mean one of two things:

- A replication definition was mistakenly dropped or never created. In either case, this means that the databases are probably suspect as they are definitely out of synch. Or…

- Tables or procedures were needlessly marked for replication. If this is the case, then a good, cheap performance improvement is to simply unmark the tables or procedure for replication. This will reduce Rep Agent processing, SQM disk i/o, SQT and DIST CPU time.
Outbound Queue Processing

...must come out.

The single biggest bottleneck in the Replication System is the outbound queue processing. As hard as this seems to be believed, the main reason for this is that the rate of applying transactions at the replicate will often be considerably slower than they were originally applied at the primary. While some of this is due to the replicated database tuning issues, a considerable part of it is also due to the processing of the outbound queue.

A key point to remember, is that when discussing the outbound processing of Replication Server internals, you are discussing threads and queues that belong to the replicate database connection and not the primary.

If you remember from the earlier internals diagram, the outbound processing basically includes the SQM for the outbound queue, the DSI thread group and the RSI thread for replication routes. These are illustrated below, with the exception of the RSI thread.

![Inbound/Outbound Division Diagram](image)

As you can imagine, the outbound queue SQM processing is extremely similar to the SQM processing for an inbound queue – basically manage stable device space allocation and perform all outbound queue write activity via the dAIO daemon. Consequently, we will begin by looking at the Data Server Interface (DSI) thread group in detail.

DSI SQT Processing

If you notice in the internals diagram above, unlike with the inbound processing, the outbound processing does not have a separate SQT thread. This is largely due to a very simple reason – transactions in the outbound queue are more than likely already in commit order. For example, if a source database is replicating to a single destination, the inbound SQT effectively sorts the transactions into commit sequence. Since this ordering is not overridden anywhere within the rest of the inbound processing, then the outbound queue is automatically in sorted order. This does not change if the primary has multiple replicates, since each replicate will have it’s own independent outbound queue that the single DIST thread is writing commit ordered transactions into. The only time this is not true is when multiple primary databases are replicating into the same replicate database – such as corporate rollup topologies. However, even in this latter case, due to MD caching of writes, providing that the transactions are small enough, the SQT will still encounter complete and contiguous transactions from each source system. If the transactions are not contiguous (replicated rows from the various sources inter-dispersed in the stable queue), the SQT will still only have a single transaction per origin in the Open/Closed/Read linked lists as the transactions are still in commit order respective to the source database. As a result, the main DSI thread queue manager (DSI) (normally called the DSI scheduler) simply calls the SQT functions when reading from the outbound queue via the SQM.
DSI SQT Performance Monitoring

This does not mean that you cannot monitor the SQT processing within the outbound queue processing. If you remember from previous, the admin who, sqt command reports both the inbound and outbound SQT processing statistics.

```
admin who, sqt
```

```
<table>
<thead>
<tr>
<th>Spid</th>
<th>State</th>
<th>Info</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>Awaiting Wakeup</td>
<td>TOKYO_DS.TOKYO_RSSD</td>
</tr>
<tr>
<td>98</td>
<td>Awaiting Wakeup</td>
<td>103:1 DIST LDS.pub2</td>
</tr>
<tr>
<td>10</td>
<td>Awaiting Wakeup</td>
<td>101 TOKYO_DS.TOKYO_RSSD</td>
</tr>
<tr>
<td>0</td>
<td>Awaiting Wakeup</td>
<td>106 SYDNEY_DS pub2sb</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Closed</th>
<th>Read</th>
<th>Open</th>
<th>Trunc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>Removed</th>
<th>Full</th>
<th>SQM Blocked</th>
<th>First</th>
<th>Trans</th>
<th>Parsed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>SQM Reader</th>
<th>Change Ogids</th>
<th>Detect</th>
<th>Orphans</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

In the above example output, the DSI SQT processing is reported in the last two lines lacking the queue designator (:1 or :0). The way this can easily be verified is by issuing a normal admin who command and comparing the spids (10 and 0 above) with the type of thread reported for those processes in the process list returned by admin who.

From a performance perspective, if you (hopefully) have tuned the Replication Server’s sqt_max_cache_size parameter (i.e. to 2MB), you may want to adjust the SQT cache for the outbound queue downward or up depending on the status of the removed an full columns in the admin who, sqt output. This can (and must) be done on a connection basis via setting the dsi_sqt_max_cache_size to a number differing from the sqt_max_cache_size. In the following sections we will take a look at why you might want to do either.

**dsi_sqt_max_cache_size < sqt_max_cache_size**

In low volume systems in which the primary transactions require a lot of sorting and are either very large or fairly small, you may wish to lower the amount of memory for the DSI SQT cache and provide that memory for other processes (such as the inbound SQT cache). The reason for this is that with a largely sorted outbound queue, much of the SQT cache may not even get used by the DSI thread. In these cases, you may want to begin with the DSI SQT cache significantly lower than the inbound SQT cache – however, still larger than the default. For example, if you determine through tuning that the inbound SQT cache works best when 4MB is available, you may only want to set the DSI SQT cache to 1MB.

**dsi_sqt_max_cache_size >= sqt_max_cache_size**

A notable exception to this is the Warm Standby implementation. In a WS topology, it is the DSI SQT thread that is actually sorting the transactions into commit order. In this case, you will probably want to set the DSI SQT cache equal to the SQT cache – or possibly even higher.

A second exception concerns the use of parallel DSI’s. When parallel DSI’s are used, the DSI thread can effectively process large amounts of row modifications as the load can be distributed among the several available DSI’s. This could result in a situation where the DSI transaction rate is higher than the amount of rows read from the outbound queue. In such situations, raising the DSI SQT cache allows the DSI to “read ahead” into the queue and begin preparing transactions before they are needed. This is especially true in high volume replication environments in which the rate of changes requires more than the default number of parallel DSI threads. In fact, consider the default...
DSI Transaction Grouping

Why Group Transactions

One function of the main DSI thread is to group multiple independent transactions from the primary into a single transaction group at the replicate. Consider the following illustration of the difference between the primary database transaction and the DSI transaction grouping:

<table>
<thead>
<tr>
<th>Primary Database Transactions</th>
<th>DSI Transaction Grouping</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin tran order_tran</td>
<td>begin tran</td>
</tr>
</tbody>
</table>
| insert into orders values (...)| insert into orders values (...)
| insert into order_items values (...)| insert into order_items values (...)
| update orders set total=...   | update orders set total=...|
| commit tran order_tran        | commit tran order_tran   |
| begin tran ship_tran          | begin tran ship_tran     |
| insert into ship_history values (...)| insert into ship_history values (...)
| Update orders set status=...  | Update orders set status=...|
| commit tran ship_tran         | commit tran ship_tran    |
| begin tran order_tran         | begin tran order_tran    |
| insert into orders values (...)| insert into orders values (...)
| insert into order_items values (...)| insert into order_items values (...)
| update orders set total=...   | update orders set total=...|
| commit tran order_tran        | commit tran order_tran   |
| begin tran order_tran         | begin tran order_tran    |
| insert into orders values (...)| insert into orders values (...)
| insert into order_items values (...)| insert into order_items values (...)
| update orders set total=...   | update orders set total=...|
| commit tran order_tran        | commit tran order_tran   |
| begin tran order_tran         | begin tran order_tran    |
| insert into orders values (...)| insert into orders values (...)
| insert into order_items values (...)| insert into order_items values (...)
| update orders set total=...   | update orders set total=...|
| commit tran order_tran        | commit tran order_tran   |

Figure 26 – Primary vs. Replicate Transaction Nesting Impact of DSI Transaction Grouping

In the example on the right, Replication Server’s DSI thread has nested the individual transactions into another transaction (begin/commit pair underlined) grouping the transactions together. The obvious question is “Why bother doing this?” The answer simply is to decrease the amount of logging on the replicate system imposed by replication and to improve the transaction delivery rate. Consider the worst-case scenario of several atomic transactions such as:

```
insert into checking_acct values (123456789,000001,"Sep 1 2000 14:20:36.321","$125.00,Chk,101)
insert into checking_acct values (123456789,000002,"Sep 1 2000 14:20:36.322","$250.00,Chk,102)
insert into checking_acct values (123456789,000003,"Sep 1 2000 14:20:36.323","$395.00,Chk,103)
insert into checking_acct values (123456789,000004,"Sep 1 2000 14:20:36.324","$12.00,Chk,104)
insert into checking_acct values (123456789,000005,"Sep 1 2000 14:20:36.325","$99.00,Chk,105)
insert into checking_acct values (123456789,000006,"Sep 1 2000 14:20:36.326","$5.32,Chk,106)
insert into checking_acct values (123456789,000007,"Sep 1 2000 14:20:36.327","$119.00,Chk,107)
insert into checking_acct values (123456789,000008,"Sep 1 2000 14:20:36.328","$1132.00,Chk,108)
```

As you notice, these fictitious transactions all were applied during an extremely small window of time. Now the question is, without transaction grouping, what would Replication Server do? The answer is, each of the above would get turned into separate individual transactions and submitted as follows (RS functions listed vs. SQL):

```
rs_begin
rs_insert - insert for check 101
rs_commit
rs_begin
rs_insert - insert for check 102
rs_commit
rs_begin
rs_insert - insert for check 103
```
rs_commit
rs_begin
rs_insert – insert for check 104
rs_commit
rs_begin
rs_insert – insert for check 105
rs_commit
rs_begin
rs_insert – insert for check 106
rs_commit
rs_begin
rs_insert – insert for check 107
rs_commit
rs_begin
rs_insert – insert for check 108
rs_commit

Which does not look that bad until you realize two very interesting facts: 1) the contents of the rs_commit function; and 2) how rs_commit is sent as compared to other functions. In regards to the former, rs_commit calls a stored procedure rs_update_lastcommit, which updates the corresponding row in the replication system table rs_lastcommit. As far as the second point, while this will be discussed in more detail in the next section, Replication Server does not batch the outer commit statements with the transaction batch if batching is enabled. Consequently, the replicate database would actually be executing something similar to:

```
begin tran
insert into checking_acct {...,101}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,102}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,103}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,104}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,105}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,106}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,107}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
begin tran
insert into checking_acct {...,108}  
-- wait for success
update rs_lastcommit ...
commit transaction  
-- wait for success
```

Why is this a problem? First, the amount of I/O has clearly doubled. Consequently, if the replicate system was already experiencing I/O problems, this would add to the problem. Secondly, the delivered transaction rate would not match that at the primary system. Consider each of the following primary database transaction scenarios:
**Concurrent User** – Concurrent users applied each transaction at the primary. At the replicate, only a single user is applying the transactions. So while the primary system can take full advantage of multiple CPU’s, group commits for the transaction log and every other feature of ASE to improve concurrency, the replicate simply has no concurrency.

**Single User/Batch** – In this scenario, a single user applies all the transactions at the primary in a large SQL batch. At the replicate, the batching is essentially undone as each of the atomic commits results in 2 network operations per transaction. This could be significant as anyone familiar with the performance penalties of not batching SQL can attest.

**Single User/Atomic** – A single user performs each of the original inserts using a single atomic transaction per network call. While the replicate might appear to be similar, consider the following. As ASE performs each I/O the user process is put to sleep. As a result, the replicate system – with twice the i/o’s – will spend twice as much time “sleeping”, consequently halving its ability to process transactions.

Simply, transaction batching is critical to replication performance – although it can be an issue with parallel or multiple DSI’s as discussed later.

**Key Concept #15:** Transaction grouping reduces I/O caused by updating replication system tables and the corresponding logging overhead at the replicate system. This also improves throughput as the replication process within the replicate database server spends less time waiting for I/O completion.

**DSI Transaction Grouping Rules**

Unfortunately, not every transaction can be grouped together. A transaction group will end any time one of the following conditions is met:

1. There are no more transactions in the DSI queue.
2. The predefined maximum number of transactions allowed in a group has been reached.
3. The current or the next transaction will make the total size of the transactions (in bytes) exceed the configured group size.
4. The next transaction is from a different origin.
5. The current or the next transaction is on disk.
6. The current or the next transaction is an orphan transaction.
7. The current or the next transaction is a rollback.
8. The current or the next transaction is a subscription (de)materialization transaction marker.
9. The current or the next transaction is a subscription (de)materialization transaction queue end marker.
10. The current or the next transaction is a dump/load transaction.
11. The current or the next transaction is a routing transaction.
12. The current or the next transaction has no begin command (i.e., it is a special RS-to-RS transaction).
13. The next transaction has a different user/password.
14. The first transaction has IGNORE_DUP_M mask on.

While this appears to be quite a long list, the rules for grouping transactions can simply be paraphrased into the rule that in order for transactions to be batched together, all of the following five conditions must be met.

1. Transactions cached in the DSI/SQT closed queue.
2. Transactions from the same origin.
3. Transactions will be applied at the replicate with the same username and password. This applies to asynchronous request functions only.
4. The transaction group size is limited by the lesser of `dsi_xact_group_size` and `dsi_max_xacts_in_group`.

5. Aborted, database/log dump, orphan, routing, and subscription transactions cannot be grouped.

The third condition will be discussed in more detail in the section on asynchronous request functions, so for now, we will ignore it. The fourth condition will be discussed in the next section on tuning transaction grouping. The last condition is due to system level reprocessing or ensuring integrity of the replicate system during materialization of subscriptions or routes and is rare – consequently not discussed. This leaves only the first two conditions that apply to most transactions.

While the first condition makes sense simply from a performance aspect, the second condition requires some thought. The reason that the transactions have to be from the same origin is due to the management of the `rs_lastcommit` table and how the DSI controls assigning the OQID for the grouped transaction. When the DSI groups transactions together, it uses the last grouped transaction’s begin record to determine the OQID for the grouped transaction. The reason is that on recovery, not using the last transaction’s OQID could result in duplicate row errors or an inconsistent database.

Consider a default grouping of 20 transactions into a single group that are applied to the replicate database server and then immediately the replicate database shuts down. On recovery, as most people are aware, the Replication Server will issue a call to `rs_get_lastcommit` to determine the last transaction that was applied. Remember, the transactions are grouped in memory – not in the stable queue. Consequently, if the OQID of the first transaction was used, then the first 19 transactions would all be duplicates – and not detected as such by the Replication Server as that was the whole reason for the comparison of the OQID in the first place!! As a result, the first 19 transactions would either cause duplicate key errors (if you are lucky) or database inconsistencies if using function strings. For that reason, when transactions are grouped together, the OQID of the last transaction’s begin record is used for the entire group.

Now then, following that logically along, since the `rs_commit` function updates only a single row in the `rs_lastcommit` table for the source database of the transaction, then all of the transactions grouped together must be from the same source. Note that currently, the DSI does not simply collect all of the closed transactions from the same source. If the third transaction in a series is from a different source database, then the group will end at two – even if the next four transactions are from the same source database as the first two. As you can imagine, a fragmented queue with considerable inter-dispersed transactions from different databases, the DSI will be applying transactions in very small groups. As mentioned earlier, the smaller the group size, the less efficient the replication mechanism due to `rs_lastcommit` and processing overhead, which leads us to the following concept:

**Key Concept #16:** Outbound queues that are heavily fragmented with inter-dispersed transactions from different source databases will not be able to effectively use transaction grouping

**Tuning DSI Transaction Grouping**

Prior to Replication Server 12.0, however, there really wasn’t a good way to control the number of transactions in a batch. The reason was that the only tuning parameter available attempted to control the transaction batching by controlling the transaction batch size in bytes – a difficult task with tables containing variable width columns and considering the varying row sizes of different tables. With version 12.0 came the ability to explicitly specify the number of original transactions that could be grouped into a larger transaction. These connection level configuration parameters are listed below.

<table>
<thead>
<tr>
<th>Parameter (Default)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dsi_xact_group_size</code> (65,536)</td>
<td>The maximum number of bytes, including stable queue overhead, to place into one grouped transaction. A grouped transaction is multiple transactions that the DSI applies as a single transaction. A value of &quot;-1&quot; means no grouping.</td>
</tr>
<tr>
<td>Parameter (Default)</td>
<td>Explanation</td>
</tr>
<tr>
<td>--------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>dsi_max_xacts_in_group (20)</td>
<td>Specifies the maximum number of transactions in a group, allowing a larger transaction group size, which may improve data latency at the replicate database. The default value is 20 and the maximum value is 100.</td>
</tr>
<tr>
<td>dsi_sqt_max_cache_size (0) (sqt_max_cache_size)</td>
<td>The number of bytes available for managing the SQT open, closed, read and truncate queues. This impacts DSI SQT processes by also being a limiter on the number of bytes in a transaction. For example, if the DSI SQT cache is too small, the number of “open” or “read” transactions will reduce the amount of cache for “closed” transactions to less than dsi_xact_group_size.</td>
</tr>
</tbody>
</table>

At first, the dsi_xact_group_size may appear to be fairly large. Remember, however, this includes stable queue overhead – which can be significant as the queue may require 4 times the storage space as the transaction log space. As a result, you may wish to tune this parameter to a higher value – especially if adjusting the dsi_max_xacts_in_group to a higher number. Additionally, consider the following:

- **dsi_xact_group_size** – Typically in RS 12.0 and higher, this parameter should be left alone unless adjusting the dsi_max_xacts_in_group to a higher number, or if tables frequently involved in transactions have a large row size (i.e. >1,000 bytes).
- **dsi_max_xacts_in_group** – If using a single DSI, this can be raised from the default of 20 – and perhaps should be if system is performing a lot of small transactions. However, in parallel or multiple DSI situations, this parameter may need to be lowered to reduce inter-thread contention.

This also kind of leads us back to the discussion on the MD library and the md_source_memory_pool setting. As mentioned then, it doesn’t have much benefit if the replicate system is only being replicated into by a single source. However, when multiple sources are involved, it helps “group” transactions from the same source closer together, and consequently helps with transaction grouping.

### DSI Function String Generation

### DSI Executer Processing

While the DSI is responsible for SQT functions and transaction grouping, it is the responsibility of the DSI Executer (DSI-E) threads to actually perform the SQL string generation, command batching and exception handling. If you remember from the earlier discussion on LTL, the replicated functions (rs_insert, rs_update, rs_delete, etc.) actually are identified by the Replication Agent. This helps the rest of the Replication Server as it does not have to perform SQL language parsing (which is not in the transaction log anyhow – something many people have a hard time understanding – the transaction log NEVER logs the SQL). However, we need to send ASCII language commands to the replicate system (or RPC’s). As a result, the DSI-E thread execution looks like the following flow diagram.
Note that in the above diagram, only “stop” errors cause the DSI to suspend. If you remember, some error actions such as ignore (commonly set to handle database change, print and other information messages), retry, etc. allow the DSI to continue uninterrupted.

**DSI Executer Performance**

Beyond DSI command batching (next section), the tuning parameters available for the DSI Executer are listed in the following table (other parameters are available, however, do not specifically address performance throughput). Note that parameters specific to parallel DSI performance are not listed here.

<table>
<thead>
<tr>
<th>Parameter (Default)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Replication Server scope</strong></td>
<td></td>
</tr>
<tr>
<td><strong>sts_cachesize</strong> (100)</td>
<td>The total number of rows cached for each cached RSSD system table. Increasing this number to the number of active replication definitions prevents Replication Server from executing expensive table lookups. From a DSI Executer performance perspective, the STS cache could be used to hold RSSD tables such as rs_systext that hold the function string definitions. Of all the parameters below, this one is probably the most critical as insufficient STS cache would result in network and potentially disk i/o in accessing the RSSD.</td>
</tr>
<tr>
<td><strong>fstr_cachesize</strong> (200,000)</td>
<td>Size of function string cache, in bytes. The value must be greater than 0. This value was used to specifically set the number of bytes for function string storage while sts_cachesize was more generic. In RS 12.0, it was decided that this was not necessary (possibly viewed as duplicative as function string RSSD rows would be in STS cache as well) and the parameter was made obsolete (although still in the documentation).</td>
</tr>
</tbody>
</table>

*Note: fstr_cachesize parameter was obsolete in 12.0.*
<table>
<thead>
<tr>
<th>Parameter (Default)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection scope</strong></td>
<td></td>
</tr>
<tr>
<td>batch (on)</td>
<td>Specifies how Replication Server sends commands to data servers. When batch is &quot;on,&quot; Replication Server may send multiple commands to the data server as a single command batch. When batch is &quot;off,&quot; Replication Server sends commands to the data server one at a time. This is &quot;on&quot; for ASE and should be on for any system that supports command batching due to performance improvements of batching. Some heterogeneous replicate systems – such as Oracle – do not support command batching, and consequently this parameter needs to be set to &quot;off&quot;</td>
</tr>
<tr>
<td>batch_begin (on)</td>
<td>Indicates whether a begin transaction can be sent in the same batch as other commands (such as insert, delete, and so on). If batch is &quot;on&quot; and the replicate supports command batching, this value should be &quot;on&quot;.</td>
</tr>
<tr>
<td><strong>db_packet_size</strong> (512)</td>
<td>The maximum size of a network packet. During database communication, the network packet value must be within the range accepted by the database. You may change this value if you have a System 10 or later SQL Server or Adaptive Server that has been reconfigured. A recommended packet size of 16,384 on high speed networks or tuned to network MTU on lower speed networks is appropriate. Values less than 2,048 are suspect and should only be used if the target system does not support larger packet sizes.</td>
</tr>
<tr>
<td>dsi_cmd_batch_size (8192)</td>
<td>The maximum number of bytes that Replication Server places into a command batch.</td>
</tr>
<tr>
<td>dsi_fadeout_time (600)</td>
<td>The number of seconds of idle time before a DSI connection is closed. A value of &quot;-1&quot; indicates that a connection will never fade out.</td>
</tr>
<tr>
<td><strong>dsi_keep_triggers</strong> (on)</td>
<td>Specifies whether triggers should fire for replicated transactions in the database. Set to &quot;off&quot; to cause Replication Server to set triggers off in the Adaptive Server database, so that triggers do not fire when transactions are executed on the connection. Set to &quot;on&quot; for all databases except standby databases. Arguably should be off for all databases. &quot;On&quot; is the default as it is the typical &quot;safe&quot; approach that Replication Server defaults assume, however, there should be compelling reasons not to have this turned &quot;off&quot; – including security as the replication maintenance user could be viewed as a &quot;trusted agent&quot; fully supportable in Bell-LaPadula and other NCSC endorsed security policies.</td>
</tr>
<tr>
<td>dsi_replication (Most: off; WS: on)</td>
<td>Specifies whether or not transactions applied by the DSI are marked in the transaction log as being replicated. When dsi_replication is set to &quot;off,&quot; the DSI executes set replication off in the Adaptive Server database, preventing Adaptive Server from adding replication information to log records for transactions that the DSI executes. Since these transactions are executed by the maintenance user and, therefore, not usually replicated further (except if there is a standby database), setting this parameter to &quot;off&quot; avoids writing unnecessary information into the transaction log. dsi_replication must be set to &quot;on&quot; for the active database in a warm standby application for a replicate database, and for applications that use the replicated consolidated replicate application model. The reason this is mentioned as a possible performance enhancement is its applicability in multiple DSI situations discussed later.</td>
</tr>
<tr>
<td>Parameter (Default)</td>
<td>Explanation</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dsi_sqt_max_cache_size (0)</td>
<td>Maximum SQT (Stable Queue Transaction interface) cache memory for the database connection, in bytes.</td>
</tr>
<tr>
<td></td>
<td>The default, &quot;0,&quot; means that the current setting of sqt_max_cache_size is used as the maximum cache size for the connection.</td>
</tr>
<tr>
<td></td>
<td>This parameter controls the use of parallel DSI threads for applying transactions to a replicate data server.</td>
</tr>
<tr>
<td>dsi_xact_group_size (65,536)</td>
<td>The maximum number of bytes, including stable queue overhead, to place into one grouped transaction. A grouped transaction is multiple transactions that the DSI applies as a single transaction. A value of &quot;-1&quot; means no grouping.</td>
</tr>
</tbody>
</table>

Some of these have been discussed before and have been included here simply for completeness. Additionally, several have to do with command batching which is discussed in the next section. Those that are highlighted are specifically applicable to DSI Executer performance.

**DSI Command Batching**

In addition to transaction grouping, another DSI feature that is critical for replication throughput is DSI command batching. While some database systems, such as Oracle, do not allow this feature – those that do gain a tremendous advantage in reducing network I/O time by batching all available SQL and sending it to the database server in a single structure. This is analogous to executing the following from isql:

```sql
-- isql script
insert into orders values (…)
insert into order_items values (…)
insert into order_items values (…)
insert into order_items values (…)
insert into order_items values (…)
insert into order_items values (…)
insert into order_items values (…)
insert into order_items values (…)
```

vs. without command batching, the same isql script would look like:

```sql
-- isql script
insert into orders values (…)
go
insert into order_items values (…)
go
insert into order_items values (…)
go
insert into order_items values (…)
go
insert into order_items values (…)
go
insert into order_items values (…)
go
insert into order_items values (…)
go
```

Anyone with basic performance and tuning knowledge will be able to tell that the first example will execute an order of magnitude faster from the client application perspective. How does this apply to Replication Server? Believe it or not, it does NOT mean that multiple transaction groups can be lumped into a large SQL structure and sent in a single batch. It does mean, however, that all of the members of a single transaction group may be sent as a single command batch – with the exception of the final commit (due to recovery reasons. The commit is withheld until all transaction statements have executed without error. If no errors, then the commit is sent separately. If errors occurred, either a rollback is issued or the DSI connection is suspended (most common) which implicitly rolls back the transaction). The way this works is as follows:

1. The DSI groups a series of transactions until one of the group termination messages is hit (for example, the maximum of 65,536 bytes).
2. The DSI executes the grouped transaction by sending `dsi_cmd_batch_size` (8192 bytes by default) sized command batches until completed.

In the example above, if the transaction group was terminated due to 65,536 byte limitation, it would be sent to the replicate database in ~8 batches of 8,192 bytes (depending on command boundaries as ASE requires that commands must be complete and not split across batches). Consequently, the effect of command batching – which is on by default for ASE replicates – is that performance of each of the transaction groups is maximized by reducing the network overhead of sending a large number of statements within a single transaction. In this way, lock time on the replicate is minimized, reducing contention between parallel DSI’s as well as contention with normal replicate database users.

Command batching is critical for large transaction performance. As you could imagine, a large transaction – especially one that gets removed from SQT due to cache limitations – will force the previous transaction group to end. As the large transaction begins, each `dsi_cmd_batch_size` bytes will be sent to the replicate database.

It might be tempting then to assume that it would be best to set `dsi_cmd_batch_size` to the same as `dsi_xact_group_size` or 65,536 by default. One problem that many people who have coded large stored procedures might remember about this – each user connection has a limited stack size in ASE for their connection. Issuing too large a batch of SQL results in stack overflow. Besides which, the best setting for `dsi_cmd_batch_size` is to set it and `db_packet_size` to the same setting for today’s faster networks – 16,384. This is partially because TCP/IP has a 16K byte limitation on many systems for the packet size, consequently larger packet sizes may be ineffective. On slower (i.e. standard 10-BaseT Ethernet or lower), the best settings would probably be setting the `dsi_cmd_batch_size` to 16,384 and tuning the `db_packet_size` to the MTU (~1,520 bytes) by setting it to 2,048. The `dsi_cmd_batch_size` should rarely (hesitating to say “never” only to avoid setting a precedence) be set to less than 8,192 no matter what `db_packet_size` is set to- and never less than 2,048 as a large data row size of >1,000 bytes might easily violate this by the time the column names, etc. are added to the command. Remember, even with the default of 512 bytes (how many of us typically set the “-A” to higher??), `isql` is faster executing batches of SQL than individual statements. So lowering `dsi_cmd_batch_size` to `db_packet_size` is typically will degrade throughput.

**Key Concept #17:** Along with transaction grouping, DSI command batching is critical to throughput to replicate systems that support it. In fact, the two working together offer the best throughput.

However, command batching is only available to language commands and not RPC calls. This is important for sites that are customizing the function string output template for replication definitions and considering using RPC calls vs. language commands for executing the procedures. The limitation actually stems from ASE’s handling of batch SQL. ASE assumes that an RPC call is atomic and that then entire message is related to the RPC call (either the procedure name or parameters) while for language commands, ASE looks at the white space and reserved word offset to determine the number of commands in the batch. Consequently, while language commands can be batched, RPC calls can not. While this may seem at odds with traditional logic arguing that RPC’s offer performance advantages through pre-compiled command processing, the performance advantages of batch SQL outweigh the slight performance gain of command parsing.

**RSI Queue Processing**

*Author’s Note: This probably should be in its own section with a more complete description of replication routing, including the earlier discussion concerning routing and performance (the SMP style discussion) and a fuller description of the internals (i.e. for example, the RSI User thread calling the MD library calls similar to DSI calling SQT calls), etc. For this version, however, we will need to be content with simply focusing on the message processing and configuration parameters.*

The Replication Server Interface (RSI) thread is the thread that reads from the outbound queue and sends transactions to replicate replication servers used for routing. Unlike the DSI interface, the RSI interface is non-transactional in nature. For example, it does not make SQT calls and does not base delivery on completed transactions. Instead, it operates much on the same principals of a Replication Agent – it simply passes the row modifications as individual messages to the replicate Replication Servers and tracks recovery on a message id basis (and consequently, it is the only mechanism in Replication Server in which orphan transactions can happen – due to a data loss in the outbound queue mainly).
The following configuration parameters are available for tuning replication routing.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rsi_batch_size</td>
<td>The number of bytes sent to another Replication Server before a truncation point is requested. The range is 1024 to 262,144. This works similar to the Replication Agent’s scan_batch_size configuration setting. This normally should not be adjusted downwards unless in a fairly unstable network environment and want the RSI outbound queue to be kept trimmed. Default: 262,144 bytes</td>
</tr>
<tr>
<td>rsi_fadeout_time</td>
<td>The number of seconds of idle time before Replication Server closes a connection with a destination Replication Server. In low volume routing configurations this may be set lower to reduce connection processing in the replicate Replication Server. Default: -1 (specifies that Replication Server will not close the connection)</td>
</tr>
<tr>
<td>rsi_packet_size</td>
<td>Packet size, in bytes, for communications with other Replication Servers. The range is 1024 to 8192. In high-speed networks, you may want to boost this to 8192. The RSI uses an 8K send buffer to hold pending messages to be sent. When the number of bytes in the buffer will exceed the packet size, the send buffer is flushed to the replicate RS. Default: 2048 bytes</td>
</tr>
<tr>
<td>rsi_sync_interval</td>
<td>The number of seconds between RSI synchronization inquiry messages. The Replication Server uses these messages to synchronize the RSI outbound queue with destination Replication Servers. Values must be greater than 0. This is analogous to the 16K buffer transfer of the Replication Agent. Default: 60 seconds</td>
</tr>
</tbody>
</table>
Replicate DataServer/Database

You gotta tune this too!!

Often when people are quick to blame the Replication Server for performance issues, it turns out the real cause of the problem is the replicate database. As with any client application, the lack of a tuned replicate database system really impedes transaction delivery rates. Two things contribute to the Replication Server’s quick blame for this – 1) As a strictly write intensive process, poor design is quickly evident; and 2) administrators will monitor replication delivery rates quicker than DBMS performance. In fact, it is an extremely rare database shop these days that regularly monitors their system performance beyond the basic CPU loading and disk I/O metrics.

Key Concept #18: Not only is a well tuned replicate dataserver crucial to Replication Server performance, but a well instrumented primary and replicate dataserver is critical to determining the root cause of performance problems when they do occur.

The purpose of this section is not to discuss how to tune a replicate dataserver as that can be extremely situational dependent. However, several points to consider and common problems associated with replication will be discussed.

Performance Monitoring

For ASE based systems, it is critical to have Historical Server and Monitor Server set up for performance monitoring of the primary, RSSD and replicate dataservers. Because of the filtering capabilities (i.e. filter by user name specifying the maintenance user login), you can get a clear picture of replication specific activity. For example, you can compare the number of physical writes on an object by object basis at the primary with the physical writes at the replicate to determine the relative ability of the replication system to maintain the rate. However, keep in mind that replication may alter some of the statistics due to transaction grouping, command batching, etc. A partial list of items to monitor includes:

- RSSD activity, particularly queries against tables that should be in STS cache
- Replicate dataserver blocking/contention involving the maintenance users
- Physical write activity per object at primary and per object done by maintenance user at replicate
- Time maintenance user spends in runnable or sleep state in the process queue
- Stored procedure execution times and frequency (for replicated stored procedures)
- Logical i/o per object performed by maintenance user (identify table scans at replicate)

Several different Historical Server views could be established to use as necessary. Keep in mind that continuous detailed monitoring is not necessary – but crucial during periods of peak demand when performance issues are suspect. Example HS views could be similar to:

```sql
-- this view tracks the amount of physical and logical I/O a maintenance user performs
hs_create_view process_object_page_io,
"Login Name", "Value for Sample",
"Process ID", "Value for Sample",
"Kernel Process ID", "Value for Sample",
"Database Name", "Value for Sample",
"Database ID", "Value for Sample",
"Object Name", "Value for Sample",
"Object ID", "Value for Sample",
"Object Type", "Value for Sample",
"Owner Name", "Value for Sample",
"Page I/O", "Value for Sample",
"Page Hit Percent", "Value for Sample",
"Logical Page Reads", "Value for Sample",
"Physical Page Reads", "Value for Sample",
"Index Logical Reads", "Value for Sample",
"Physical Index Reads", "Value for Sample",
"Logical Index Reads", "Value for Sample",
"Logical Index Reads", "Value for Sample",
"Logical Index Reads", "Value for Sample"
go
hs_create_filter process_object_page_io, "Login Name", "Value for Sample", "maintuser_name"
```
-- this view tracks the procedure execution and I/O performed by the maintenance user
hs_create_view process_procedure_page_io,
"Login Name", "Value for Sample",
"Process ID",
"Kernel Process ID",
"Procedure Database Name",
"Procedure Database ID",
"Procedure Name",
"Procedure ID",
"Procedure Execution Count",
"Procedure CPU Time",
"Procedure CPU Time",
"Procedure Elapsed Time",
"Procedure Elapsed Time",
"Page I/O",
"Page Hit Percent",
"Logical Page Reads",
"Index Logical Reads",
"Physical Page Reads",
"Index Physical Reads",
"Page Writes",

hs_create_filter process_procedure_page_io, "Login Name", "Value for Sample", "maintuser_name"

-- this view tracks the lock contention involving the maintenance user
hs_create_view process_lock_contention,
"Login Name",
"Process ID",
"Kernel Process ID",
"Database Name",
"Database ID",
"Object Name",
"Object ID",
"Object Type",
"Owner Name",
"Lock Type",
"Lock Status",
"Page Number",
"Locks Being Blocked Count",
"Blocking Process ID",
"Demand Lock",

hs_create_filter process_lock_contention, "Lock Status", "Value for Sample", eq, 1, 3
hs_create_filter process_lock_contention, "Login Name", "Value for Sample", "maintuser_name"

-- this view tracks when the maintenance user had to issue a demand lock
hs_create_view demand_locks_obj,
"Process ID",
"Login Name",
"Object ID",
"Database Name",
"Owner Name",
"Lock Status",
"Lock Type",
"Page Number",
"Demand Lock",
"Blocking Process ID",

hs_create_filter demand_locks_obj, "Login Name", "Value for Sample", "maintuser_name"
hs_create_filter demand_locks_obj, "Demand Lock", "Value for Sample", eq, "Y"

-- this view monitors maintenance user general activity
hs_create_view process_activity,
"Login Name",
"Process ID",

hs_create_filter process_activity, "Login Name", "Value for Sample"
go

Other views can be created on an as needed basis. In addition, since Historical Server generates table schemas for storing the data in SQL database, the data can be loaded into a database and more advanced analysis executed – for example to determine if the blocking process was a maintenance user process for parallel DSI contention (resolvable as discussed later).

**Replicate DataServer Resources**

In addition to normal dataserver and transaction tuning, you can help the replication server performance through adjusting resource governing. For example, by analyzing the output of the last HS view, you may find that the maintenance user is spending very little actual CPU time and a considerable time in a “runnable” state vs. “running”. Attempting to improve Replication Server performance would not help at all. Instead, you need to adjust the amount of resources that the maintenance user is getting. This can be accomplished in several ways:

- **Execution Priority** - by raising the execution priority of the maintenance user login, the Replication Server will get more than it’s fair share of CPU cycles on a platform that other “normal” application users are also utilizing. This is a good solution to try in OLTP systems where Replication Server is being used to automate workflow.

- **Engine Access** – you may need to restrict the engine access of other applications to free up CPU resources for the maintenance user. For example, if replicating to a DSS system with parallel query enabled, you may wish to restrict the DSS query users to 80% of the number of CPU’s (i.e. 8 CPU’s on a 10 CPU system) and the other engines reserved for maintenance user or unrestricted the maintenance user to run on all the available engines.

- **RS Data Cache** – set up a separate data cache for replication specific objects such as rs_threads, rs_lastcommit, etc. This will prevent them from being flushed out of cache vs. tables used for other queries on the replicate system.

**Trigger/Procedure Execution Time**

Trigger and procedure execution time are extremely, extremely critical. Remember, in order to maintain commit order, the Replication Server basically applies the transactions in sequence – even in parallel DSI scenarios, the threads block and wait for the commit order. As a result, while procedure execution is great for Replication Server performance from thread processing perspective, the net effect is that as soon as a long procedure begins execution, the following transactions in the queue effectively are delayed. Note, that this is not unique to stored procedures – long running transactions will have the same effect (i.e. replicating 50,000 row modifications in a single transaction vs. a procedure that modifies them have the same effect at the replicate system – however, the procedure is much less work for the Replication Server processing).

As a result, particular attention should be paid to stored procedure and trigger execution times (if you for some odd reason opt not to turn triggers off for that connection). Any stored procedure that employs cursors, logged I/O in tempdb, etc. should be candidates for rewriting for performance.

**Procedures with “Select/Into”**

The latter example probably raised a quick “but..but..” from developers who are quick to state that replicating procedures with “select/into..” is not possible due to “DDL in transaction” errors at the replicate system. Very true if procedure replication is only at the basic level – which typically is not the optimal strategy for procedure replication. While this may seem to be more appropriately discussed in the primary database section earlier, the transaction “wrapping” effect of Replication Server has often caused application developers to change the procedure logic at the primary. Case in point, procedures with select/into execute fine at the primary, however, fail at the replicate due to DDL in tran errors. Many developers then are quick to re-write both to eliminate the select/into – not only affecting the
performance at the replicate, but also endangering performance at the primary. So, in a way, it does make sense to discuss it here.

The best way to decide what to do with procedures containing “select/into” is by assessing the number of physical changes actually made to the real tables the procedure modifies and the role of the worktable created in tempdb. Several scenarios are discussed in the following sections. A summary table is included first for ease of reference between the scenarios.

<table>
<thead>
<tr>
<th>Solution</th>
<th>Applicability</th>
</tr>
</thead>
<tbody>
<tr>
<td>replicate tables vs. procedure</td>
<td>• complex (long run time) row identification</td>
</tr>
<tr>
<td></td>
<td>• small number of real rows modified</td>
</tr>
<tr>
<td>Work table &amp; subprocedure</td>
<td>• complex (long run time) row identification</td>
</tr>
<tr>
<td></td>
<td>• small number of rows in work table</td>
</tr>
<tr>
<td></td>
<td>• large number or rows in real tables</td>
</tr>
<tr>
<td>procedure rewrite without select/into</td>
<td>• row identification easy</td>
</tr>
<tr>
<td></td>
<td>• work tables contain large row counts</td>
</tr>
<tr>
<td></td>
<td>• large number of rows modified in real table</td>
</tr>
</tbody>
</table>

**Replicate Affected Tables vs. Procedures**

In this case, it is a classic case of replicating the wrong object. In some cases, the stored procedure may use a large number of temporary tables to identify which rows to modify or add to the real database in a “list paring” concept. In this case, the final number of rows affected in replicated tables is actually fairly small. Consider the following example:

*Update all of the tax rates for minority owned business within the tax-free empowerment zone to include the new tax structures.*

Since these empowerment zones typically encompass only an area several blocks in size, the number of final rows affected will probably be only a couple dozen. However, the logic to identify the rows may be fairly complicated (i.e. a certain linear distance from a epicenter) and may require “culling” down the list of prospects using successive temp tables until only the desired rows are left. For example, the first worktable may be a table simply to get a list of businesses and their range to the epicenter – possibly using the zip code to reduce the initial list evaluated. The second list would be constrained to only those within the desired range that are minority owned. The pseudo code would look something like:

```sql
select business_id, minority_owner_ship, (range formula) into #temptable_1
from businesses
where zip_code in (12345,12346)

select business_id, minority_owner_ship, distance into #temptable_2
from #temptable_1
where distance < 1
and minority_owner_ship > 0.5

update businesses
set tax_rate = tax_rate -.10
from #temptable_2 t2, businesses b
where b.business_id=t2.business_id
```

Now, lets take a look at what if this was in a procedure. The first temporary table creation might take several seconds simply due to the amount of data being processed and the second may also take several seconds due to the table scan that would be required for the filtering of data from the first temp table. The net effect would be a procedure that requires (just for sake of discussion) possibly 20 seconds for execution – 19 of which are the two temp table creations. The decision to replicate the rows or the procedure then becomes on of determining whether the average number of rows modified by the procedure take longer to replicate than the time to execute the procedure at the replicate. For instance, let’s say that when executed, the average execution of the procedure is 20 seconds modifying 72 rows. If it takes 10 seconds to move the 72 rows through Replication Server and another 13 seconds to apply the rows via the DSI, it still may be better to replicate the rows vs. changing the procedure to use logged I/O and permanent worktables as that might slow down the procedure execution to 35 seconds.
Worktable & Subprocedure Replication

However, in many cases, it is simply too much to replicate the actual rows modified. Take the above example again, only this time, let's assume that the target area contains thousands of businesses. Replicating that many rows would take too long. However, think of the logic in the original procedure at the primary:

1. Step 1 – Identify the boundaries of the area
2. Step 2 – Develop list of businesses within the boundaries
3. Step 3 – Update the businesses tax rates

Now think about it. Step 1 really needs a bit more logic. In this example, identifying the boundaries as the outer cross streets does not help you identify whether an address is within the boundary unless employing some form of grid system ala Spatial Query Server (SQS). The real logic would probably be more likely:

1. Step 1 – Identify the outer boundaries of the area
2. Step 2 – Identify the streets within the boundaries
3. Step 3 – Identify the address range within each street
4. Step 4 – Develop list of businesses with address between range on each street
5. Step 5 – Update the businesses tax rates

Up through step 3, the number of rows are fairly small. Consequently the logic for a stored procedure could be similar to:

(Outer procedure – outer boundaries as parameters)

Inner procedure

Update business tax rate where address between range and on street.

As a result, you simply need to replicate the worktable containing the street number ranges and the inner procedure. The procedure at the primary then might look like:

create procedure set_tax_rate
    @streetnum_n int, @street_n varchar(50),
    @streetnum_s int, @street_s varchar(50),
    @streetnum_e int, @street_e varchar(50),
    @streetnum_w int, @street_w varchar(50),
    @target_demographic varbinary(255),
    @new_tax_rate decimal(3,3)
as begin
    -- logic to identify N-S streets in boundary using select/into
    -- logic to identify E-W streets in boundary using select/into
    begin tran
    insert into street_work_table
        select @@spid, streetnum_n, streetnum_s, streetname
        from #NS_streets
    union all
    select @@spid, streetnum_e, streetnum_w, streetname
        from #EW_streets
    exec set_tax_rate_sub @@spid, @target_demographic, @new_tax_rate
    commit tran
    return 0
end
create procedure set_tax_rate_sub
    @proc_id int,
    @target_demographic varbinary(255),
    @new_tax_rate decimal(3,3)
as begin
    update businesses
    set tax_rate= @new_tax_rate
    from businesses b, street_work_table swt
    where swt.streetname=b.streetname
    and b.streetnum between swt.low_streetnum and swt.high_streetnum
    and swt.process_id = @proc_id
    and b.demographics & @target_demographics > 0
    delete street_work_table
    where process_id=@proc_id
    return 0
end

By replicating the worktable (street_work_table) and the inner procedure (set_tax_rate_sub) instead of the outer procedure, the difficult logic to identify the streets between the others is not performed at the replicate, allowing the use of select/into at the primary database for this logic, while reducing the number of rows actually replicated to the replicate system. Note the following considerations:

- Inner procedure performs cleanup on the worktable. This reduces the number of rows replicated as only the inserts into the worktable get replicated from the primary.
• @@spid is parameter to the inner procedure and column in the worktable. The reason for this is that in multi-user situations, you may need to identify which rows in the worktable are for which user’s transactions. Since the spid at the replicate will be the spid of the maintenance user and not the same as at the primary, it must be passed to the subprocedure so that the maintenance user knows which rows to use.

• The inner procedure call and inserts into the worktable are enclosed in a transaction at the primary. This is due to the simple fact that if the procedure hits an error and aborts, the procedure execution was successful according to the primary ASE. As a result it would still be replicated and attempted at the replicate. By enclosing the inserts and proc call in a transaction, the whole unit could be rolled back at the primary, resulting in a mini-abort in the RS that would purge the rows from the inbound queue.

The last point is fairly important. Any procedure that is replicated should be enclosed in a transaction at the primary. This will allow user-defined exits (raiserror, return –1) to be handled correctly provided that the error handling does a rollback of the transaction. Despite the fact an error is raised and a negative return status returned from the procedure, it still is a successful procedure execution according to ASE, consequently replicated to all subscribing databases where the same raiserror would occur resulting in a suspended DSI.

A crucial performance suggestion for the above is to have the clustered index on the worktable have the spid and one or more of the other main columns as indexed columns. For example, in the above example, the clustered index might include spid, and streetname. Then if the real data table (businesses) has an index on streetname, the update via join can use the index even if no other SARG (true in the above case) is possible.

While this technique may appear to have limited applicability, in actuality, it probably resolves most of the cases in which a select/into is used at the primary database and not all the rows are modified in the target table (establishing the fact some criteria must exist – replicate the criteria vs. the rows). Situations it is notably applicable for include:

Area Bounded Criteria – DML involving area boundaries identified via zip codes, area code + phone exchange, countries, regions, etc. A classic example is the “mark all blood collections from regions with E-Boli outbreak as potentially hazardous” example often used in replication design examples as good procedure replication candidates. The list of blood donations would be huge, but the list of collection centers located in those regions is probably very small.

Specified List Criteria – In certain situations, rather than using a range, a specified list is necessary to prevent unnecessarily updating data inclusive in the range at the replicate (a consolidated system) but not in the primary. For example, a list of personnel names being replicated from a field office to the headquarters. This could include dates, account numbers, top 10 lists, manufacturers, stores, etc.

As well as any other situation in which a fairly small list of criteria exists compared to the rows actually modified.

Procedure Rewrite without Select/Into

This, unfortunately, is the most frequent fallback for developers suddenly faced with the select/into at replicate problem – and agreeably, sometimes it is necessary. However, this usually requires permanent working tables in which the procedure makes logged inserts/updates/deletes. This should only be used when the identifying criteria is the entire set of rows or a range criteria that is huge in itself. An example is if a procedure is given a range of N-Z as parameters. While it is possible to create a list of 13 characters and attempt the above, the end result is the same – thousands of rows will be changed. A classic case would be calculating the finance charges for a credit card system. In such a situation – even if the “load” was distributed across every day of the month by using different “closing dates” – tens of thousands to millions of rows would be updated each execution of the procedure. Since most credit cards operate on an average daily balance to calculate the finance charges, the first step would be to get the previous month’s balance (hopefully stored in the account table), subtract any payments (as these always apply to “old” balances first). This is a bit more difficult than simply taking the average and dividing by the number of days. Consider the following table:

<table>
<thead>
<tr>
<th>Day</th>
<th>Charge</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Begin</td>
<td>1,000.00</td>
<td>1,000.00</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>1,000.00</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1,000.00</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>1,000.00</td>
</tr>
<tr>
<td>4</td>
<td>50.00</td>
<td>1,050.00</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>1,050.00</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>1,050.00</td>
</tr>
</tbody>
</table>
As you can see, there is no way to simply take the sum of the new charges ($900) and get the final answer. As a result, the system needs to first calculate the daily balance for each account and then insert the average daily balance multiplied by some exorbitant interest rate (i.e. 21% for department store cards) for the finance charge. For sake of argument, let’s assume this is done via a series of select/into’s (possible with about 3-4 – an exercise left for the reader). Obviously, no matter what time the procedure runs, it will run for several hours on a very large row count. Replicating the procedure is a must as replicating all the row changes at the end of every day (assuming every day is a “closing date” for 1/30th of the accounts), could be impractical. Consequently, instead of using select/into’s to generate the average daily balances, a series of real worktables would have to be used.

### Separate Execution Connection

This last example (finance charges on average daily balance) clearly illustrates a problem though in replicating stored procedures. At the primary system – assuming no contention at the primary – the finance charges procedure could happily run at the same time as user transactions (assuming the finance charge procedure used a cursor to avoid locking the entire table). However, as described before, in order to guarantee that the transactions are delivered in commit order, the Replication Server applies the transactions serially. Consequently, once the procedure started running at the replicate, it would several hours before any other transactions could begin. Additionally, at the replicate, the entire update would be within a transaction – if it didn’t fail due to exhausting the locks, the net result would be a slow lockdown of the table. This, of course, is extremely unsatisfactory.

One way around this is to employ a separate connection strictly for executing this and other business maintenance. In doing so, normal replicated transactions could continue to be applied while the maintenance procedure executed on it’s own. The method to achieve this is based on multiple (not parallel – multiple) DSI’s which is covered later in this section. Needless to say, there are many, many considerations to implementing this which are covered later, consequently, this should only be used when other methods have failed and procedure replication is really necessary. One of those considerations is the impact on subsequent transactions that used/modified data modified by the maintenance procedure. Due to timing issues with a separate execution connection, it is fully possible that the update makes it to the replicate first – only to be clobbered by later execution within the maintenance record.
One of the other advantages to this approach, is that statement and transaction batching could both be turned off. This would allow the procedure at the replication to contain the select/into provide that system administrators were willing for a manual recovery (similar to system transactions). With both statement and transaction batching off, the following procedure would work.

```
create procedure proc_w_select @parm1 int
as begin
    declare @numtrans int
    select @numtrans=@@trancount
    while @@trancount > 0 commit tran

    -- select into logic
    begin tran
    -- updates to table
    commit tran
    while @@trancount < @numtrans begin tran
    return 0
end
```

This is similar to the mechanism used for system transactions such as DDL or truncate table. In the case of system transactions, Replication Server submits the following:

```
rs_begin
rs_commit
-- DDL operation
rs_begin
rs_commit
```

The way this works is that the rs_commit statements update the OQID in the target database. During recovery, only three conditions could exist:

- **rs_lastcommit OQID < first rs_commit OQID** – In this case, recovery is fairly simple as the empty transaction prior to the DDL has not yet been applied. Consequently, the RS can simply begin with the transaction prior to the DDL.

- **rs_lastcommit OQID >= second rs_commit OQID** – Similar to the above, recovery is simple as this implies that the DDL was successful since the empty transaction that followed it was successful. As a result, Rep Server can begin with the transaction following the one for which the OQID was recorded.

- **rs_lastcommit OQID = first rs_commit OQID** – Here all bets are off. Reason is that one of two possible situations exists. Either 1) the empty transaction succeeded but the DDL was not applied (replicate ASE crashed in middle); or 2) both were applied. Since the DDL operation is not within an rs_commit, the OQID is not updated when it finishes. Consequently the administrator has to check the replicate database and make a conscious decision whether or not to apply the system transaction. Hence the added “execute transaction” option to resume connection command. By specifying execute transaction, the administrator is telling RS to re-apply the system transaction as it never really was applied. If instead it had run but the second rs_commit had not, then simply leaving it off the resume connection is sufficient.

Accordingly, by committing and re-beginning the transactions at the procedure boundaries, you are not sure if the proc finished if the OQID is equal to the OQID prior to the proc execution. If it was successful, resume connection DS.DB skip transaction provides similar functionality to leaving of “execute transaction” for system transactions.

However, it is critical that the procedure be fully recoverable – possibly even to a point where it could recover from a previous incomplete run. If the actual data modifications were made outside a transaction, then when a failure occurs during the execution, reapplied the procedure after recovery would result in duplicate data. So, for example, the finance charge procedure would only develop the list of average monthly balances from accounts that did not already have a finance charge for that month.

**Concurrency Issues**

In replicate only databases, concurrency is mainly an issue between the parallel DSI threads or when long running procedures execute and lock entire tables. However, in shared primary configurations – workflow systems or other systems in which the data in the replicate is updated frequently, concurrency could become a major issue. In this case,
user transactions and Rep Server maintenance user transactions could block/deadlock each other. This may require decreasing the dsi_max_xacts_in_group parameter to reduce the lock holding times at the replicate as well as ensuring that long running procedures replicated to that replicate database are designed for concurrent environments.

**Key Concept #19:** In addition to concurrency issues between maintenance user transactions when using Parallel DSIs, if the replicate database is also updated by normal users, considerable contention between maintenance user and application users may exist. Reducing transaction group sizes as well as designing long running procedures to not cause contention are crucial tasks to ensuring the content does not degrade business performance at the replicate or Replication Server throughput.

Similar to any concurrency issue, depending on what resources are the source of contention, it may be necessary to use different locking schemes, etc. at the replicate than at the primary (or same if Warm Standby). Consider the following activities:

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Additional Indexes</td>
<td>Additional indexes, particularly if replicating to a denormalized schema or data warehouse could increase contention. While not necessarily avoidable, it may require a careful “pruning” of OLTP specific indexes.</td>
</tr>
<tr>
<td>DOL Locking</td>
<td>Eliminate index contention and data row contention by implementing DOL locking at the replicate system.</td>
</tr>
<tr>
<td>Table Partitioning</td>
<td>Provide parallel DSI’s multiple last pages to avoid contention without implementing DOL locking.</td>
</tr>
<tr>
<td>Triggers Off</td>
<td>Have RS DSI disable triggers – especially data validation triggers</td>
</tr>
</tbody>
</table>

Obviously, the above list is not complete, but may provide ideas to resolve contention issues when the contention is not due to the holding of locks longer due to transaction grouping.
Parallel DSI Performance

*I turned on Parallel DSI’s and only got a 30% improvement - what happened?*

The answer is that if using the default settings, not a whole lot of parallelism is experienced. In order to understand parallel DSI’s, a solid foundation in Replication Server internal processing is necessary. This goes beyond just understanding the functions of the internal threads – it also means understanding how the various tuning parameters as well as types of transactions affect replication behavior, particularly the DSI. In the following sections, we will discuss the need for parallel DSI, internal threads, tuning parameters, serialization methods, special transaction processing and considerations for replicate database tuning.

Need for Parallel DSI

There are five main bottlenecks in the Replication Server:

1. Replication Agent transaction scan/delivery rate
2. Inbound SQT transaction sorting
3. Distributor thread subscription resolution
4. DSI transaction delivery rate
5. Stable Queue/Device I/O rate

In early 10.x versions of Replication Server, it was noticed that the largest bottleneck in high volume systems was #4 – DSI transaction delivery rate. The reason was very simple. At the primary database, performance was achieved by concurrent processes running on multiple engines using a task efficient threading model. On the other hand, at the replicate database, Replication Server was limited to a single process. Consequently, if the aggregate processing at the primary exceeded the processing capability of a single process, the latency would increase dramatically. Much of this time was actually not spent on processing as most replication systems were typically handling simple insert/update/delete statements, but rather the “sleep” time waiting for the I/O to complete. Consider the following diagram.

![Diagram showing the need for parallel DSI]

It should be noted that in the above figure, the numbers are fictitious. However, it does illustrate the point how a single threaded delivery process can quickly become saturated. Early responses to this issue “talked” around it by attributing this to Replication Server’s ability to “flatten” out peak processing to a more “manageable” steady-state transaction rate. While this may be appealing to some, organizations with 24x7 processing requirements or those with OLTP during the day and batch loading at night quickly realized that this “flattening” required a full time of little or no activity during which replication would catch up. Due to normal information flow, the organizations did not have this time to provide.
The obvious solution was to somehow introduce concurrency into the replication delivery. The challenge was to do so without breaking the guarantee of transactional consistency. The result was that in version 11.0, Parallel DSI’s were introduced to improve the replication system delivery rates.

**Key Concept #20 – Replication/DSI throughput is directly proportionate to the degree of concurrency within the parallel DSI threads.**

**Parallel DSI Internals**

Earlier in one of the first sections of this paper, we discussed the internal processing of the Replication Server. From this aspect, very little is different for Parallel DSI’s, however, considerable skill and knowledge is necessary to understand how these little differences are best used to bring about peak throughput from Replication Server. While this section discusses the internals and configuration/tuning parameters, later sections will focus on the serialization methods as they are key to throughput, as well as tuning Parallel DSI’s.

**Parallel DSI Threads**

The earlier diagram discussing basic Replication Server internal processing included in the illustration Parallel DSI’s (step 11 in the below)

![Diagram](image)

**Figure 29 – Replication Server Internals with Parallel DSI’s**

While the DSI thread is still responsible for transaction grouping, etc., it is the responsibility of the DSI Executor threads to perform the function string translation, apply the transactions and perform error recovery. Up to 20 Parallel DSI threads can be configured per connection. However, after a certain number of threads, adding more will not increase throughput.

**rs_threads processing**

As mentioned earlier (and repeatedly), the Replication Server guarantees transactions are applied in the same order at the replicate as at the primary. At first glance, this would seem an impossible task where Parallel DSI’s are employed – a long running procedure on DSI 1 ..and DSI 2 might get ahead. To prevent this, Replication Server implements a synchronization point at the end of every transaction by way of the rs_threads table.

```sql
create table rs_threads
(
    id int, -- thread id
    seq int, -- one up used for detecting rollbacks
    pad1 char(255), -- padding for rowsize.
)
```
This table is manipulated using the following functions used only when Parallel DSI is implemented.

<table>
<thead>
<tr>
<th>Function</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs_initialize_threads</td>
<td>Used during initial connection to setup rs_threads table. Issued shortly after rs_usedb in the sequence.</td>
</tr>
<tr>
<td>rs_update_threads</td>
<td>Used by a thread to block its row in the rs_threads table to ensure commit order and also to set the sequence number for rollback detection.</td>
</tr>
<tr>
<td>rs_get_thread_seq</td>
<td>Used by a thread to determine when to commit by selecting the previous thread’s row in rs_threads.</td>
</tr>
<tr>
<td>rs_get_thread_seq_noholdlock</td>
<td>Similar to above, but only used when isolation_level_3 is the serialization method.</td>
</tr>
</tbody>
</table>

To understand how this works, consider an example in which 5 Parallel DSI threads are used. During the initial connection processing during recovery, Replication Server will first issue the rs_initialize_threads function immediately after the rs_usedb. This procedure simply performs a delete of all rows (logged delete vs. truncate table due to heterogeneous support), and then inserts blank rows for each DSI initializing seq value to 0.

During processing, when Parallel DSI’s are in use, the first statement a DSI issues immediately following the begin transaction for the group is similar to the following:

```sql
create procedure rs_update_threads
    @rs_id int,
    @rs_seq int
as
    update rs_threads set seq = @rs_seq where id = @rs_id
go
```

Each DSI simply calls the procedure with its thread id (i.e. 1-5 in our example) and the seq value plus one from the last transaction group (the initial call uses a value of 0). Since this update is within the transaction group, it has the effect of blocking the thread’s row during the transaction group’s duration. Following this, normal transaction statements within the transaction group are sent as normal.

After all the transaction statements have been executed, the DSI then attempts to select the previous thread’s row from the rs_threads table using the rs_get_thread_seq function. If the previous thread has not yet committed, then the thread is blocked (due to lock contention) by the update lock on the row by the previous thread. If the previous thread has committed, then the lock is not held, consequently, the current thread possibly also can commit. Ignoring the effects of serialization method on transaction timing, this could be illustrated as in the below diagram. Note that in each case, each subsequent thread is blocked and waiting on the previous thread’s update on rs_threads.
To anyone who has monitored their system and checked object contention, they probably thought all of the blocking on rs_threads was a problem. As you can tell from the above, it is actually deliberate. The theory of the above is that transactions can acquire locks and execute in parallel – but due to the rs_threads locking mechanism, the transactions are still committed in order (1-20 in the above). After all threads have transactions pending, the DSI Scheduler simply waits until the last one commits prior to starting over with thread #1. This behavior could result in a “bursty” round-robin throughput if it were not for the serialization method controlling the transaction delivery timing.

The first question that comes to mind for many is: “What happens if one of the threads hits an error and rollsback its transaction? Wouldn’t the next thread simply commit?” The answer is no. This is where the seq column comes in and the realization why rs_get_thread_seq has seq in the name. As each rs_get_thread_seq function call is made, it returns the seq column for the previous thread. This value is simply compared to the previous value. If it is equal to the previous value, then an error must have occurred and subsequent transactions need to rollback as well. However, if the seq value is higher than the previous seq value for that thread, then the current thread can commit its transaction.
Thread Sequencing

As mentioned, the parallel transactions are submitted to each of the threads in order. Each one blocks until the previous ones commit. The process does not start over with thread #1 until the last thread has committed. This is somewhat deliberate. If thread #1 received a sequence of quick atomic inserts while the others received long running procedures, thread #1 could theoretically receive its second batch prior to thread #2 even getting to the point where it would select the seq. The result would be that it would be waiting for the last thread while the second would be waiting for the first – a typical deadlock in which all would have to be rolled back. However, the result of this is the following behavior:

- “Bursty” processing in which a slight gap of no processing exists for each thread until the last thread has committed.
- Low degree of parallelism prior to no further return in performance gain.

The first point is best understood by looking at the illustration below (again, ignore thread timing as that will be discussed in the section on serialization).

As you can see, each thread has a gap of no processing between each batch. This gap is arbitrary and is determined by the serialization method and the length of time for each transaction to commit. This behavior can mistakenly lead system administrators to believe that the thread is not processing anything at all as each time they check, the individual...
thread is in “receive sleep” mode. To really verify if the thread is performing work, simply do a dirty read on rs_threads for that particular thread id (i.e. select * from rs_threads at isolation read uncommitted). That is not to say that some threads may not be performing work – large transaction threads in particular are only used for large transactions (discussed later).

The second point in the discussion was that after a low degree of parallelism, the point of no return is reached in which no further DSI’s will achieve performance improvement. This point is extremely dependent on the serialization method, as will be seen later, along with DSI transaction group processing speed. For example, consider the following scenario in which each thread does not receive the next batch until nearly the end of the first.

![Figure 33 – Example of 10 Parallel DSI’s with delayed transactions](image)

In the above, the point of no return is probably in the 3-5 DSI range – after which, the lack of activity on the other threads result in extremely limited true parallelism. On the other hand, if the transaction timing is similar to the illustration on the previous page, more DSI’s could achieve greater performance if contention is minimized.

**Configuration Parameters**

There are several configuration parameters that control Parallel DSI’s.

<table>
<thead>
<tr>
<th>Parameter (Default)</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsi_large_xact_size (100)</td>
<td>The number of commands allowed in a transaction before the transaction is considered to be large for using a single parallel DSI thread. The minimum value is 4. The default is probably far too low for other than strictly OLTP systems. Mixed load systems should consider raising this.</td>
</tr>
<tr>
<td>dsi_num_large_xact_threads (0)</td>
<td>The number of parallel DSI threads to be reserved for use with large transactions. The maximum value is one less than the value of dsi_num_threads. More than 2 are probably not effective. See section on parallel DSI.</td>
</tr>
<tr>
<td>dsi_num_threads (1)</td>
<td>The number of parallel DSI threads to be used. The maximum value is 20. See section on parallel DSI.</td>
</tr>
<tr>
<td>Parameter (Default)</td>
<td>Explanation</td>
</tr>
<tr>
<td>------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>dsi_serialization_method</td>
<td>The method used to maintain serial consistency between parallel DSI threads, when applying transactions to a replicate data server. Values are:</td>
</tr>
<tr>
<td>(default: wait_for_commit)</td>
<td>* isolation_level_3* - specifies that transaction isolation level 3 locking is to be used in the replicate data server.</td>
</tr>
<tr>
<td></td>
<td>* single_transaction_per_origin* - prevents conflicts by allowing only one active transaction from a primary data server.</td>
</tr>
<tr>
<td></td>
<td>* wait_for_commit* - maintains transaction serialization by instructing the DSI to wait until one transaction is ready to commit before initiating the next transaction.</td>
</tr>
<tr>
<td></td>
<td>* none* - assumes that your application is designed to avoid conflicting updates, or that lock protection is built into your database system.</td>
</tr>
<tr>
<td></td>
<td>See section on parallel DSI</td>
</tr>
<tr>
<td>dsi_sqt_max_cache_size</td>
<td>Maximum SQT (Stable Queue Transaction interface) cache memory for the database connection, in bytes.</td>
</tr>
<tr>
<td>(default: 0)</td>
<td>The default, &quot;0,&quot; means that the current setting of sqt_max_cache_size is used as the maximum cache size for the connection.</td>
</tr>
<tr>
<td></td>
<td>This parameter controls the use of parallel DSI threads for applying transactions to a replicate data server. The more DSI threads you plan on using, the more dsi_sqt_max_cache_size you may need.</td>
</tr>
<tr>
<td>parallel_dsi</td>
<td>Provides a shorthand method for configuring parallel DSI threads. A setting of &quot;on&quot; configures these values:</td>
</tr>
<tr>
<td>(default: off)</td>
<td>dsi_num_threads = 5</td>
</tr>
<tr>
<td></td>
<td>dsi_num_large_xact_threads = 2</td>
</tr>
<tr>
<td></td>
<td>dsi_serialization_method = &quot;wait_for_commit&quot;</td>
</tr>
<tr>
<td></td>
<td>dsi_sqt_max_cache_size = 1 million bytes.</td>
</tr>
<tr>
<td></td>
<td>A setting of &quot;off&quot; configures these parallel DSI values to their defaults. You can set this parameter to &quot;on&quot; and then set individual parallel DSI configuration parameters to fine-tune your configuration.</td>
</tr>
</tbody>
</table>

As illustrated by the single parameter “parallel_dsi”, many of these work together. Note that parallel_dsi sets several configuration values to what would appear to be fairly low numbers. However, due to the serialization method, these settings are typically the most optimal. More DSI threads will not necessarily improve performance.

**Serialization Methods**

**Key Concept #21 - Serialization Method has nothing to do with transaction order.** No matter which serialization method, transactions at the replicate are always applied in commit order. However, it does control the timing of transaction delivery with Parallel DSI’s in order to reduce contention caused by conflicts between the DSI’s.

One of the most difficult concepts to understand is the difference between the serialization methods. The best way to describe this is that the serialization method you choose depends on the amount of contention that you expect between the parallel threads. Some of this you can directly control via the dsi_max_xacts_in_group tuning parameter. The more transactions grouped together, the higher the probability of contention as the degree of parallelism increases or
the higher the probability of contention with other users on the system. This will become more apparent as each of the serialization methods will be described in more detail in the following sections.

**wait_for_commit**

The default setting for dsi_serialization_method is “wait_for_commit”. This serialization method assumes that there will be considerable contention between the parallel transactions. As a result, the next thread’s transaction group is not sent until the previous thread’s statements have all completed successfully and the commit is being sent. This results in the thread timing that we saw earlier in which execution was more staggered than parallel as illustrated below.

![Thread timing with dsi_serialization_method = wait_for_commit](image)

As discussed earlier, this timing sequence would have limited scalability beyond 3-5 parallel DSI threads. However, it assures that contention between the threads does not result in one rolling back – which would cause all those that follow to rollback as well.

**none**

“None” does not infer that no serialization is used. What it really means is that no (none) contention is expected between the Parallel DSI threads. As a result, the thread transactions are submitted nearly in parallel (timing then more a factor of the number of transactions in the closed queue in the SQT cache). This looks like the earlier illustration similar to the below.

![Thread timing with dsi_serialization_method = none](image)

**Parallel DSI Contention**

This method provides the greatest scalability – the more DSI’s involved, the higher the throughput. However, it also means a higher probability of contention causing rollback of a significant number of transactions (remember, if one rolls back, the rest do as well). Remember – the threads are already blocked on each other’s rows in rs_threads – deliberately – to ensure commit order is maintained. Any contention between threads, then, is more than likely going to cause a deadlock. Consider the following illustration.
In the example above, two deadlocks exist – threads 1 & 2 are deadlocked since thread 2 is waiting on thread 1 to commit as normal (rs_threads) yet thread 2 started processing its update on table B prior to thread 1 (assuming the same row hence the contention). As a result, #2 is waiting on #1 and #1 is waiting on #2 – a classic deadlock. Threads 3 & 4 are similarly deadlocked. Interestingly enough, one of the more frequent tables “blamed” for deadlocks in replicated environments is the rs_threads table. As you can see – this is rather deliberate. Consequently deadlocks involving rs_threads should not be viewed as contention issues with rs_threads, but rather an indication of contention between the transactions the DSI’s were applying.

The biggest problem with this is that once one thread rollsback (typical response for a deadlock), all the subsequent threads will rollback as well. In order to prevent the contention from continuing and causing the same problems all over again, the Replication Server will retry the remaining transactions serially (one batch at a time) before resuming parallel operations. Obviously, a rollback followed by a serial transaction delivery will cause performance degradation if it happens frequently enough. However, a small number of occurrences are probably not a problem. During a benchmark at a customer site, using the default wait_for_commit resulted in the inbound queue rapidly getting one hour behind the primary bcp transaction. Switching to “none” drained the queue in 30 minutes as well as keeping up with new records. During these 30 minutes, the Replication Server encountered 3 rollbacks per minute – ordinarily excessive, but in this case, the serialization method of none was outperforming the default choice. However, at another customer site, a parallel transaction failed every 3-4 seconds – and no performance gain was noted in using "none" over “wait_for_commit”. As usual, this illustrates the point that no one-size-fits-all approach to performance tuning works and that each situation brings its own unique problem set.

Avoiding Contention

While the book states that “This method assumes that your application is designed to avoid conflicting updates, or that lock protection is built into your database system.” it is not as difficult to achieve as you think. Basically, if you do not have a lot of contention at the primary, then contention at the replicate may be a direct cause of system tuning settings at the replicated DBMS and not due to the transactions. If the contention is system induced, you need to first determine the type of contention involved and whether it involves. Consider the following matrix of contention and possible resolutions.
### Contention

<table>
<thead>
<tr>
<th>Contention</th>
<th>Possible Resolution(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Last page contention</td>
<td>Change clustered index, partition table or use datarow locking.</td>
</tr>
<tr>
<td>Index contention</td>
<td>Use datapage locking or reduce dsi_max_xacts_in_group</td>
</tr>
<tr>
<td>Row contention</td>
<td>Reduce dsi_max_xacts_in_group until contention reduced</td>
</tr>
</tbody>
</table>

Note that nowhere in the above did we suggest changing the serialization method to “wait_for_commit”. If the problem is system induced as compared to the primary – yes, wait_for_commit will resolve it – however, the impact on throughput can be severe. In almost any system, a serialization method of “none” should be the goal. Backing off from that goal too quickly when other options exist could have a large impact on the ability of Replication Server to achieve the desired throughput. Keep in mind that even 2 threads running completely in parallel with a serialization of “none” may be better than 5 or 6 using “wait_for_commit”.

### isolation_level_3

Serialization method “isolation_level_3” is identical to “none” with the addition that Replication Server first issues “set isolation level 3” via rs_set_isolation_level_3 function. However, as one would expect, this could increase contention between threads dramatically due to select locks being held throughout the transaction. While Replication Server is normally associated with write activity, a considerable amount of read activity could occur in the following:

- Declarative integrity (DRI always holds locks)
- Select statements inside replicated procedures
- Trigger code if not turned off for connection
- Custom function strings
- Aggregate calculations, etc

Consequently, isolation_level_3 should ONLY be used when absolutely necessary to ensure repeatable reads from the aspect of the replicated transactions. Consider for example the normal “phantom read” problem where a process scanning the table reads a row – the row is moved as a result of an update, and then the row is re-read. In a normal system, this is simply avoided by having the scanning process invoke isolation level 3 via the set command. However, if you think about it, no one ever mentions having the offending writer invoke isolation level 3. The reason for that is that it would be unnecessary as once the read scans the row to be updated, it holds the lock until the read completes – thereby blocking the writer and preventing the problem. In this case, most of Replication Server’s transactions will be as the writer, so, it probably is in the same role as the offending writer in the phantom read – no isolation level three required.

As a result, isolation level three should only be used when the replicated transaction needs to perform a repeatable read. Of course, the most obvious example of this is when performing aggregation for data elements that are not aggregated at the primary. This could be a scenario similar to replicating to a DSS system or a denormalized design in which only aggregate rollups are maintained. Even in these cases however, isolation level 3 may not be necessary as alternatives exist. Consider the classic case of the aggregate. Let’s assume that a bank does not keep the “account balance” stored in the primary system (possibly because the primary is a local branch and may not have total account visibility??). When replicating to the central corporate system, the balance is needed to ensure timely ATM and debit card transactions. Of course, this could be implemented as a repeatable read triggered by the replicated insert, update, delete or whichever DML operation. However, it is totally unnecessary. Because Replication Server has access to the complete before and after images of the row, a function string similar to the following could be constructed:

```sql
alter function string <redef_name>.rs_update
for rs_default_function_class
output language
'update bank_account
where <pkeycol> = ?tran_id!new?'
```

This maintains the aggregate without isolation level three required – and much more importantly – without the expensive scan of the table to derive the delta. By exploiting function strings – or by encapsulating the set isolation command within procedure or trigger logic, you may find that you can either void using isolation level three or restrict it only to those transactions from the primary that truly need it.
In summary, in addition to the contention increase simply from holding the locks on select statements, a possibly bigger performance issue when isolation level three is required is the extra i/o costs of performing the scans that the repeatable reads focus on – all within the scope of the DSI transaction group. As a result, isolation level 3 should be avoided unless absolutely necessary – ANSI compliance is not a good enough reason.

**single_transaction_per_origin**

The single_transaction_per_origin serialization method is mainly used for corporate rollup scenarios. Although clearly applicable for corporate rollups, another implementation for which single_transaction_per_origin works well is the shared primary or any other model in which the target replicated database is receiving data from multiple sources.

In this serialization method, since the transactions are from different origin databases, there should not be any contention between transactions. For example, stock trades in Chicago, San Francisco, Toronto, Tokyo and London are completely independent of each other – consequently their DML operations would not interfere with each other except in cases of updates of aggregate balances. However, within each site – for example, transactions from Chicago – some significant amount of contention may exist. By only allowing a single transaction per origin, each DSI could simply be processing a different sites transaction – consequently, the transaction timing is similar to none or isolation_level_3 in that the Parallel DSI threads are not staggered waiting for a the previous commit. From an internal threads perspective, it would resemble:
From a performance perspective, single_transaction_per_origin may not have as high of a throughput as other methods such as none. Consider the following:

*Origin Transaction Balance* – single_transaction_per_origin works best in situations where all the sites are applying transactions evenly. In global situations where normal workday at one location is offset from the other sites, this is not true. Instead, all of the transactions for a period of time come from the same origin – and consequently are single threaded.

*Single Origin Error* – Consider what happens if one of the replicated transactions from one of the sites fails for any reason. All DSI threads are suspended and the queue fills until the one site’s transaction is fixed and connection resumed. This could cause the outbound and inbound queues to rapidly fill – possibly ending up with a primary transaction log suspend.

*Origin Transaction Rate* – Again, each individual site effectively has a single DSI of all the parallel DSI’s to use. If the source system has a very high transaction volume, the outbound queue will get behind quickly.

Either one of these situations is fairly common and could cause apparent performance throughput to appear much lower than normal. While the error handling is easily spotted from the Replication Server error log, the source transaction rate or the balance of transactions is extremely difficult to determine on the fly.

**Large Transaction Processing**

One of the most commonly known and frequently hit problems with Replication Server is processing large transactions. In earlier sections, the impact of large transactions on SQT cache and DIST/SRE processing were discussed. This section takes a close look at how large transactions affect the DSI thread. It should be noted that it is at the DSI that a transaction is defined as “large”. While a transaction may be “large” enough to be flushed from the SQT cache – it still can be too small to qualify as a large transaction.

**Parallel DSI Tuning**

**DSI Tuning Parameters**

There really only are two tuning parameters for large transactions. Both of these are only applicable to Parallel DSI implementations. The tuning parameters are:
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>dsi_large_xact_size</td>
<td>100</td>
<td>The number of commands allowed in a transaction before the transaction is considered to be large for using a single parallel DSI thread. It also controls behavior such as early conflict detection in Parallel DSI threads and signals whether transaction can be applied prior to commit record being seen. The minimum value is 4. However, 100 may be too small for most production systems as well.</td>
</tr>
<tr>
<td>dsi_num_large_xact_threads</td>
<td>0</td>
<td>The number of parallel DSI threads to be reserved for use with large transactions. The maximum value is one less than the value of dsi_num_threads. This value directly impacts the number of threads available for normal transaction processing (num_threads – dsi_num_large_xact_threads). In reality, due to early conflict detection behavior, more than 2 threads is probably suspect as normal rs_threads processing will prevent more than 2 from being effective.</td>
</tr>
</tbody>
</table>

The key tuning parameter of both of these is dsi_large_xact_size. When a transaction exceeds this limit, the DSI processes it as a large transaction. In doing so, the DSI does the following:

3. Allow the transaction to be sent to the replicate without waiting for the commit record to be read.
4. Use a dedicated large transaction DSI
5. Each dsi_large_xact_size rows, the DSI will attempt to provide early detection

An important note is that this is only applicable to Parallel DSI. If Parallel DSI is not used, large transactions are processed normally with no special handling.

**SQT Open Queue Processing**

The reference manual states that large transactions begin to be applied by the DSI thread before the DSI sees the commit record. This is accomplished by the DSI processing large transactions from the “Open” queue vs. the more normal “Closed” queue in the SQT cache. This does have an immediate negative effect in Warm Standby systems that a large transaction may be rolled back at the primary – and need to be rolled back at the replicate. How can this happen?? Simple. Consider the case of a fairly normal bcp of 1,000 rows into a replicated table (slow bcp so row changes are logged). As the row changes are logged, they are forwarded to the Replication Server by the Rep Agent long before the commit is even submitted to the primary system. If at the default, after 100 rows have been processed to the Replication Server, the transaction would be labeled as a large transaction. As a result, the DSI would start applying the transaction’s row changes immediately without waiting for a commit (in fact, the commit may not even have been submitted to the primary yet). Now, should the bcp fail due to row formatting problems – it will need rolled back - not only in the primary, but also at the replicate as the transaction has already been started.

With such a negative, why is this done?? The answer is simple – transaction rollbacks in production systems are extremely rare (or should be!!) – therefore this issue is much more of an exception and not the norm. In fact, for normal (non-Warm Standby) replication, the commit had to have been issued at the primary and processed in the inbound queue or it would not have even got to the outbound queue. In addition, the benefit of this approach far outweighs the very small amount of risk. Consider the latency impact of waiting until the commit is read in the outbound queue as illustrated below by the following timeline:
Without starting to apply the transaction until the commit is read, several problems can occur. First, as illustrated above, the overall latency of the transaction is extended. In the bottom DSI execution of the transaction (labeled DSI -> RDS (large xactn)), it finishes well before it would if it waited until the transaction was moved to the SQT Closed queue. This is definitely an important benefit for batch processing to ensure that the batch processing finishes at the replicate prior to the next business day beginning. Consider the above example. If each time unit equaled an hour (although 2 hours for DIST/SRE processing is rather ludicrous) at the transaction began at the primary at 7:00pm, it would finish at the replicate at 7:00am the next morning using large transaction thread processing. Without it, the transaction would not finish at the replicate until 10:00am – 2 hours into business processing.

The latency savings for this is really evident in Warm Standby. Remember, for Warm Standby, the Standby DSI is reading from the inbound queue’s SQT cache. Normal (small) transactions, of course, are not sent to the Standby database until they have committed. However, since a large transaction reads from the SQT “Open” queue, it is fully possible that the Standby system will start applying the transaction within seconds of it starting at the primary and would commit within nearly the same time. Compare the following timeline with the one above.

However, the above will only happen if large transactions run in isolation. The problem is that if a large transaction begins to be applied and another smaller transaction commits prior to the large transaction, the large transaction is rolled back and the smaller concurrent transaction committed in order. After the smaller transaction commits, the large transaction does not restart from the beginning automatically - but rather waits until the commit is actually received before it is reapplied. This probably is due to the expense of large rollback’s and the aspect that if it the rollback occurs
once, it is likely to occur again. This behavior is easily evident by performing the following in a Warm Standby configuration:

1. Configure the DSI connections for parallel DSI using the default parallel_dsi='on' setting.
2. Begin a large transaction at the primary (i.e. a 500 row insert into table within an explicit transaction). At the end of the transaction place a `waitfor delay '00:03:00'` immediately prior to the commit.
3. Use a dirty read at the replicate to confirm large transaction is started.
4. Perform an atomic insert into another table at the primary (allow to implicitly commit)
5. Use a dirty read at the replicate to confirm large transaction rolled back and does not restart until delay expires and transaction commits.

As a result, attempts to tune for and allocate large transaction threads will be negated if smaller/other transactions are allowed run concurrently and commit prior to the large transaction(s). This behavior, coupled with the “early conflict detection” and other logic implemented in large transaction threads to avoid excessive rollbacks is a very good reason to avoid the temptation - especially in Warm Standby - to reduce dsi_large_xact_size with hopes of improving throughput and reducing latency.

**Key Concept #22:** Large transaction DSI handling is intended to reduce the double "latency penalty" that waiting for a commit record in the outbound queue introduces in normal replication and latency as well as switch active timing issues associated with Warm Standby. However, it is nearly only useful when large transactions run in isolation (such as serial batch jobs).

Having said that, large transactions run concurrently (provided started in order of commit) such as concurrent purge routines may be able to execute without the rollback/wait for commit behavior. However, concurrent large transactions may not experience the desired behavior as will be discussed in the next section.

**Parallel DSI Processing**

In addition to beginning to process large transactions before the commit record is seen by the DSI/SQT, if using Parallel DSI’s, the Replication Server also processes the large transaction slightly differently during execution. Two of these differences are in the early conflict detection and in utilization of reserved DSI threads set aside for large transactions.

**Early Conflict Detection**

Another factor of large transactions that the dsi_large_xact_size parameter controls is the timing of early conflict detection. This is stated in the Replication Server Administration manual as “After a certain number of rows (specified by the dsi_large_xact_size parameter), the user thread attempts to select the row for the next thread to commit in order to surface conflicting updates.” What this really means is the following. During processing of large transactions, every dsi_large_xact_size rows, the DSI thread attempts to select the sequence number of the thread before it. So, for example, for a large transaction of 1,000 statements (i.e. a bcp of 1,000 rows), the Replication Server would insert an `rs_get_threadseq` every 100 rows (assuming dsi_large_xact_size is still the default of 100). By doing this, if there is a situation in which the large transaction is blocking the smaller one, a deadlock is caused, thus “surfacing” the conflict. This is illustrated in the diagram below, in which thread #2 is being blocked by a conflicting insert by thread #3.
The reason for this is the extreme expense of rollbacks and the size of large transactions. To put this in perspective, try a large transaction in any database within an explicit transaction and roll it back vs. allowing it to commit. Although performance varies from version to version of ASE as well as the transaction itself, a normal transaction may take a full order of magnitude longer to rollback than it takes to fully execute (i.e. a transaction with an execute time of 6 minutes may require an hour to rollback). By surfacing the offending conflict earlier rather than later, the rollback time of the large transaction is reduced. This is crucial as no other transaction activity is re-initiated until all the rollbacks have completed. Consequently, without the periodic check for contention by selecting rs_threads every dsi_large_xact_size rows, a large transaction could have a significantly large “penalty” (i.e. 900 rows for the bcp example). This is illustrated in the below diagram – a slight modification of the above – with the intermediate rs_thread selects grayed out.
Now then, getting back to the point earlier discussed in the previous section – the temptation to reduce dsi_large_xact_size until most transactions qualify – with the goal of reducing latency. To understand why this is a bad idea, consider the following points:

- Large transactions are never grouped. Consequently, this eliminates the benefits of transaction grouping and increase log I/O and rs_lastcommit contention.

- In order to ensure most transactions qualify, dsi_large_xact_size has to be set fairly low (i.e. 10). The problem with this is that every 10 rows, the large DSI threads would block waiting for the other threads to commit. If the average transaction was 20 statements and 5 large transaction threads were used, the first would have all 20 statements executing while the other 4 would execute up to the 10th and block. The higher the ratio of dsi_large_xact_size to average transaction size, the more the performance degradation. By contrast – a serialization method of “none” would let all 5 threads execute up to the 20th statement before blocking.

- The serialization between large transaction threads is essentially none up to the point of the first dsi_large_xact_size rows – since we are not waiting for the commits at all (let alone waiting until they are ready to be sent). If the transactions have considerable contention between them to the extent wait_for_commit would have been a better serialization method, the large transactions could experience considerable rollbacks and retries. After the first dsi_large_xact_size rows, the rs_threads blocking changes the remainder of the large transaction to more of a wait_for_commit serialization.

The last bullet takes a bit of thinking before it can be understood. Let say we have a novice Replication System Administrator (named Barney) who has diligently read the manuals, took the class – but didn’t test his system with a full transaction load (nothing abnormal here – in fact, it is rarity – and a shame – these days to note that few if any of large IT organizations stress test their applications or even have such a capability). However, being a “daring” individual, Barney decides to capitalize on the large transaction advantage of reading from the SQT Open queue and sets dsi_num_threads to 5, dsi_num_large_xact_threads to 4 and finally sets dsi_large_xact_size to 5 (his average number of SQL statements set from the application – a web order entry system). Now then, let’s assume due to triggered updates for shipping costs, inventory tracking, customer profile updates, etc., the 5 SQL statements expands to a total of 12 statements per transaction (not at all hard). What Barney assumes he is getting looks similar to the following:

The expectation: everything is done at T05. What Barney actually gets is more like:

![Figure 43 – Wishful Concurrent Large Transaction DSI Threads](image)
This illustrates how the first dsi_large_xact_size rows are similar to a serialization method of “none” while those statements after transition to more of a wait_for_commit. By the way, consider the impact if the last statement in thread 4 conflicts with one of the first rows in thread 5. A rollback at T12.

Now, the unbeliever would be quick to say that the dsi_large_xact_size could be increased to exactly the rows in the transaction (i.e. 12) at which point we would really have the execution timings in the earlier figure. Possibly – be real hard as the number of statements in a transaction is not a constant. However, remember – we have now lost transaction grouping, introduced a high probability of contention/rollbacks, increased load on rs_lastcommit and replicate transaction log – all for very little gain in latency for smaller transactions. While not denying that in some very rare instances of Warm Standby with a perfect static transaction size with no contention between threads that there is a probability that this type of implementation might help a small amount – the reality is that it is highly improbable - especially given the concurrent transaction induced rollback earlier discussed.

Thread Allocation

A little known and undocumented fact is that dsi_num_large_xact_threads are reserved out of dsi_num_threads exclusively for large transactions. That means only 3 threads are available for processing normal transactions if you set the default connection parameter of “parallel_dsi” to “on” without adjusting any of the other parameters (parallel_dsi “on” sets dsi_num_threads to 5 and dsi_num_large_xact_threads to 2 – leaving only 3 threads for normal transactions of <100 rows (at default)). This can surprise some administrators – who in checking their replicate dataserver – discover that “only” a few of the configured threads are active. Combining this with the previous topic yields another key to understanding Parallel DSI’s:

**Key Concept #23:** For most systems, it is extremely doubtful that more than 2 large transaction threads will improve performance. In addition, since large transaction threads are “reserved”, increasing the number of large transaction threads may require increasing the total number of threads to avoid impacting (small) normal transaction delivery rates.

Maximizing Performance with Parallel DSI’s

By now, you have enough information to understand why the default settings for the parallel_dsi connection parameter are what they are in respect to threading – and why this may not be the most optimal. Consider the following review of points from above:

- In keeping with Replication Server’s driving philosophy of maximizing resilience, the default serialization method is “wait_for_commit” as this minimizes the risk of inter-thread contention causing significant rollbacks.
- When using the “wait_for_commit” serialization method, only 3 Parallel DSI’s will be effective. Using more than this number will not bring any additional benefit.

- For most large transactions – due to the early conflict detection algorithm – no more than 2 large transaction threads will be effective. After this point, no more benefit will be realized as the next large transaction could reuse the first thread.

However, this may not be even close to optimal as the assumption is that there will be significant contention between the Parallel DSI’s and the large transactions are significantly higher than dsi_large_xact_size setting. If this is not true for your application (typically the case), then the default “parallel_dsi” settings are inadequate. To determine the optimal settings, you need to understand the profile of the transactions you are replicating, eliminate any replication or system induced contention at the replicate and develop Parallel DSI profiles of settings corresponding to the transaction profile during each part of the business day.

Understanding Replicated Transaction Profile

In order to determine if the contention at the replicate (if there is any) is due to replication or schema induced contention, you need to develop a sense of the transaction profile being executed at the primary during each part of the business day. Consider the following fictitious profile:

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Type</th>
<th>Time Range</th>
<th>Volume (tpd)</th>
<th>Leading Contention Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute Trade</td>
<td>OLTP</td>
<td>0830 - 1630</td>
<td>500,000</td>
<td>get next trade id</td>
</tr>
<tr>
<td>Place Order</td>
<td>OLTP</td>
<td>0700 - 1900</td>
<td>750,000</td>
<td>get next order id</td>
</tr>
<tr>
<td>Adjust Stock Price</td>
<td>OLTP</td>
<td>0830 - 1630</td>
<td>625,000</td>
<td>place order read</td>
</tr>
<tr>
<td>401K Deposit</td>
<td>Batch</td>
<td>0500 – 0700</td>
<td>125,000*</td>
<td>mutual fund balance</td>
</tr>
<tr>
<td>Money Market Deposit</td>
<td>OLTP</td>
<td>0900 - 1700</td>
<td>1,000</td>
<td>central fund update</td>
</tr>
<tr>
<td>Money Market Check</td>
<td>Batch</td>
<td>1800 - 2200</td>
<td>750</td>
<td>central fund withdrawal</td>
</tr>
<tr>
<td>Close Market</td>
<td>Batch</td>
<td>1700 - 1930</td>
<td>1</td>
<td>isolation level 3 aggregation</td>
</tr>
<tr>
<td>Purge Historical</td>
<td>Batch</td>
<td>2300 - 2359</td>
<td>1</td>
<td>Index maintenance</td>
</tr>
</tbody>
</table>

* Normalized for surge occurring on regular periodic basis

Note that the first two OLTP transactions have a monotonic key contention issue. When replicating this transaction, the id value will be known, therefore, this will not cause contention at the replicate. Accordingly, we would be most interested in what the second leading cause of contention is, however, we may not be able to determine that as the first one may be masking it.

Also, in the above list of sample transactions, some of the OLTP transactions not only affect individual rows of data representing one type of business object (such as customer account) – but they also affect either an aggregate (central fund balance) or other business object data. The contention could be on the latter. For example, each individual 401K pay deposit affects the individual investor’s account. In addition, it also adjusts their particular fund’s pool of receipts with which the fund manager uses for reinvestment. It is the activity against the fund data that could be the source of contention and not the individual account data.

Eliminating Contention at Replicate

Having understood where the contention occurs at the primary, you then have to look at where contention is at the replicate. It is unfortunate, but almost in every case in which a customer has called Sybase Support with Replication Server performance issues, few have bothered to investigate if and where contention is the cause. This is especially true in Warm Standby scenarios in which the replicate system is the only updated by the Replication Server (and attempting a serialization method of “none”). Additionally, in the few cases where the administrators have been brave enough to attempt the “none” serialization method, as soon as the first error that occurs stating a parallel transaction failed and had to be retried in serial, the immediate response is to switch back to wait for commit vs. eliminating the contention – or even determining if that level of contention is acceptable. In one example of the latter, during a bulk load test in a large database, the queue got 1GB behind after 1 hour using “wait_for_commit”. After switching to “none”, the queue was fully caught up in 30 minutes. However, during that period, approximately 3 parallel
transactions failed per minute and were retried in serial. The trade-off was considered more than acceptable – 90 errors and empty queue vs. no errors and 1GB backlog. Just think though – if you were able to eliminate the contention that caused even 50% of the failures – the number of additional transactions per minute would be at least equivalent to the number of DSI’s. For example, in this case, 10 DSI’s were in use. This means an extra 15 transactions (3 * 0.50 * 10) could have been applied per minute – or 450 transactions during that time. And this is an extremely low estimate as we have not include the time it took to reapply the transactions in serial – during which the system could still be applying the transactions in parallel.

Which brings us back to the point – how can we eliminate the contention at the replicate? The answer is (of course) it all depends on what the source of contention is – is it contention introduced as a result of replication or contention between replication and other users.

**Replication Induced Contention**

As discussed earlier, replication itself can induce contention – frequently resulting in the decision to use unoptimal Parallel DSI serialization methods. For normal transactions, a serialization method of “none” will achieve the highest throughput. The goal is to eliminate any replication induced contention that is preventing use of “none” and then to assess whether the level of parallel transaction retries is acceptable. As discussed earlier, the main cause of contention directly attributable to replication is the transaction grouping. Transaction grouping is a good feature, however, at its default of 20 transactions per group, it can frequently lead to contention at the replicate that didn’t exist in the primary. The easiest way to resolve this is to simply reduce the dsi_max_xacts_in_group parameter until most of the contention is resolved. A possible strategy is to simply halve the dsi_max_xacts_in_group repeatedly until the replication induced contention is nearly eliminated. While it is theoretically possible to eliminate all replication-induced contention caused by transaction grouping in this manner, there is a definite tradeoff in eliminating transaction grouping and the associated increase in log and I/O activity and a limited acceptance of some contention. This means you will need to be willing to accept some degree of parallel transactions failing and being retried. If you remember, in an earlier session we mentioned that in one system, Replication got 1GB behind using “wait_for_commit”. By switching to “none”, Replication Server not only was able to keep up, it was able to fully drain the 1GB backlog in less than 30 minutes. During that time, however, an average of 3 parallel transactions per minute failed and were retried. This was completely acceptable considering the relative gain in performance.

**Concurrency Induced Contention**

In a sense, the transaction grouping is a form of concurrency that is causing contention. In addition to transaction grouping, the mere fact that Parallel DSI’s are involved means that the individual Parallel DSI’s could experience contention between them as well as with other users on the system. Possible areas of contention include:

- Replication to aggregate rollups in which many source transactions are all attempting to update the same aggregate row (i.e. total sales) in the destination database.
- DML applied serially at source that is being applied in parallel at replicate in which contention exists. For example, a (slow) bcp at primary does not have any contention. However, if the bcp specified a batch size (using –b), then the Replication Server may send the individual batches using Parallel DSI’s. The result is last page contention or index contention at the replicate.
- Replicated transactions that had contention at the primary.
- Transactions that have contention at the replicate due to the timing of delivery where at the primary no contention existed due to different timings. The timing difference could be the result of Replication Server component availability (i.e. Replication Agent was down) or due to long running transactions at the replicate delaying the first transaction until the conflicting transaction was also ready to go (i.e. a long running procedure at replicate would delay further transactions).

How and if this contention could be eliminated depends on the type of contention. For example, where contention exists at index or page level for data tables, but not on the same rows, changing the replicate system to use datapage or datarow locking may bring relief.

**Developing Parallel DSI Profiles**

Similar to managing named data caches in Adaptive Server Enterprise, you may have to establish DSI profiles to manage replication performance during different periods of activity. Consider the following table of example settings:
Developing a similar profile for your replication environment will enable the Replication Server to avoid potentially inhibitive deadlocks and retries during long transactions such as large bcp and high incidence SQL statements typical of post-daily processing routines. For small and large bcp loads, however, remember to use the –B option to breakup potentially queue filling bulk loads of data.

Key Concept #24: Maximum performance using Parallel DSI’s can only be achieved after replication and concurrency caused contention is eliminated and DSI profiles (based on the transaction profile) are developed to minimize contention between Parallel DSI’s.
Text/Image Replication

Okay, just exactly how is Replication Server able to replicate non-logged text/image updates???

The fact that Replication Server is able to do this surprises most people. However, if you think about it – the same way that ASE had to provide the capability to insert 2GB of text into a database with a 100MB log – Replication Server had to provide support for it – AND also be able to insert this same 2GB of text into the replicate without logging it for the same reason. The unfortunate problem is that text/image replication can severely hamper Replication Server performance – degrading throughput by 400% or more in some cases. Unfortunately, other than not replicating text, not a lot can be done to speed this process up.

Text/Image Datatype Support

To understand why not, you need to understand how ASE manages text. This is simply because the current biggest limiter on replicating text is the primary and replicate ASE’s themselves. While we are discussing mainly text/image data, remember, this applies to off row java objects as well as these are simply implemented as image storage. Throughout this section, any reference to “text” datatypes should be treated as any one of the three Large Object (LOB) types.

Text/Image Storage

From our earliest DBA days, we are taught that text/image data is stored in a series of page chains separate from the main table. This allows an arbitrary length of text to be stored without regard to the data page limitation of 2K (or ~1960 bytes). Each row that has a text value stores a 16-byte value – called the “text pointer” or textptr – that points to the where the page chain physically resides on disk. While this is good knowledge, a bit more knowledge is necessary for understanding text replication.

Unlike normal data pages with >1900 bytes of storage, each text page can only store 1800 bytes of text. Consequently a 500K chunk of text will require at least 285 pages in a linked page chain for storage. The reason for this is that each text page contains a 64-byte Text Image Page Statistics Area (TIPSA) and a 152-byte Sybase Text Node (st-node) structures located at the bottom of the page.

Typically, a large text block (such as 500K) will be stored in several runs of sequential pages – with the run length depending on concurrent I/O activity to the same segment and available contiguous free space. For example, the 285 pages needed to store 500K of text may be arranged in 30 runs of roughly 10 pages each. Prior to ASE 12.0, updating the end of the text chain – or reading the chain starting at a particular byte offset (as is required in a sense), meant beginning at the first page and scanning each page of text until the appropriate byte count was reached. As of ASE 12.0, the st-node structure functions similar to the Unix File System’s I-node structure in that it contains a list of the...
first page in each run and the cumulative byte length of the run. For simplicity sake, consider the following table for a 64K text chunk spread across 4 runs of sequential pages on disk:

<table>
<thead>
<tr>
<th>Page Run (page #’s)</th>
<th>st-node page</th>
<th>byte offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 (300-307)</td>
<td>300</td>
<td>14400</td>
</tr>
<tr>
<td>16 (410-425)</td>
<td>410</td>
<td>43200</td>
</tr>
<tr>
<td>8 (430-437)</td>
<td>430</td>
<td>57600</td>
</tr>
<tr>
<td>5 (500-504)</td>
<td>500</td>
<td>65536</td>
</tr>
</tbody>
</table>

This allows ASE to rapidly determine which page needs to be read for the required byte offset without having to scan through the chain. Depending on how “fragmented” the text chain is (i.e. how many runs are used) and the size of the text chain itself, the st-node may require more than 152 bytes. Rather than use the 152 bytes on each page and force ASE to read a significant portion of the text chain simply to read the st-node the first 152 bytes are stored on the first page while the remainder is stored in it’s own page chain (hence the increase in storage requirements for ASE 12.0 for text data vs. 11.9 and prior systems).

It goes without saying, then, that Adaptive Server Enterprise 12.0+ should be considerably faster at replicating text/image data then preceding versions. Thanks to the st_node index, the Replication Agent read of the text chain will be faster and the DSI delivery of text will be faster as neither one will be forced to repeatedly re-read the first pages in the text chain simply to get to the current byte offset where currently reading/writing text.

The first page in the chain – pointed to by the 16-byte textptr is called the First Text Page or FTP. It is somewhat unique in that when a text chain is updated, it is never deleted (unless the data row is deleted). This is surprising but true and still true when setting the text value explicitly to null still leaves this page allocated – simply empty. The textptr is a combination of the page number for the FTP plus a timestamp. The FTP is important to replication because it is on this page that the TIPSA contains a pointer back to the data row it belongs to. So, while the data row contains a textptr to point to the FTP, the FTP contains the Row ID (RID) back to the row. Should the row move (i.e. get a new RID), the FTP TIPSA must be updated. The performance implications of this at the primary server is fairly obvious (consequently, movements of data rows containing text columns should be minimized)

**Standard DML Operations**

Text and image data can be directly manipulated using standard SQL DML Insert/Update/Delete commands. As we also were taught, however, this mode of manipulation logs the text values as they are inserted or updated and is extremely slow. The curious might wonder how a 500K text chunk is logged in a transaction log with a fixed log row size. The answer is that the log will contain the log record for the insert and subsequent log records with up to 450 bytes of text data – the final number of log records dependent on the size of the text and the session’s textsize setting (i.e. set textsize 65536).

**SQL Support for Text/Image**

In order to speed up text/image updates and retrievals as well as provide the capability to insert text data larger than permissible by the transaction log, Sybase added two other verbs to the Transact SQL dialect – readtext and writetext. Both use the textptr and a byte offset as input parameters to determine where to begin read or writing the text chunk. In addition, the writetext command supports a NOLOG parameter which signals that the text chunk is not to be logged in the transaction log. Large amounts of text simply can be inserted or updated through repetitive calls to writetext specifying the byte offset to be where previous writetext would have terminated.

Of special consideration from a replication viewpoint is that the primary key for the row to which the text belongs is never mentioned in the writetext function. The textptr is used to specifically identify which text column value is to be changed instead of the more normal where clause structure with primary key values. Hold this thought until the section on Replication Agent processing below.

**Programming API Support**

Anyone familiar with Sybase is also familiar (if only in name) with the Open Client programming interface - which is divided into the simple/legacy DB-Lib (Database Library) API interface and the more advanced CT-Lib (Client
Library) interface. Using either, standard SQL queries – including DML operations – can be submitted to the ASE database engine. Of course, this is one way to actually modify the text or image data – but as we have all heard, DML is extremely slow at updating text/image and forces us to log the text as well (which may not be supportable). Consequently, both support API calls to read/write text data to ASE very similar to the readtext/writetext functions described above. For example, in CT-Lib, ct_send() is used to issue SQL statements to the dataserver while ct_get_data() and ct_send_data() are used to read/write text respectively. Similar to writetext, ct_send_data supports a parameter specifying whether the text data is to be logged. Note that while we have discussed these functions as if they followed readtext/writetext implementation, in reality, the API functions basically set the stage for the SQL commands instead of the other way around. In any case, similar to writetext, the sequence for inserting a text chunk using the CT-LIB interface would look similar to:

- ct_send() -- send the insert statement with dud data for text (init pointer)
- ct_send() -- retrieve the row to get the textptr just init’d
- ct_send_data() -- send the first text chunk
- ct_send_data() -- send the next text chunk
- ct_send_data() -- send the last text chunk

The number of calls dependent on how large of a temporary buffer the programmer wishes to use to read the text (probably from a file) into memory and pass to the database engine. A somewhat important note is that the smaller the buffer, the more likely the text chain will be fragmented and require multiple series of runs.

Of all the methods currently described, the ct_send_data() API interface is the fastest method to insert or update text in a Sybase ASE database.

**RS Implementation & Internals**

Now that we now how text is stored and can be manipulated, we can begin applying this knowledge to understand what the issue is with replicating text.

**sp_setreptable Processing**

If not the single most common question, the question “Why does sp_setreptable take soooo long when executed against tables containing text or image columns?” certainly ranks in the top ten questions asked to TSE. The answer is truthfully – to fix an oversight that ASE engineering “kinda forgot”. If you remember from our previous discussion, the FTP contains the RID for the data row in its TIPSA. The idea is that simply by knowing what text chain you were altering, you would also know what row it belongs to. This is somewhat important. If a user chose to use writetext or ct_send_data(), a lock should be put on the parent row to avoid data concurrency issues. However, ASE engineering chose instead to control locking via locking the FTP itself. In that way (lazily) they were protected in that updates to the data row also would require a lock on the FTP (and would block if someone was performing a writetext) and concurrent writetexts would block as well. Unfortunately for Replication Server Engineering, this meant that ASE never maintained the TIPSA data row RID if the RID was never initialized – which frequently was the case – especially in databases upgraded from previous releases prior to ASE 12.0. In order to support replication, the TIPSA must be initialized with the RID for each data row. Consequently, sp_setreptable contains an embedded DBCC command that scans the table and for each data row that contains a valid textptr, it updates the column’s FTP TIPSA with the RID. Since a single data row may contain more than one text or image column, this may require more than one write operation. To prevent phantom reads and other similar issues, this is done within the scope of a transaction, effectively locking the entire table until this process completes. Unfortunately, as you can imagine, this is NOT a quick process. On a system with 500,000 rows of data containing text data (i.e. 500,000 valid text pointers), it took 5 hours to execute sp_setreptable (effectively 100,000 textptrs/hour – usual caveat of your time may vary is applicable). An often used metric is that the time required is the same as that to build a new index (assuming a fairly wide index key so the number of i/o’s are similar).

**Key Concept #25:** The reason sp_setreptable takes a long time on tables containing text/image columns, is that it must initialize the First Text Page’s TIPSA structure to contain the parent row’s RID.

There is a supported method around this problem provided that existing text values in a database will never be manipulated via writetext or ct_send_data(). That method is to use the legacy sp_setreplicate procedure which does not
support text columns and then call sp_setrepcol as normal to set the appropriate mode (i.e. replicate_if_changed). This executes immediately and supports replication of text data manipulated through standard DML operations (insert/update/delete) as well as new text values created with the writetext and ct_send_data methods and slow bcp operations.

### Replication Agent Processing

Now, the nagging question – “Why on earth is initializing the FTP TIPSA with the RID so critical??” Some may already have guessed. If a user specifies a non-logged writetext operation and only modifies the text data (i.e. no other columns in row changed), then it would be impossible for the Replication Server to determine which row the text belonged to at the replicate. Remember, replicated databases have their own independent allocation routines, consequently, even in Warm Standby, there is no way to guarantee that because a particular text chain starts at page 23456 at the primary that the identical page will be used at the replicate. This is especially true in non-Warm Standby architectures such as shared primary or corporate rollup scenarios in which the page more than likely will be allocated to different purposes (perhaps an OAM page in one, while a text chain in the other).

As a result, the Replication Server MUST be able to determine the primary keys for any text column modified. As you could guess, this lot falls to the task of the Replication Agent. While we have used the term “NOLOG” previously, as those with experience know, in reality, there is no such thing as an “unlogged operation” in Sybase. Instead, operations are considered “minimally logged” – which means that while the data itself is not logged, the space allocations for the data are logged (required for recovery). In addition to logging the space allocations for text data, the text functions internal within ASE check to see what the replication status is for the text column any time it is updated. If the text column is to be replicated, ASE inserts a log row in the transaction log containing the normal logging information (transaction id, object id, etc.) as well as the textptr.

The Replication Agent reads the log record, extracts the textptr and parses the page number for the text chain. Then it simply reads the FTP TIPSA for the RID (itself a combination of a page number and row id) along with table schema information (column names and datatypes as normal) and reads the parent row from the data page. If the text chain was modified with a writetext, the Replication Agent tells the Replication Server what the primary keys were by first sending a rs_datarow_for_writetext function with all of the columns and their values.

**Key Concept #26**: The Replication Agent uses the FTP TIPSA RID to locate the parent row and then constructs a replicated function rs_datarow_for_writetext to send with the text data to identify the row at the replicate.

In either case – text modified via DML or writetext – similar to transaction logging of text data, in order to send data to the Replication Server, the Replication Agent must break up the text into multiple chunks and send via multiple rs_writetext “append” calls. An example of this from a normal logged insert of data is illustrated in the below LTL block (notice the highlighted sections).
already has started to in order to find the RID on the FTP TIPSA.

As you could guess, even when not logging the text, the Replication Agent can simply read the text chain (after all, it
call to rs_get_textptr to retrieve the textptr for the text chain allocation just created. Once it receives the textptr, the
typically an update statement for the text column setting it to a temporary string constant. It then follows this with a

At the replicate, we are lacking something fairly crucial – the
contain the insert and multiple inserttext log records. The replication agent, as we saw from above, translates this into
Replicated Text Functions

A couple of points are illustrated above:

- The base function (insert/update) contains the replication status and also whether or not the column contains
data. In the last example, “notrep” refers to the fact that the text chain is empty.
- The text replication is passed through a series of rs_writetext append first, append, append, …., append last
functions with each specifying the number of bytes.

As you could guess, even when not logging the text, the Replication Agent can simply read the text chain (after all, it
already has started to in order to find the RID on the FTP TIPSA).

Key Concept #27: Similar to the logging of text data, text data is passed to the
Replication Server by “chunking” the data and making multiple calls until all the
text data has been sent to the Replication Server.

RS & DSI Thread Processing

As far as Replication Server, text data is handled no differently than any other, except of course, that the DIST thread
needs to associate the multitude of rows with the subscription on the DML function (rs_insert) or as designated by the
rs_datarow_for_writetext. You may have wondered previously why the rs_datarow_for_writetext didn’t simply
contain only the primary key columns vs. the entire row. There actually are two reasons: 1) the DBA may have been
lazy and not actually identified the primary key (used a unique index instead); and 2) subscriptions on non-primary key
searchable columns would be useless. The latter is probably the most important of the two – without all of the
columns, if a site subscribed to data from the primary based on a searchable column (i.e. state in pubs2.authors), the
site would probably never receive any text data. However, by providing all data, the DIST thread can check for
searchable columns within the data row to determine the destination for the text values.

The bulk of the special handling for text data within the Replication Server is within the DSI thread. First, the DSI
thread treats text as a large transaction. In itself, this is not necessarily odd as often text write operations result in a
considerable number of rows in the replication queues. However, the biggest difference is how the DSI handles the
text from a replicated function standpoint.

Replicated Text Functions

As we discussed earlier, when a text row is inserted using regular DML statements at the primary, the primary log will
contain the insert and multiple inserttext log records. The replication agent, as we saw from above, translates this into
the appropriate rs_insert and rs_writetext commands. At the replicate, we are lacking something fairly crucial – the
textptr. Consequently, the DSI first sends the rs_insert as normal and then follows it with a call to rs_init_textptr –
typically an update statement for the text column setting it to a temporary string constant. It then follows this with a
call to rs_get_textptr to retrieve the textptr for the text chain allocation just created. Once it receives the textptr, the
DSI uses the CT-LIB ct_send_data() function to actually perform the text insert. From a timeline perspective, this looks like the below:

For text inserted at the primary using writetext or ct_send_data, the sequence is little different. As we discussed before, because the textreq function within the ASE engine is able to determine if the text is to be replicated – even when a non-logged text operation is performed, ASE will put a log record in the transaction log. The Replication Agent in reading this record, retrieves the RID from the TIPSA and then creates an rs_datarow_for_writetext function. After that, the normal rs_writetext functions are sent to the Replication Server. The DSI simply does the same thing. It first sends the rs_datarow_for_writetext to the replicate. It then is followed by the rs_init_textptr and rs_get_textptr functions as above.

The role of rs_datarow_for_writetext is actually two fold. Earlier, we discussed the fact that it is used to determine the subscription destinations for the text data. For rows inserted with writetext operations, it is also used to provide the column values to the rs_init_textptr and rs_get_textptr function strings so the appropriate row for the text can be identified at the replicate and have the textptr initialized.

The sequence of calls for replicating text modified by writetext or ct_send_data is illustrated below:
This brings the list of function strings to 4 for handling replicated text. Thankfully, if using the default function classes (rs_sqlserver_function_class or rs_default_function_class), these are generated for you. However, what if you are using your own function class?? If using your own function class, you will not only need to create these four function strings, but you will also need to understand the following:

- Text function strings have column scope. In other words, you will have to create a series of function strings for each text/image column in the table. If you have 2 text columns, you will need two definitions for rs_get_textptr, etc.
- The textstatus modifier available for text/image columns in normal rs_insert, rs_update, rs_delete as well as rs_datarow_for_writetext, rs_init_textptr is crucial to avoid allocating text chains when no text data was present at the primary.

In regards to the first bullet, the text function strings for each text column is identified by the column name after the function name. In the following paragraphs, we will be discussing these functions in a little bit more detail.

**Text Function Strings**

Consider the pubs2 database. In that database, the blurbs table contains biographies for several of the authors in a column named “copy”. If we were to create function strings for this table, they might resemble the below:

```sql
create function string blurbs.rs_datarow_for_writetext;copy
for sqlserver2_function_class
output language ''
```

Note the name of the column in the function string name definition. As noted earlier, the rs_datarow_for_writetext is sent when a writetext operation was executed at the primary. In the default function string classes, this function is empty for the replicate – the rs_get_textptr function is all that will be necessary. However, in the case of a custom function class, you may want to have this function perform something – for example insert auditing or trace information into an auditing database.

Typically the next function sent is the rs_init_textptr, which might look like the below:

```sql
create function string blurbs.rs_textptr_init;copy
for sqlserver2_function_class
output language 'update blurbs set copy = ''Temporary text to be replaced''
                               where au_id = ?au_id!new?'
```

This, at first appears to be a little strange. However, remember, we need a valid text pointer before we start using writetext operations. But since we haven’t sent any text yet….kind of a catch-22 situation. Consequently, we simply use a normal update command to insert some temporary text into the column knowing that the real text will begin at an offset of 0 and therefore will write over top of it. Note that in the examples in the book, it sets the column to a null value. This can be problematic. Although setting a text column to null is supposed to allocate a text chain, in earlier versions of SQL Server, it was no guarantee that setting the text column to null would do so (in fact, it seemed that ~19 bytes of text was the guidelines for System 10.x). In addition, due to the space taken up for a text chain, there is no guarantee that future versions of ASE will not “short circuit” this problem by not allocating a text chain when the text is null. Consequently, it is best to ensure that the text chain is indeed allocated when needed.

After initializing the textptr, the next function Replication Server sends is the rs_get_textptr function.

```sql
create function string blurbs.rs_get_textptr;copy
for sqlserver2_function_class
output language 'select copy from blurbs
                               where au_id = ?au_id!new?'
```

Those who have worked with SQL text functions may be surprised at the lack of a textptr() function call in the output mask as in “select textptr(copy) from …”. This is deliberate. Those familiar with CT-Lib programming know that when a normal select statement without the textptr function is used, it is the pointer itself that is bound using ct_bind() and ct_fetch() calls. The textptr() function solely exists so that those using the SQL writetext and readtext commands can pass it a valid textptr. The CT-Lib API essentially has it built-in as it is only with the subsequent ct_get_data() or ct_send_data() calls that the actual text is manipulated. Since Replication Server uses CT-Lib API calls to manipulate text, the textptr() function is then unnecessary.
Of special note, it is often the lack of a valid textptr – or more than one – that frequently will cause a Replication Server DSI thread to suspend. If this should happen, check the queue for the proper text functions as well as check the RSSD for fully defined function string class. The error could be transient, but it also could point to database inconsistencies where the parent row is actually missing.

Finally, the text itself is sent using multiple calls to rs_writetext. The rs_writetext function can perform the text insert in three different ways. The first is the more normal writetext equivalent as in:

```sql
create function string blurbs.rs_writetext;
for rs_sqlserver2_function_class
output writetext
use primary log
```

In this example, RS will use ct_send_data() API calls to send the text to the replicate using the same log specification that was used at the primary. While this is the simplest form of the rs_writetext functions, it is probably the most often used as it allows straightforward text/image replication between two systems that provide ct_send_data() for text manipulation (and therefore one of the biggest problems in replicating through gateways). An alternative is the RPC mechanism, which can be used to replicate text through an Open Server:

```sql
create function string blurbs.rs_writetext;
for gw_function_class
output rpc
'execute update_blurbs_copy
@copy_chunk = ?copy!new?,
@au_id = ?au_id!new?,
@last_chunk = ?rs_last_text_chunk!sys?,
@writetext_log = ?rs_writetext_log!sys?'
```

This also could be used to replicate text from a source database to a target in which the text has been split into multiple varchar chunks. Note that in this case, two system variables are used to flag whether this is the last text chunk and whether it was logged at the primary. The former could be used if the target is buffering the data to ensure uniform record lengths (i.e. 72 characters) and to handle white space properly. When the last chunk is received, the Open Server could simply close the file – or if a dataserver, it could update the master record with the number of varchar chunks. Note that the Replication Server handles splitting the chunks of text into 255 byte or less chunks avoiding datatype issues.

The final method for rs_writetext is in fact to prevent replication via no output.

```sql
create function string blurbs.rs_writetext;
for rs_sqlserver2_function_class
output none
```

Which disables text replication no matter what the setting of sp_setrepcol.

**Text Function Modifiers**

The second aspect of text replication that takes some thought, is the role of the text variable modifiers. While other columns support the usual old and new modifiers for function strings as in `?au_lname!new?`, text does not support the notion of a before and after image. The main reason for this, is that while the text rows may be logged, unlike normal updates to tables, the before image is not logged when text is updated. Additionally, if the primary application opts not to log the text being updated, the after image isn’t available from the log either. While it is true that the text does get replicated, so that in a sense an “after image” does exist, remember, that text is replicated in chunks, consequently a single cohesive after image is not available. Even if it were, the functionality would be extremely limited as the support for text datatypes is extremely reduced.

However text columns do support two modifiers: new and text_status. Before you jump and say “wait a minute, didn’t you just say…”, the answer is sort of. In the previous paragraph, we were referring to the old and new as it applies to the before and after images captured from the transaction log. The new text modifier instead refers to the current chunk of text contents without referring to whether it is the old or new values. For example, if left at “always_replicate”, if a primary transaction updates a column in the table other than the text column and minimal column replication is not on, then the text column will be replicated. In this scenario, the “new” chunks are really the “old” values which are still the same. The whole purpose of “new” in this sense was to provide an interface into the text chunks as they are provided through the successive rs_writetext commands. An example of this can be found near the end of the previous section when discussing the RPC mechanism for replicating text to Open Servers (which could then write it to a file). In that example (repeated below), the “new” variable modifier was used to designate the text chunk string vs. the columns text status.
create function string blurbs.rs_writetext;copy
for gw_function_class
output rpc
'execute update_blurbs_copy
@copy_chunk = ?copy!new?,
@au_id = ?au_id!new?,
@last_chunk = ?rs_last_text_chunk!sys?,
@writetext_log = ?rs_writetext_log!sys?'

For non-RPC/stored procedure mechanisms, text columns also support the text_status variable modifier, which specifies whether the text column actually contains text or not. The values for text_status are:

<table>
<thead>
<tr>
<th>Hex</th>
<th>Dec</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>0</td>
<td>Text field contains NULL value, and the text pointer has not been initialized.</td>
</tr>
<tr>
<td>0x0002</td>
<td>2</td>
<td>Text pointer is initialized.</td>
</tr>
<tr>
<td>0x0004</td>
<td>4</td>
<td>Real text data will follow.</td>
</tr>
<tr>
<td>0x0008</td>
<td>8</td>
<td>No text data will follow because the text data is not replicated.</td>
</tr>
<tr>
<td>0x0010</td>
<td>16</td>
<td>The text data is not replicated but it contains NULL values.</td>
</tr>
</tbody>
</table>

During normal text replication, these modifiers are not necessary. However, if using custom function strings, these status values allow you to customize behavior at the replicate – for example, avoiding initializing a text chain when no text exists at the primary. Consider the following:

```
create function string blurbs_rd.rs_update
for my_custom_function_class with overwrite
output language
  if ?copy!text_status? < 2
    -- do nothing since no text was modified
  else if ?copy!text_status? = 2 or ?copy!text_status? = 4
    insert into text_change_tracking (xactn_id, key_val)
    values (?rs_origin_xactn_id!sys?,?au_id!new?)
  else if ?copy!text_status? = 8
    -- text is not replicated
  else if ?copy!text_status? = 16
    insert into text_change_tracking (xactn_id, key_val, text_col)
    values (?rs_origin_xactn_id!sys?, ?au_id!new?,
      "(text was deleted or set to null at the primary)"
    )
```

The above function string – or one similar – could be used as part of an auditing system that would only allocate a text chain when necessary – and also signal when the primary text chain may have been eliminated via being set to null.

**Performance Implications**

As mentioned earlier, the throughput for text replication is much, much lower than for non-text data. In fact, during a customer benchmark in which greater than 2.5GB/hr was sustainable for non-text data, only 600MB/hr was sustainable for text data (or 4x worse). The reason for this degradation is somewhat apparent from the above discussions.

**Replication Agent Processing**

It goes without saying that if the text or image data isn’t logged, then the Replication Agent has to read it from disk – and more than likely physical reads. While the primary transaction may have only updated several bytes by specifying a single offset in the writetext function, the Replication Agent needs to read the entire text chain.

As it reads the text chain, if the original function was a writetext or ct_send_data, it first has to read the row’s RID from the FTP TIPSA, read the row from the base table and construct the rs_datarow_for_writetext function as well. Then as it begins to scan the text chain, it begins to forward the text chunks to the Replication Server. While reading the text chain, all other Rep Agent activity in the transaction log is effectively paused. In highly concurrent or high volume environments, this could result in the Replication Agent getting significantly behind.
Replication Server Processing

Within the Replication Server itself, replicating text can have performance implications. First, it will more than likely fill the SQT cache – and also be the most likely victim of a cache flush meaning it will have to be read from disk. Consequently, not only will the stable queue I/O be higher due to the large number of rs_writetext records required, but also during the transaction sorting, it is almost guaranteed that it will have to be re-read from disk.

The main impact within the Replication Server however, is at the DSI thread. Consider the following points:

- Text transactions can’t be batched
- The DSI has to get the textptr before the rest of the text can be processed. This requires more network interaction than most other types of commands.
- Each rs_writetext function is sent via successive calls to ct_send_data(). While this is the fastest way to handle text, it is not fast. Consider the fact that in ASE versions prior to ASE 12.0, the database engine would have to scan the text pages to find the byte offset. Consequently, processing a single rs_writetext is slower than an rs_insert or other similar normal DML function.

Net Impact

Replicating text will always be considerably slower than regular data. If not that much text is crucial to the application, then replicating text may not have that profound of an impact on the rest of the system. However, if a lot of text is expected, then performance could be severely degraded. At this juncture, application developers have really only three choices:

1. Replicate the text and endure the performance degradation.
2. Use custom function strings to construct a list of changed rows and then asynchronously to replication, have an extraction engine move the text/image data
3. Don’t replicate text/image at all

Which one is best is determined by the business requirements. For most workflow automation systems, the text is irrelevant and therefore simply can be excluded from replication. However, for high availability architectures involving a Warm Standby, text replication is required.
Asynchronous Request Functions

Just exactly why were Asynchronous Request Functions invented for anyway???

It is an even toss up as to which replication topic is least understood – text replication, Parallel DSI’s, or asynchronous request functions. Even for those who understand what they do, they don’t understand the impact that they could have on replication performance. In this section, we will be taking a close look at Asynchronous Request Functions and the performance implications of using them.

Purpose

During normal replication, it is impossible for a replicated data item to be re-replicated back to the sender or sent on to other sites (without the old LTM “–A” mode or the current send_maint_xacts_to_replicate configuration for Replication Agent). However, in some cases this might be necessary. There are many real-life scenarios in which a business unit needs to submit a request to another system and have the results replicated back. While it is always possible to have the first system simply execute a stored procedure that is empty of code as a crude form of messaging, the problem with this is that the results are not replicated back to the sender. The reason is simple – the procedure would be executed at the target by the maintenance user – whose transactions are filtered out. It is also possible to configure the replication agent to not filter out the maintenance user, but that could lead to the “endless loop” replication problem. Since we are discussing it, the obvious solution is asynchronous request functions. Sometimes, however, it might not be the obvious answer as it can get overlooked. In the next couple of sections, we discuss several scenarios of real-life situations in which asynchronous request functions make sense.

Key Concept #28: Asynchronous Request Functions were intended for a replicate system to be able to asynchronously request the primary perform some changes and then re-replicate those changes back to the replicate

Web – Internal Requests

Let’s assume we are working for a large commercial institution such as a bank or a telephone utility company. As part of our customer service (and to stay competitive), we have created a web site for our customers to view online billing/account statements or whatever. However, to protect our main business systems from the ever-present hackers and to ensure adequate performance for internal processes, we have separated the web-supported database from the database used by internal applications (a very, very good idea that is rarely implemented). In addition, to make this site work for us and to reduce the number of customer service calls handled by operators, we would like the customer to be able to change their basic account information (name, mailing address) as well as perform some basic operations (online bill pay, transfer funds). Sounds pretty normal right???

The problem with this is, how do you handle the name changes, etc.?? In some systems, you can’t – you have to provide a direct interface to the main business systems. However, with Replication Server, you simply implement each of the customer’s actions as “request functions”, in which the request for a name change, bill payment, whatever is forwarded to the main business system, processed and then the results replicated back. You could easily picture this as being something similar to:

![Figure 48 – Typical Web/Internal Systems Architecture](image)

In fact, the way most commercial bank web sites work, this architecture is extremely viable and reduce the risk to mission critical systems by isolating the main business systems from the load and security risks of web users.
Corporate Change Request

In many large systems, some form of corporate controlled data exists which can only be updated at the corporate site. A variation of this is a sort of change nomination process in which the change nomination is made to the headquarters and due to automated rules, the change is made. One example in which this applies is a budget programming system. As lower levels submit their budget requests, the corporate budget is reduced and the budgeted items replicated back to subscribing sites. At the headquarters system, rules such as whether or not the amount exceeds certain dollar thresholds based on the type of procurement etc. could be in place.

This scenario is a bit different than most as the local database would not be strictly executing a request function. More than likely, a “local change” would be enacted – i.e. a record saved in the database with a “proposed” status. Once the replicated record is received back from headquarters, it simply overwrites the existing record. In addition, due to the hierarchical nature of most companies, a request from a field office for a substantial funding item may have to forwarded through intermediates – in affect, the request function is replicated on to other more senior organizations due to approval authority rules.

Update Anywhere

Whoa!!! This isn’t supposed to be able to be done with Sybase Replication Server. For years we have been taught the sanctity of data ownership and woe to the fool who dared to violate those sacred rules as they would be forever cursed with inconsistent databases.

Not. Consider the fact that you and your spouse are both at work…only you happen to be traveling out of the area. Now, picture a bad phone bill (or something similar) in which you both call to change the address, account names or something – but provide slightly different information (i.e. work phone number). The problem is that by being in two different locations and using the same toll-free number, you were probably routed to different call centers with (gasp) different data centers. The fledgling Sybase DBA answer is this can’t be done. However, keep in mind, that the goal is to have all of the databases consist – which of the two sets of data is the most accurate portrayal of the customer information is somewhat irrelevant. Having that in mind, look at the following architecture.
No matter what order request 1 or 2 occur in, the databases will all have the same answer. The reason? We are exploiting the commit sequence assurance of Replication Server. In this case, it is the commit sequence of the request functions at the “arbitrator”. If request #2 commits first, then it will get response A and request #1 will get response B. Since commit order is guaranteed via Replication Server, then every site will have the response (A) from request 2 applied ahead of the response (B) from request 1.

Implementation & Internals

Now that we have established some of the reasons why a business might want to do Asynchronous Request Functions, the next thing to consider is how they are implemented. Frequently, another reason administrators don’t implement request functions is the lack of understanding who to set it up. In this section, we will explore this and how the information gets to the replication server.

Replicate Database & Rep Agent

Perhaps before discussing what happens internally, a good idea might be to review the steps necessary to create an asynchronous request function.

Implementing Asynchronous Request Functions

In general, the steps are:

1. If not already established, make sure source database is established as a primary database for replication (i.e. has a Rep Agent, etc.)
2. Create the procedure to function as the asynchronous request function. This could be an “empty” procedure – or could have logic to perform “local” changes (i.e. set a status column to “pending”).
3. Mark the procedure for replication in the normal fashion (sp_setrepproc)
4. Create a replication definition for the procedure, specifying the primary database as the target (or recipient) desired and not the source database actually containing the procedure.
5. Make sure the login names and passwords are in synch between the servers for users who have permission to execute the procedure locally (including those who can perform DML operations if proc is embedded in a trigger).

6. Ensure that the common logins have permission to execute the procedure at the recipient database.

A bit of explanation might be in order for the last three. Regarding step #4, the typical process of replicating a procedure from a primary to a replicate involves creating a replication definition and subscription as normal similar to:

```
create function replication definition my_proc_name
with primary at HQ.funding
deliver as 'hq_my_proc_name'
(...param list...)
searchable parameters (...param list...)
```

At PRS
```
create subscription my_proc_name_sub
for my_proc_name
with replicate at NY.funding
```

At RRS
```
hq_my_proc_name
```

Figure 51 – Applied (Normal) Procedure Replication Definition Process

For request functions, the picture changes slightly to:

```
create function replication definition ny_my_proc_name
with primary at HQ.funding
deliver as 'ny_req_my_proc_name'
 (...param list...)
searchable parameters (...param list...)
```

At PRS
```
create subscription ny_my_proc_name_sub
for ny_my_proc_name
with replicate at NY.funding
(no subscription)
```

At RRS
```
y_my_proc_name
```

Figure 52 – Asynchronous Request Function Replication Definition Process

Note that in the above example, the “with primary at” clause specifies the recipient (HQ in this case) and not the source (NY) and that the replication definition was created at the primary PRS for the recipient. One way to think of it is that an asynchronous request function replication definition functions as both a replication definition and subscription.

A couple of points that many might not consider in implementing request functions:

- A single replicated database can submit request functions to any number of other replicated databases. Think of a shared primary configuration of 3 or more systems. Any one of the systems could send a request function to any of the others.

- While a single site can send request functions to any number of sites, a single request function can only be sent to a single recipient site. This restriction is due to the fact a single procedure needs to have a unique replication definition and that definition can only specify a single “with primary at” clause.

- In order to send a request function to another system, a route must exist between the two replicated systems.

**Replication Agent Processing**

Essentially, there is nothing unique about Replication Agent processing for request functions. As with any stored procedure execution, when a request function procedure is executed, an implicit transaction is begun. While described in general terms in the LTL table located in the Replication Agent section much earlier, the full LTL syntax for “begin transaction” is:

```
distribute begin transaction 'tran name' for 'username'/ - encrypted_password
```

Consequently, the username and encrypted password are packaged into the LTL for the Replication Server. The reason for this is as you probably guessed – the fact that the Replication Server executes the request function at the destination as the user who executed it at the primary (more on this in the next section). As a result, Replication Agent processing for request functions is identical to the processing for an applied function.
Replication Server Processing

Since the source database processing is identical to applied functions, it is within the Replication Server that all of the magic for request functions happens. This happens in two specific areas – the inbound processing and the DSI processing.

Inbound Processing

As discussed earlier, within the inbound processing of the replication server, not much happens as far as row evaluation until the DIST thread. Normally, this involves matching replicated rows with replication definitions, normalizing the columns and checking for subscriptions. In addition, for stored procedure replication definitions, this process also involves determining if the procedure is an applied or request function. Remember: the name for a replication definition for a procedure is the same as the procedure name, and that due to the unique naming constraint for replication definitions, there will only be one replication definition with the same name as the procedure. Consequently, determining if the procedure is a request function or not is easily achieved simply by checking to see if the primary database for the replication definition is the same as the current source connection (i.e. connection for which the SQM belongs to). If not, then the procedure is a request procedure. Following the SQM, the DIST/SRE fails to find a subscription and simply needs to read the primary at clause to determine the “primary” database that is intended to receive the request function. The DIST/SRE then writes the request function to the outbound queue, marking it as a request function.

DSI Processing

Within the outbound queue processing of a request function, the only difference is in the DSI processing. When a request function is processed by a DSI, the following occurs:

- The DSI-S stops batching commands and submits all commands up to the request function.
- The DSI-E disconnects from the replicate dataserver and reconnects as the username and password from the request function transaction record.
- The DSI-E executes the request function. If more than one request function has been executed in a row by the same user, all are executed individually.
- The DSI-E disconnects from the replicate and reconnects as either the maintenance user or different user. The latter is applicable when back-to-back request functions are executed by different users at the primary.

Once the request function(s) have been delivered, the DSI resumes “normal” processing of transactions as the maintenance user until the next request function is encountered.

Recipient Database Processing

The second difference in request function processing takes place at the replicate database. If you remember from our earlier discussion, the Replication Agent filters log records based on the maintenance user name returned from the LTL “get maintenance user” command. Since the DSI applies the request function by logging in as the same user at the primary, then any modification performed by the request function execution is eligible for replication back out of the recipient database. If the procedure listed in the “deliver as” clause of the request function replication definition is itself marked for replication, then the procedure invoked by the request function will be replicated as an applied function. If not, then any individual DML statements on tables marked for replication and/or sub-procedures marked for replication will be replicated as normal. A couple of points for consideration:

- The destination of the modifications be replicated out of the recipient is not limited to the site that originally made the request function call. Since at this point normal replication processing is in effect, normal subscription resolution specifies which sites receive the modifications due to the request function.
- The “deliver as” procedure itself (or a sub-procedure) could be a request function in which case the request is “forwarded up the chain” while the original request function serves as “notification” to the immediate supervisory site that the subordinate is making a request.
Key Concept #29: An Asynchronous Request Function will be executed at the recipient by the same user/password combination as the procedure was executed by at the originating site. Because it is not executed by the maintenance user, changes made by the request function are then eligible for replication.

Performance Implications

By now, you have begun to realize some of the power – and possibilities – of request functions. However, they do have downside – it degrades replication performance. Consider the following:

- Replication command batching/transaction grouping is effectively terminated when a request function is encountered (largely due to the reconnection issue).

- Replication Server must first disconnect/reconnect as the request function user, establish the database context, execute the procedure, and then disconnect/reconnect as the maintenance user. Ignoring the procedure execution times, the two disconnect/reconnects could consume a considerable portion of time when a large number of request functions are involved.

- In the typical implementation, the request functions at the originator are often empty, while at the recipient there is a sequence of code. Consequently, at the originator, transactions that follow the request function appear to execute immediately. However, at the recipient, they will be delayed until the request function completes execution.

Normally the latter is not much of an issue, but some customers have attempted to use request functions as a means of implementing “replication on demand” in which a replicate periodically executes a request function that at the primary flips a “replicate_now” bit (or something similar). If the number of rows affected are very large, then this procedure’s execution could be significantly longer than expected.

In summary, request functions will impede replication performance by “interrupting” the efficient delivery of transactions. Obviously, the degree to which performance is degraded will depend on the number and frequency of the request functions. This should not deter Replication System Administrators from using request functions, however, as they provide a very neat solution to common business problems.
Multiple DSI’s

Multiple DSI or Parallel DSI – which is which or are they the same???

The answer to this question takes a bit of history. Prior to version 11.0, Parallel DSI’s were not available in Replication Server. However, many customers were already hitting the limit of Replication Server capabilities due to the single DSI thread. Accordingly, several different methods of implementing multiple DSI’s to the same connection were developed and implemented so widely that it was even taught in Sybase’s “Advanced Application Design Using Replication Server (MGT-700) course by late 1995 and early 1996.

This does not mean the two methods are similar as there is one very key difference between the two. Parallel DSI’s guarantee that the transactions at the replicate will be applied in the same order. Multiple DSI’s do not – in fact, exploit this to achieve higher throughput.

WARNING: Because the safeguards ensuring commit order are deliberately bypassed, Multiple DSI’s are not fully supported by Sybase Technical Support. If you experience product bugs such as stack traces, dropped LTL, etc., then Sybase Technical Support will be able to assist. However, if you experience data loss or inconsistency then Sybase Technical Support will not be able to assist in troubleshooting.

Concepts & Terminology

Okay, if you’ve read this far, then the above warning didn’t deter you. Before discussing Multiple DSI’s, however, a bit of terminology and concepts need to be discussed to ensure we each understand what is trying to be stated. Throughout the rest of this section, the following definitions are used in association with the following terms:

- **Parallel DSI** – Internal implementation present in the Replication Server product that uses more than one DSI thread to apply replicated transactions. Transaction commit order is still guaranteed, despite number of threads or serialization method chosen.

- **Multiple DSI** – A custom implementation in which multiple physical connections are created to the same database, in effect implementing more than one DSI thread. Transaction commit order is not guaranteed and must be controlled by design.

- **Serialized Transactions** – Transactions that must be applied in the same order to guarantee the same database result and business integrity. For example, a deposit followed by a withdrawal. Apply these in the opposite order may not yield the same database result as the withdrawal will probably be rejected due to a lack of sufficient funds.

- **Commit Consistent** – Transactions applied in any order will always yield the same results. For example transactions at different Point-Of-Sale (POS) checkout counters or transactions originating from different field locations viewed from the corporate rollup perspective.

**Key Concept #30:** If using the Multiple DSI approach, you must ensure that your transactions are “commit consistent” or employ your own synchronization mechanism to enforce proper serialization when necessary.

Performance Benefits

Needless to say, Multiple DSI’s can achieve several orders of magnitude higher throughput than Parallel DSI’s. One customer processing credit card transactions reported achieving 10,000,000 transactions per hour. If you think this is unrealistic, in late 1995, a U.S. Government monitored test demonstrated a single Replication Server (version 10.5) replicating 4,000,000 transactions per 24 hour period to three destinations – each transaction a stored procedure with typical embedded selects and averaging 10 write operations (40,000,000 write operations total) against SQL Server 10.0 with only 5 DSI’s. That’s a total of 12,000,000 replicated procedures for a total of 120,000,000 write operations processed by a single RS in a single day against a database engine with known performance problem!!! So 10,000,000 a hour with RS 11.x is could be believable. Such exuberance however needs to be tempered with the cold reality that in
order to achieve this performance, a number of design changes had to be made to facilitate the parallelism and extensive application testing to ensure commit consistency had to be done. It cannot be understated – Multiple DSI’s can be a lot of work – you have to do the thinking the Replication Server Engineering has done for you with Parallel DSI’s.

In order to best understand the performance benefits of Multiple DSI’s over Parallel DSI’s, you need to look at each of the bottlenecks that exist in Parallel DSI’s and see how Multiple DSI’s overcome them. While the details will be discussed in greater detail later, the performance benefits from Multiple DSI’s stem from the following:

- **No Commit Order Enforcement** – by itself, this is the source of the biggest performance boost as transactions in the outbound queue are not delayed due to long running transactions (i.e. remember the 4 hour procedure execution example) or just simply waiting for their “turn” to commit.

- **Not Limited to a Single Replication Server** – Multiple DSI’s lends itself extremely well to involving multiple Replication Servers in the process – achieving an MP configuration currently not available within the product itself.

- **Independent of Failures** – If a transaction fails with Parallel DSI, activity halts – even if the transactions that follow it have no dependence on the transaction that failed (i.e. corporate rollups). As a consequence, Multiple DSI’s prevent large backlogs in the outbound queue reducing recovery time from transaction failures.

- **Cross-Domain Replication** – Parallel DSI’s are limited to replicating to destinations within the same Replication domain as the primary. Multiple DSI’s have no such restriction and in fact, extend easily to support large-scale cross-domain replication architectures (different topic outside scope of this paper).

### Implementation

While the Sybase Education course MGT-700 taught at least three methods for implementing Multiple DSI’s, including altering the system function strings, the method discussed in this section will focus on that of using multiple maintenance users. The reason for this is the ease and speed of setup and the least impact on existing function definitions (i.e. you don’t end up creating a new function class). Implementing Multiple DSI’s is a sequence of steps:

1. Implementing multiple physical connections
2. Ensuring recoverability and preventing loss
3. Defining and implementing parallelism controls

#### Implementing multiple physical connections

The multiple DSI approach uses independent DSI connections for delivery. Due to the unique index on the rs_databases table in the RSSD, the only way to accomplish this is to fake out the Replication Server and make it think it is actually connecting to multiple databases instead of one. Fortunately, this is easy to do. Since Replication Server doesn’t check the name of the server it connects to, all we need to do is “alias” the real dataserver in the Replication Server’s interfaces file. For example, lets assume we have a interfaces file similar to the following (Solaris):

```
CORP_FINANCES
master tli /dev/tcp \x00224b782f650950000000000000000000
query tli /dev/tcp \x00224b782f650950000000000000000000
```

Based on our initial design specifications, we decide we need a total of 6 Multiple DSI connections. Given that the first one counts as one, we simply need to alias it five additional times.

```
CORP_FINANCES
master tli /dev/tcp \x00224b782f650950000000000000000000
query tli /dev/tcp \x00224b782f650950000000000000000000

CORP_FINANCES_A
master tli /dev/tcp \x00224b782f65095000000000000000000000
query tli /dev/tcp \x00224b782f65095000000000000000000000

CORP_FINANCES_B
master tli /dev/tcp \x00224b782f65095000000000000000000000
query tli /dev/tcp \x00224b782f65095000000000000000000000

CORP_FINANCES_C
master tli /dev/tcp \x00224b782f65095000000000000000000000
```
Once this is complete, the Multiple DSI’s can simply be created by creating normal replication connections to CORP_FINANCES.finance_db, CORP_FINANCES_A.finance_db, CORP_FINANCES_B.finance_db, etc. However, before we do this, there is some addition work we will need to do to ensure recoverability (discussed in next section).

To get a clearer picture of what this accomplishes, however, as we mentioned Replication Server now thinks it is replicating to \( n \) different replicate databases instead of one. Because of this, it creates separate outbound queues and DSI threads to process each connection. The difference between this and Parallel DSI’s is illustrated in the following diagrams.

In the above drawings, only a single replication server was demonstrated. However, in Multiple DSI’s each of the connections could be from a different replication server. Consider the following – the first being the more normal multiple replication server implementation using routing to a single replication server, while the second demonstrates Multiple DSI’s - one from each Replication Server.
While the RRS could use Parallel DSI’s, as we have already discussed, long transactions or other issues could degrade performance. In addition, only a single RSI thread is available between the two Replication Servers involved in the routing. While this is normally sufficient, if a large number of large transactions or text replication is involved, it may also be a bottleneck. Additionally, this has an inherent fault in that if any one of the transactions from any of the source sites fail, all of the sites stop replicating until the transaction is fixed and the DSI is resumed.

In contrast, consider a possible Multiple DSI implementation:

In this case, each RS could still use Parallel DSI’s to overcome performance issues within each and in addition, since they are independent, a failure of one does not cause the others to backlog.

A slight twist of the latter ends up with a picture that demonstrates the ability of Multiple DSI’s to provide a multi-processor (MP) implementation.
Investments
Trading System
Investments
Trading System

Figure 57 – MP Replication Achieved via Multiple DSI’s

Note that the above architecture really only helps the outbound processing performance. All subscription resolution, replication definition normalization, etc. is still performed by the single replication server servicing the inbound queue. However, systems with high queue writes, extensive function string utilization or other requirements demonstrating a bottleneck in the outbound processing, the MP approach may be viable.

Ensuring Recoverability and Preventing Loss

While the multiple independent connections do provide a lot more flexibility and performance, they do present a problem – recoverability. The problem is simply this: with a single rs_lastcommit table and commit order guaranteed, Parallel DSI’s are assured at restarting from that point and not incurring any lost or duplicate transactions. However, if using Multiple DSI’s, the same is not true. Simply because the last record in the rs_lastcommit table refers to transaction id 101 does not mean the transaction 100 was applied successfully – or that 102 has not been already applied. Consider the following picture:

Plausible Scenarios:

1 - c committed after a, b, & d (long xactn)
2 - a, b, d suspended first
3 - a, b, d rolled back due to deadlocks

Figure 58 – Multiple DSI’s with Single rs_lastcommit Table

Consider the three scenarios proposed above. In each of the three, you would have no certainty that tran OQID 42 should be next. As a result, it is critical that each Multiple DSI has its own independent set of rs_lastcommit, rs_thread tables as well as associated procedures (rs_update_lastcommit).

Unfortunately, a DSI connection does not identify itself, consequently there are only two choices available:

1. Use a separate function class for each DSI. Within the class, call altered definitions of rs_update_lastcommit to provide distinguishable identity. For example, add a parameter that is hard-coded to the DSI connection (i.e. “A”), or call a variant of the procedure such as rs_update_lastcommit_A.
2. Exploit the ASE permission chain and use separate maintenance users for each DSI. Then create separate rs_lastcommit, etc. owned by each specific maintenance user.

3. Multiple maintenance users with changes to the rs_lastcommit table to accommodate connection information and corresponding logic added to rs_update_lastcommit to set column value based on username.

While the first one is obvious – and obviously a lot of work as maintaining function strings for individual objects could then become a burden, the second takes a bit of explanation. The third one is definitely an option and is perhaps the easiest to implement. The problem is that with high volume replication, the single rs_lastcommit table could easily become a source of contention. In addition to rs_lastcommit, a column would have to be added to rs_threads as it has no distinguishable value either – along with changes to the procedures which manipulate these tables (rs_update_lastcommit, rs_get_thread_seq, etc.). However, it does have the advantage of being able to handle identity columns and other maintenance user actions requiring “dbo” permissions. While separate maintenance user logins are in fact used, each are aliased as dbo within the database. The modifications to the rs_lastcommit and rs_threads tables (and their corresponding procedures such as rs_update_lastcommit, rs_get_lastcommit, etc.) would be to add a login name column. Since this is system information available through suser_name() function, the procedure modifications would simply be adding the suser_name() function to the where clause. For example, the original rs_lastcommit table, rs_get_lastcommit and rs_update_lastcommit are as follows:

```sql
-- Drop the table, if it exists. */
if exists (select name
    from sysobjects
    where name = 'rs_lastcommit' and type = 'U')
begin
    drop table rs_lastcommit
end
go

/* Create the table.
** We pad each row to be greater than a half page but less than one page
** to avoid lock contention.
*/
create table rs_lastcommit
(
orign int,
origin_qid binary(36),
secondary_qid binary(36),
orign_time datetime,
dest_commit_time datetime,
pad1 binary(255),
pad2 binary(255),
pad3 binary(255),
pad4 binary(255),
pad5 binary(4),
pad6 binary(4),
pad7 binary(4),
pad8 binary(4)
)
go
create unique clustered index rs_lastcommit_idx on rs_lastcommit(origin)
go

/* Drop the procedure to update the table. */
if exists (select name
    from sysobjects
    where name = 'rs_update_lastcommit' and type = 'P')
begin
    drop procedure rs_update_lastcommit
end
go
/* Create the procedure to update the table. */
create procedure rs_update_lastcommit
@orign int,
@origin_qid binary(36),
@secondary_qid binary(36),
orign_time datetime
as
update rs_lastcommit
    set origin_qid = @origin_qid, secondary_qid = @secondary_qid,
orign_time = @orign_time,
dest_commit_time = getdate()
    where origin = @orign
if (@@rowcount = 0)
```
begin
    insert rs_lastcommit (origin, origin_qid, secondary_qid, origin_time, dest_commit_time, pad1, pad2, pad3, pad4, pad5, pad6, pad7, pad8)
    values (@origin, @origin_qid, @secondary_qid, @origin_time, getdate(), 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)
end

go
/* Drop the procedure to get the last commit. */
if exists (select name
    from sysobjects
    where name = 'rs_get_lastcommit' and type = 'P')
begin
    drop procedure rs_get_lastcommit
end

go
/* Create the procedure to get the last commit for all origins. */
cREATE PROCEDURE rs_get_lastcommit
as
    select origin, origin_qid, secondary_qid
    from rs_lastcommit

go
Note that the last procedure, rs_get_lastcommit, normally retrieves all of the rows in the rs_lastcommit table. The reason for this is that the oqid is unique to the source system – but if there are multiple sources as can occur in a corporate rollup scenario – there may be duplicate oqid’s. Consequently, the oqid and database origin id (from RSSD..rs_databases) is stored together. During recovery, as each transaction is played back, the oqid and origin are used to determine if the row is a duplicate.

If using the multiple login/altered rs_lastcommit approach, then you simply need to add a where clause to each of the above procedures and the primary key/index constraints. For rs_lastcommit, this becomes (modifications highlighted):

/* Drop the table, if it exists. */
if exists (select name
    from sysobjects
    where name = 'rs_lastcommit' and type = 'U')
begin
    drop table rs_lastcommit
end

go
/* ** Create the table. **
*** We pad each row to be greater than a half page but less than one page
*** to avoid lock contention. *
*/
-- modify the table to add the maintenance user column.
cREATE TABLE rs_lastcommit
(
    maint_user varchar(30),
    origin int,
    origin_qid binary(36),
    secondary_qid binary(36),
    origin_time datetime,
    dest_commit_time datetime,
    pad1 binary(255),
    pad2 binary(255),
    pad3 binary(255),
    pad4 binary(255),
    pad5 binary(4),
    pad6 binary(4),
    pad7 binary(4),
    pad8 binary(4)
)

go
-- modify the unique index to include the maintenance user
CREATE UNIQUE CLUSTERED INDEX rs_lastcommit_idx ON rs_lastcommit (maint_user, origin)
go

/* Drop the procedure to update the table. */
if exists (select name
    from sysobjects
    where name = 'rs_update_lastcommit' and type = 'P')
begin

/* Create the procedure to update the table. */
create procedure rs_update_lastcommit
@origin int,
@origin_qid binary(36),
@secondary_qid binary(36),
@origin_time datetime
as
-- add maint_user qualification to the where clause.
update rs_lastcommit
set origin_qid = @origin_qid, secondary_qid = @secondary_qid,
origin_time = @origin_time,
dest_commit_time = getdate()
where origin = @origin
and maint_user=suser_name()
if (@@rowcount = 0)
begin
-- add the maintenance user login to insert statement
insert rs_lastcommit (maint_user, origin, origin_qid, secondary_qid,
origin_time, dest_commit_time,
pad1, pad2, pad3, pad4, pad5, pad6, pad7, pad8)
values (suser_name(), @origin, @origin_qid, @secondary_qid,
@origin_time, getdate(),
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)
end
/* Drop the procedure to get the last commit. */
if exists (select name
from sysobjects
where name = 'rs_get_lastcommit' and type = 'P')
begin
drop procedure rs_get_lastcommit
end
go
/* Create the procedure to get the last commit for all origins. */
create procedure rs_get_lastcommit
as
-- add maint_user to the (previously nonexistent) where clause
select origin, origin_qid, secondary_qid
from rs_lastcommit
where maint_user = suser_name()
go

Similar changes will need to be done to the rs_threads table and associated procedure calls as well. It is important to avoid changing the procedure parameters. Fortunately, all retrieval and write operations against the rs_lastcommit table are performed through stored procedure call (similar to an API of sorts). By not changing the procedure parameters and due to the fact that all operations occur through the procedures, we do not need to make any changes to the function strings (reducing maintenance considerably). Why this is necessary at all is discussed under section describing the Multiple DSI/Multiple User implementation.

Note that at the same time, we could alter the table definition to accommodate max_rows_per_page or datarow locking and eliminate the row padding (thereby reducing the amount of data logged in the transaction log for rs_lastcommit updates). However, other than the reduction in transaction log activity, this will gain little in the way of performance. It is a useful technique to remember, though, as ASE 12.5 will support larger page sizes (i.e. 16KB vs. 2KB), which invalidates the normal rs_lastcommit padding. So if implementing RS 12.1 or less on ASE 12.5 you may need to modify these tables anyhow.

While useful for handling identity and simple to implement, the third alternative above may provide slightly greater performance by eliminating any contention on the rs_lastcommit table. By using separate maintenance users, you can exploit the way ASE does object resolution and permission checking. It is a little known fact (but still documented), that when you execute a SQL statement in which the object’s ownership is not qualified, ASE will first look for an object of that name owned by the user (as defined in sysusers). If one is not found, then it searches for one owned by the database owner – dbo. So if “fred” is a user in the database and there is two tables: 1) fred.authors; and 2) dbo.authors, and fred issues “select * from pubs2..authors”, authors will be resolved to fred.authors. On the other hand, if Mary issues “select * from pubs2..authors”, since no mary.authors exists, authors will be resolved to dbo.authors. Consequently, by using separate maintenance users and individually owned rs_lastcommit, etc. tables, we have the following:
This then addresses the problems in the scenario we discussed earlier and changes the situation to the following:

**Figure 59 – Multiple Maintenance Users with Individual rs_lastcommits**

**Plausible Scenarios:**

1. c committed after a, b, & d (long xactn)
2. a, b, d suspended first
3. a, b, d rolled back due to deadlocks

**Figure 60 – Multiple DSI’s with Multiple rs_lastcommit tables**

Now, no matter what the problem, each of the DSI’s recovers to the point where it left off.

**Key Concept #31:** The Multiple DSI approach uses independent DSI connections set up via aliasing the target dataserver.database. However, this leads to a potential recoverability issue with RS system tables that must be handled to prevent data loss or duplicate transactions.
Detailed Instructions for Creating Connections

Now that we now what we need to do to implement the multiple DSI’s and how to ensure recoverability, the next stage is to determine exactly how to achieve it. Basically, it comes down to a modified rs_init approach or performing the steps manually (as may be required for heterogeneous or OpenServer replication support). Each of the below requires the developer to first create the aliases in the interfaces file.

**Manual Multiple DSI Creation**

Despite what it sounds, the manual method is fairly easy, but does require a bit more knowledge about Replication Server. The steps are:

1. Add the maintenance user logins (sp_addlogin). Create as many as you expect to have Multiple DSI’s plus a few extra.
2. Grant maintenance user logins replication_role. Do not give them sa_role. If you do, when in any database, the maintenance user will map to “dbo” user vs. the maintenance user desired – consequently incurring the problem with rs_lastcommit.
3. Add the maintenance users to the replicated database. If identity values are used, one may have to be aliased to “dbo”. If following the first implementation (modifying rs_lastcommit), all may be aliased to dbo.
4. Grant all permissions on tables/procedures to replication_role. While you could grant permissions to individual maintenance users, by granting permissions to the role, you reduce the work necessary to add additional DSI connections later.
5. Make a copy of $SYBASE/$SYBASE_RS/scripts/rs_install_primary. Alter the copy to include the first maintenance user as owner of all the objects. Use isql to load the script into the replicate database. Repeat for each maintenance user.
6. Create connections from Replication Server to the replicate database. If the database will also be a primary database and data is being replicated back out, pick one of the maintenance users to be the “maintenance user” and specify the log transfer option

   ```
   create connection to data_server.database
   set error class [to] rs_sqlserver_error_class
   set function string class [to] rs_sqlserver_function_classset username [to] maint_user_name
   [set password [to] maint_user_password ]
   [set database_param [to] 'value']
   [set security_param [to] 'value']
   [with [log transfer on, dsi_suspended]]
   [as active for logical_ds.logical_db | as standby for logical_ds.logical_db [use dump marker]]
   ```

7. If replicate is also a primary, add the maintenance user to Replication Server (create user) grant the specified maintenance user connect source permission in the Replication Server. For all other maintenance users, alter the connection and set replication off (if desired).
8. Configure the Replication Agent as desired.

**Modified rs_init Method**

The modified rs_init method is the easiest and ensures that all steps are completed (none are accidentally forgotten). It is very similar to the above in results, but less manual steps.

1. Make a copy of $SYBASE/$SYBASE_RS/scripts/rs_install_primary (save it as rs_install_primary_orig). Alter the rs_install_primary to include the first maintenance user as owner of all the objects.
2. Run rs_init for replicate database. Specify the first maintenance user. Repeat steps 1-2 until all maintenance users created. If using the modified rs_lastcommit approach, you can simply repeat step 2 until done.
3. If identity values are used, one may have to be aliased to “dbo” (drop the user and add an alias).
4. (Same as above). Grant all permissions on tables/procedures to replication_role. While you could grant permissions to individual maintenance users, by granting permissions to the role, you reduce the work necessary to add additional DSI connections later.
5. Use `sp_config_rep_agent` to specify the desired maintenance user name and password for the Replication Agent. Not that all maintenance users have probably been created as Replication Server users. This is not a problem, but can be cleaned up if desired.

6. Rename the `rs_install_primary` script to a name such as `rs_install_primary_mdsi`. Rename the original back to `rs_install_primary`. This will prevent problems for future replication installations not involving multiple DSI’s.

**Single rs_lastcommit with Multiple Maintenance Users**

If for maintenance reasons or other, you opt not to have multiple rs_lastcommit tables and instead wish to use a single table, you will have to do the following (note this is a variance to either of the above, so replace the above instructions as appropriate):

1. Make a copy of `rs_install_primary`. Depending on manual or `rs_init` method, edit the appropriate file and make the following changes:
   a. Add column for maintenance user `suid()` or `suser_name()` to all tables and procedure logic. This includes adding column to tables such as `rs_threads` without anything. Procedure logic should select `suid()` or `suser_name()` for use as column values.
   b. Adjust all unique indexes to include `suid()` or `suser_name()` column.

    2. Load script according to applicable manual or `rs_init` instructions above.

**Single rs_lastcommit with Single Maintenance User**

This method employs the use of function string modifications and really is only necessary if the developers really want job security due to maintaining function strings. The steps are basically:

1. Make a copy of `rs_install_primary` and save it as `rs_install_primary_orig`. Modify the original as follows:
   a. Add column for DSI to each table as well as parameter to each procedure. This includes tables such as `rs_threads`, `rs_lastcommit` and their associated procedures.
   b. Adjust all unique indexes to include DSI column.

    2. Load script using `rs_init` as normal. This will create the first connection.

    3. Create a function string class for the first DSI (inherit from default). Modify the system functions for `rs_get_thread_seq`, `rs_update_lastcommit`, etc. to specify the DSI. Repeat for each DSI.

    4. Alter the first connection to use the first DSI’s function string class.

    5. Create multiple connections from Replication Server to replicate database for remaining DSI’s using the create connection command. Specify the appropriate function string class for each.

    6. Rename the `rs_install_primary` script to a name such as `rs_install_primary_mdsi`. Rename the original back to `rs_install_primary`. This will prevent problems for future replication installations not involving multiple DSI’s.

    7. Monitor replication definition changes during lifecycle. Manually adjust function strings if inheritance does not provide appropriate support.

**Defining and Implementing Parallelism Controls**

The biggest challenge to Multiple DSI’s is to design and implement the parallelism controls in such a way that database consistency is not compromised. The main mechanism for implementing parallelism is through the use of subscriptions, and in particular the subscription where clause. Each aliased database connection (Multiple DSI) subscribes to a different data – either at the object level or through the where clause. As a result, two transactions executed at the primary might be subscribed to by different connections and therefore have a different order of execution at the replicate than they had at the primary. The following rules **must** be followed to ensure database consistency:

1. Parallel transactions **must** be commit consistent.

   2. Serial transactions **must** use the same DSI connection.
3. If not 1 & 2, you must implement your own synchronization point to enforce serialization.

**Parallel Subscription Mechanism.**

In many cases, this is not as difficult to achieve as you would think. The key, however, is to make sure that the where clause operations for any one connection are mutually exclusive from every other connection. This can be done via a variety of mechanisms, but is usually determined by two aspects: 1) the number of source systems involved; and 2) the business transaction model.

**Single Primary Source**

In some cases, a single primary source database provides the bulk of the transactions to the replicate. As a result, it is the transactions from this source database that must be processed in parallel using the Multiple DSI’s. In this situation, each of the Multiple DSI’s subscribes to different transactions or different data through one of the following mechanisms:

- **Data Grouping** – In this scenario, different DSI’s subscribe to a different subset of tables. This is most useful when a single database is used to process several different types of transactions. The transactions affect a certain small number of tables unique to that data. An example of this might be a consolidated database in which multiple stations in a business flow all access the same database. For example, a hospital’s outpatient system may have a separate appointment scheduling/check-in desk, triage treatment, lab tests and results, pharmacy, etc. If each “group” of tables that support these functions are subscribed to by different DSI’s, they will be applied in parallel at the replicate.

- **Data Partitioning** – In this scenario, different DSI’s subscribe to different sets of data from the same tables, typically via a range or discrete list. An example of the former may be that a DSI may subscribe to A-E or account numbers 10000-20000. An example of a discrete list might be similar to a bank in which one DSI subscribes to checking accounts, the other credit card transactions, etc.

- **User/Process Partitioning** – In this scenario, different DSI’s subscribe to data modified by different users. This is most useful in situations where individual user transactions need to be serialized, but are independent of each other’s. Probably one of the more frequently implementable, this includes situations such as retail POS terminals, banking applications, etc.

- **Transaction Partitioning** – In this scenario, different DSI’s subscribe to different transactions. Typically implemented in situations involving a lot of procedure-based replication, this allows long batch processes (i.e. interest calculations) to execute independent of other batch processes without either “blocking” the other through the rs_threads issue.

The first two and last are fairly easy to implement and typically do not require modification to existing tables. However, the user/process partition might. If the database design incorporates an audit function to record the last user to modify a record and user logins are enforced, then such a column could readily be used as well.

However, in today’s architectures, frequently users are coming through a middleware tier (such as a web or app server) and are using a common login. As a result, a column may have to be added to the main transaction tables to hold the process id (spid) or similar value. In many cases, the spid itself could be hard to develop a range on as load imbalance and range division may be difficult to achieve. For example, a normal call center may start with only a few users at 7:00am, build to 700 concurrent users by 09:00am and then degrade slowly to a trickle from 4:00pm to 06:00pm. If you tried to divide the range of users evenly by spid, you would end up with some DSI’s not doing any work for a considerable period (4 hours) of the workday. On the other hand, the column could store the mod() of the spid (i.e. @@spid%10) – remembering that the result of mod(n) could be zero through n-1 (i.e. mod(2) yields 0 & 1 as remainders). Note that as of ASE 11.9, global variables are no longer allowed as input parameter defaults to stored procedures.

**Multiple Primary Sources**

Multiple primary source system situations are extremely common to distributed businesses needing a corporate rollup model. Each of the regional offices would have it’s own dedicated DSI thread to apply transactions to the corporate database. As mentioned earlier, this has one very distinct advantage over normal replication in that an erroneous transaction from one does not stop replication from all the others by suspending the DSI connection. When multiple primary source systems are present, establishing parallel transactions are fairly easy due to the following:
No code/table modifications - Since each source database has its own dedicated DSI, from a replication standpoint, it resembles a 1:1 straightforward replication.

Guaranteed commit consistency - Transactions from one source system are guaranteed commit consistent from all others. This is true even in cases of two-phased commit distributed transactions affecting several of the sources. Since in each case an independent Rep Agent, inbound queue processing and oqid’s are used for the individual components of a 2PC transaction, it would be impossible for even a single Replication Server to reconstruct the transaction into a single transaction for application at the replicate.

Parallel DSI support – While this doesn’t appear to add benefit if the multiple DSI’s are from a single source, in the case of multiple sources, it can help with large transactions (due to large transaction threads) and medium volume situations through tuning the serialization method (none vs. wait_for_commit), etc.

Handling Serialized Transactions

In single source systems, it is frequent that a small number of transactions still need to be serialized no matter what the parallelism strategy you choose. For example, if a bank opts for using the account number, probably 80-90% of the transactions are fine. However, in the remaining 10-20% are transactions such as account transfers that need to be serialized. For example, if a typical customer transfers funds from a savings to a checking account, if the transaction is split due to the account numbers, the replicate system may be inconsistent for a period of time. While this may not affect some business rules, if an accurate picture of fund balances is necessary, this could cause a problem similar to the typical isolation level 3/phantom read problems in normal databases. Consequently, after defining the parallelism strategy, a careful review of business transactions needs to be conducted to determine which ones need to be serialized.

Once determined, the handling of serialized transactions is pretty simple – simply call a replicated procedure with the parameters. While this may necessitate an application change to call the procedure vs. sending a SQL statement, the benefits in performance at the primary are well worth it. In addition, because it is a replicated procedure, the individual row modifications are not replicated – consequently, the Multiple DSI’s that subscribe to those accounts do not receive the change. Instead, another DSI reserved for serialized transactions (it may be more than one DSI – depending on design) subscribes to the procedure replication and delivers the proc to the replicate.

The above is a true serialized transaction example. For the most part, serializing the transactions simply means ensuring that all the ones related are forced to use the same DSI. At that stage, the normal Replication Server commit order guarantee ensures that the transactions are serialized within respect one another. The most common example is to have transactions executed by the same user serialized – or impacting the same account serialized. For example, a hospital bill containing billable items for Anesthesia and X-ray. As long as the bill invoice number is part of the subscription and the itemization, then by subscribing by invoice, the transaction is guaranteed to arrive at the replicate as a complete bill – and within a single transaction.

However, there may not be a single or easily distinguishable set of attributes that can be easily subscribed to for ensuring transaction serialization within the same transaction. If such is the case, then the rs_id column becomes very useful. During processing, the primary database can simply assign an arbitrary transaction number (up to 2 billion before rollover) and store it in a column added similar to the user/spid mod() column described earlier. By using bitmask subscription, the load could be evenly balanced across the available Multiple DSI’s.

Serialization Synchronization Point

There may be times when it is impossible to use a single procedure call to replicate a transaction that requires serialization and the normal parallel DSI serialization is counter to the transactions requirements. This normally occurs when a logical unit of work is split into multiple physical transactions – possibly even executed by several different users. A classic case – without even parallel DSI - is when the transaction involves a worktable in one database and then a transaction in another database (pending/approved workflow). Another example, a store procedure at the primary call may generate a work table in one database using a select/into and then call a sub-procedure to further process and insert the rows. Of course, since both transactions originate from two different databases, read by two different Rep Agents, and delivered by two different DSI connections, the normal transactional integrity of the transaction is inescapably lost. Similarly, even when user/process id is used for the parallelism strategy, Multiple DSI connections will wreak havoc on transactional integrity and serialization – simply because there is no way to guarantee that the transaction from once connection will always arrive after the other.
The answer is “Yes”. The question “Is there a way to ensure transactions are serialized?” However, the technique is a bit reminiscent of rs_threads. If you remember, rs_threads imposes a modified “dead man’s latch” to control commit order. A similar mechanism could be constructed to the same thing through the use of stored procedures or function string coding. The core logic would be:

**Latch Create** – Basically some way to ensure that the latch was clear to begin with. Unlike rs_threads where the sequence is predictable, in this case, it is not, consequently a new latch should be created for each serialized transaction

**Latch Wait** – In this case, the second and successive transactions if occurring ahead of the first transaction need to sense that the first transaction has not taken place and wait.

**Latch Set** – As each successive transaction begins execution, the transaction needs to set and lock the latch.

**Latch Block** – Once the previous transactions have begun, the following transactions need to block on the latch so that as soon as the previous transactions commit, they can begin immediately.

**Latch Release** – When completed, each successive transaction needs to clear its lock on the latch. The last transaction should destroy the latch by deleting the row.

This is fairly simple for two connections, but what if 3 or more are involved? Even more complicated, what if several had a specific sequence for commit? For example, lets consider the classic order entry system in which the following tables need to be updated in order: order_main, order_items, item_inventory, order_queue. Normally, of course, the best approach would be to simply invoke the parallelism based on the spid of the person entering the order. However, for some obscure reason, this site can’t do that – and want to divide the parallelism along table lines. So, we would expect 4 DSI’s to be involved – one for each of the tables. The answer is we would need a latch table and procedures similar to the following at the replicate:

```sql
-- latch table
create table order_latch_table (
    order_number int not null,
    latch_sequence int not null,
    constraint order_latch_PK primary key (order_number)
)
lock datarows
go

-- procedure to set/initialize order latch
create procedure create_order_latch @order_number int, @thread_num rs_id
as begin
    insert into order_latch_table values (@order_number, 0)
    return (0)
end
go

-- procedure to wait block and set latch
create procedure set_order_latch
    @order_number int,
    @thread_seq int,
    @thread_num rs_id
as begin
    declare @cntrow int
    select @cntrow=0
    -- make sure we are in a transaction so block holds
    if @@trancount = 0
        begin
            rollback transaction
            raiseerror 30000 "Procedure must be called from within a transaction"
            return(1)
        end
    -- wait until time to set latch
    while @cntrow=0
        begin
            waitfor delay "00:00:02"
            select @cntrow=count(*)
            from order_latch_table
            where order_number = @order_number
            and latch_sequence = @thread_seq -1
            at isolation read uncommitted
        end
    -- block on latch so follow-on execution begins immediately
    -- once previous commits
```
update order_latch_table
  set latch_sequence = @thread_seq
  where order_number = @order_number
-- the only way we got to here is if the latch update worked
-- otherwise, we'd still be blocked on previous update
-- In any case, that means we can exit this procedure and allow
-- the application to perform the serialized update
return (0)
end

go

-- procedure to clear order latch
create procedure destroy_order_latch @order_number int, @thread_num rs_id
as begin
  delete order_latch_table
  where order_number = @order_number
  return (0)
end

go

It is important to note that the procedure body above is for the replicate database. At the primary, the procedure will
more than likely have no code in the procedure body as there is no need to perform serialization at the primary
(transaction is already doing that). In addition, it is possible to combine the “create” and “set” procedures into a single
procedure that would first create the latch if it did not already exist.

The way this works is very simple - but does require the knowledge of which threads will be applying the transactions.
For example, consider the following pseudo-code example:

Begin transaction
  Insert into tableA
  Update tableB
  Insert into tableC
  Update table B
  Commit transaction

Now, assuming tables A-C will use DSI connections 1-3 and need to be applied in particular order (i.e. A inserts a new
financial transaction, while B updates the balance and C is the history table), the transaction at the primary could be
changed to:

Begin transaction
  Exec SRV_create_order_latch @order_num, 1
  Insert into tableA
  Exec SRV_set_order_latch @order_num, 1, 2
  Update into tableB
  Exec SRV_set_order_latch @order_num, 2, 3
  Insert into tableC
  Insert into tableC
  Exec SRV_set_order_latch @order_num, 3, 2
  Update into tableB
  Exec SRV_set_order_latch @order_num, 3, 2
  Insert into tableC
  Insert into tableC
  Exec SRV_destroy_order_latch @order_num, 1
  Commit transaction

Note that the SRV prefix on the procedures in the above is to allow the procedure replication definition to be unique vs.
other connections. The “deliver as” name would not be prefaced with the server extension. Also, note that the first “set
latch” is sent using the second DSI. If you think about it, this makes sense as the first statement doesn’t have to wait
for any order - it should proceed immediately. In addition, the procedure execution calls above could be placed in
triggers, reducing the modifications to application logic - although this would require the trigger to set the latch for the
next statement, changing the above to:

Begin transaction
  Insert into tableA
    Exec SRV_create_order_latch @order_num, 1
    Insert into tableA
  Exec SRV_set_order_latch @order_num, 1, 2
    Select @seg_num=sequence_num from order_latch_table
    where order_number = @order_num
  Exec SRV_set_order_latch @order_num, @seg_num, 2
  Update into tableB
    Select @seg_num=sequence_num from order_latch_table
    where order_number = @order_num
  Exec SRV_set_order_latch @order_num, @seg_num, 3
  Insert into tableC
    Select @seg_num=sequence_num from order_latch_table
    where order_number = @order_num
  Exec SRV_set_order_latch @order_num, @seg_num, 3
  Insert into tableC
    Select @seg_num=sequence_num from order_latch_table
  Exec SRV_set_order_latch @order_num, @seg_num, 3
  Insert into tableC
    Select @seg_num=sequence_num from order_latch_table
where order_number = @order_num
Exec SRV_set_order_latch $order_num, $seq_num, 2
Update into tableB
Select $seq_num=sequence_num from order_latch_table
where order_number = @order_num
Exec SRV_set_order_latch $order_num, $seq_num, 3
Commit transaction

In which the indented calls are initiated by the triggers on the previous operation. Note that the above also uses variables for passing the sequence. This is simply due to the fact that the trigger is generic and can’t tell what number of operations preceded it. As a result, the local version of the latch procedures would have to have some logic added to track the sequence number for the current order number and each “set latch” would have to simply add one to the number.

-- latch table
create table order_latch_table (
    order_number int not null,
    latch_sequence int not null,
    constraint order_latch_PK primary key (order_number)
)
lock datarows

-- procedure to set/initialize order latch
create procedure SRV_create_order_latch @order_number int, @thread_num rs_id
    as begin
    insert into order_latch_table values (@order_number, 1)
    return (0)
    end

-- procedure to wait block and set latch
create procedure SRV_set_order_latch
    @order_number int,
    @thread_seq int,
    @thread_num rs_id
    as begin
    update order_latch_table
    set latch_sequence = latch_sequence+1
    where order_number = @order_number
    end

-- procedure to clear order latch
create procedure SRV_destroy_order_latch @order_number int, @thread_num rs_id
    as begin
    delete order_latch_table
    where order_number = @order_number
    return (0)
    end

However, you should also note that the destroy procedure never gets called - it would be impossible from a trigger to know when the transaction is ended. A modification to the replicate versions of rs_lastcommit procedure could perform the clean up at the end of each batch of transactions.

Design/Implementation Issues

In addition to requiring manual implementation for synchronization points, implementing multiple DSI’s has other design challenges.

Multiple DSI’s & Contention

Because Multiple DSI’s mimic the Parallel DSI serialization method “none”, they could experience considerable contention between the different connections. However, unlike Parallel DSI’s - the retry from deadlocking is not the “kindler-gentler” approach of applying the offending transactions in serial and printing a warning. Instead, they one that was rolled back (in this case the order (i.e. thread 2 vs. thread 1) is not known, so the wrong victim may be rolled back and the transaction attempted again and again until the DSI suspends due to exceeding the retries. For example, in a 1995 case study using 5 Multiple DSI connections for a combined 200 tps rate, 30% of the transactions deadlocked at the replicate. Of course, in those days, the number of transactions per group was not controllable and attempts to use
the byte size were rather cumbersome. In the final implementation, transaction grouping was simply disabled and the additional I/O cost of rs_lastcommit endured.

As a result, it is even more critical to tune the connections similar to the Parallel DSI/ dsi_serialization_method=none techniques discussed earlier. Namely:

- Set dsi_max_xacts_in_group to a low number (3 or 5)
- Use datapage or datarow locking on the replicate tables
- Change clustered indexes or partition the table to avoid last page contention

**Identity Columns & Multiple DSI**

As partially discussed before, this could cause a problem. If the parallelism strategy chosen is one based on the table/table subset strategy, then simply aliasing one of the DSI connections to “dbo” and ensuring that all transactions for that table use that DSI connection is a simple strategy. Parallel DSI's may also have to be implemented for that DSI connection as well.

However, if not - for example the more classic user/process strategy, the real solution is to simply define the identity at the replicate as a “numeric” vs. “identity”. This should not pose a problem as the identity - with the exception of Warm Standby - does not have any valid context in any distributed system. Think about it. If not a Warm-Standby, define the context of identity!! It doesn’t have any - and in fact, if identities are used at multiple sites - field sites for example, at a corporate rollup, it would have to be combined with the site identifier (source server name from rs_source_ds) to ensure problems with “duplicate” rows do not happen.

**Multiple DSI's & Shared Primary**

Again, as we mentioned before, you need to consider the problem associated with Multiple DSI’s if the replicate is also a primary database. Since the DSI connections use aliased user names, the normal Replication Agent processing for filtering transactions based on maintenance user name will fail - consequently re-replicating data distributed from Multiple DSI’s. Normally. However, as mentioned, it is extremely simple to disable this by configuring the connection parameter “dsi_replication” to “off”.

However, the re-replication of data modifications may be desirable. For instance, in large implementations, the replicate may be an intermediate in the hierarchical tree. Or, it could be viewed as a slight twist on the asynchronous request functions earlier described. Only in this case, normal table modifications could function as asynchronous requests. For example, order entry database could insert a row into a “message queue” table for shipping. At the shipping database, the replicated insert triggers inserts into the “pick” queue and the status is replicated back to the order entry system. And so on.

**Business Cases**

Despite their early implementation as a mechanism to implement parallelism prior to Parallel DSI’s, Multiple DSI’s still have applicability in most of today’s business environments. By now, you may be getting the very correct idea that Multiple DSI’s can contribute much more to your replication architecture than just speed. In this section we will take a look at ways that Multiple DSI’s can be exploited to get around normal performance bottlenecks as well as entertaining business solutions.

**Long Transaction Delay**

In several of the previous discussions, we illustrated how a long running transaction – whether it be a replicated procedure or several thousand individual statements within a single transaction – can cause severe delays in applying transactions that immediately followed them at the primary. For example, if a replicated procedure requires 4 hours to run, then during the 4 hours that procedure is executing, the outbound queue will be filling with transactions. As was mentioned in one case, this could lead to an unrecoverable state if the transaction volume is high enough that the remaining time in the day is not enough for the Replication Server to catch up.

Multiple DSI’s can deftly avoid this problem. While in Parallel DSI’s, the rs_threads table is used to ensure commit order, no such mechanism exists for Multiple DSI’s. Consequently, while one DSI connection is busy executing the long transaction, other transactions can continue to be applied through the other DSI connections. This is particularly
useful in handling overnight batch jobs. Normal daily activity could use a single DSI connection (it still could use parallel DSI’s on that connection though!), while the nightly purge or store close out procedure would use a separate DSI connection. Consider the following illustration:

The approach is especially useful for those sites which normally Replication Server is able to maintain the transaction volume even during peak processing - but gets behind rapidly due to close of business processing and overnight batch jobs.

Commit Order Delay

Very similarly, large volumes of transactions that are independent of each other end up delaying one-another simply due to commit order. Consider the average Wal-Mart on a Friday night, with 20+ lanes of checkout counters. It the transactions are being replicated, transactions from the express lane would have to wait for the others to execute at the replicate and commit in order, even though the transactions are completely independent and commit consistent. Again, because commit consistency is a prerequisite, Multiple DSI’s allow this problem to be overcome by allowing such techniques as dedicating a single DSI connection for each checkout counter. Similarly, in many businesses, there are several different business processes involved in the same database. Again, these could use separate DSI connections to avoid being delayed due to a high volume of activity for another business process. Consider the following:

Contention Control

Another reason for Multiple DSI’s is to allow better control of the parallelism and consequently reduce the contention by managing transactions explicitly. For example, in normal Parallel DSI, a typical online daemon process (such as a workflow engine) will log in using a specific user id. At the primary, there would be no contention within its transactions simply due to only a single thread of execution. However, with parallel DSI enable, considerable contention may occur at the replicate as transactions are indiscriminately split among the different threads. As a result, in the case of aggregates, etc. at the replicate, considerable contention may result. With multiple queuing engines involved, the contention could be considerable. By using Multiple DSI’s, all of the transactions for one user (i.e. a queuing engine) could be directed down the same connection - minimizing the contention between the threads.

Another example of this is also present in high volume OLTP situations such as investment banking in which a few small accounts (investment funds) incur a large number of transactions during trading and compete with small transactions from a large user base investing in those funds. However, it also can happen in retail banking from a different perspective. Granted, any single account probably does not get much activity. And when it does, it is dispersed between different transactions over (generally) several hours. However, given the magnitude of the accounts, if even a small percentage of them experience timing related contention, it could translate to a large contention issue during replication. 1% of 1,000,000 is 1,000 - which is still a large number of transactions to retry when an alternative
exists. In the example below, however, every transaction that affected a particular account would use the same connection and as a result would be serialized vs. concurrent and much less likely to experience contention.

![Diagram of account numbers](image)

**Figure 63 - Multiple DSI Approach to Managing Contention**

One of the advantages to this approach is that where warranted, Parallel DSI’s can still be used. While this is nothing different than other Multiple DSI situations, in this case, it takes on a different aspect as different connections can use different serialization methods. For example, one connection in which considerable contention might exist would use “wait_for_commit” serialization, while others use “none”.

**Corporate Rollups**

One of the most logical places for Multiple DSI implementation is corporate rollup. No clearer picture of commit consistency can be found. The problem is that Parallel DSI’s are not well equipped to handle corporate rollups. Consider the following:

- If one DSI suspends, they all do. Which means they all begin to back up - not just the one with the problem. As a result the aggregate of transactions in the backup may well exceed possible delivery rates.
- Single Replication Server for delivery. While transactions may be routed from several different sources, it places the full load for function string generation and SQL execution on a single process.
- Large Transactions issues. Basically, as stated before, a system becomes essentially single threaded with a large transaction due to commit order requirements. Given several sites executing large transactions and the end result is that corporate rollups have extreme difficulty completing large transactions in time for normal daily processing.
- Limited Parallelism. At a maximum, Parallel DSI only supports 20 threads. While this has proven conclusively to be sufficient for extremely high volume at even half of that, with extremely large implementations (such as nation-wide/global retailers), it still can be two few.
- Mixed transaction modes. In “follow-the-sun” type operations limit the benefits of “single_transaction_per_source” as the number of sources active concurrently performing POS activity may be fairly low while others are performing batch operations. Consequently, establishing Parallel DSI profiles is next to impossible as the different transaction mixes are constant.

Multiple DSI’s can overcome this by involving multiple Replication Servers, limiting connection issues to only that site and allowing large transaction concurrency (within the limits of contention at replicate, of course). In fact, extremely large-scale implementations can be developed. Consider the following:
In the above example, each source maintains its own independent connection to the corporate rollup as well as the intermediate (regional) rollup. This also allows a field office to easily “disconnect” from one reporting chain and “connect” to the other simply by changing the route to the corporate rollup as well as the regional rollup and changing the aliased destination to the new reporting chain (note: while this may not require dropping subscriptions, it still may require some form of initialization or materialization at the new intermediate site). While not occurring on a regular basis (hopefully), this reduces the IT workload significantly when re-organizations occur.

Asynchronous Requests

Addition to parallel performance, another performance benefit for Multiple DSI’s could be as a substitute for asynchronous request functions. As stated earlier, request functions have the following characteristics:

- Designed to allow changes to be re-replicated back to the originator or other destinations.
- Can incur significant performance degradation in any quantity due to reconnection and transaction grouping rules.
- Require synchronization of accounts and passwords.

Multiple DSI’s natively allow the first point but by-pass the last two quite easily. The replicated request functions could simply be implemented as normal procedure replication with the subscription being an independent connection to the same database. In this way, transaction grouping for the primary connection is not impeded, and the individual maintenance user eliminates the administrative headache of keeping the accounts synchronized.

Cross Domain Replication

Although a topic better addressed by itself, perhaps one of the more useful applications in Multiple DSI’s is as a mechanism to support cross-domain replication. Normally, once a replication system is installed and the replication domain established, merging it with other domains is a difficult task of re-implementing replication for one to the domains. However, this may be extremely impractical as it disables replication for one of the domains during this process - and is a considerable headache for system developers as well as those on the business end of corporate mergers who need to consider such costs as part of the overall merger costs.

The key to this is that a database could participate in multiple domains simply by being “aliased” in the other domain the same way as Multiple DSI approach - because in a sense it is simply a twist on Multiple DSI’s - each domain would have a separate connection. Consider the following:
Once the concept of Multiple DSI’s is understood, cross-domain replication becomes extremely easy. However, it is not without additional issues that need to understood and handled appropriately. As this topic is much better addressed on its own, not a lot of detail will be provided, however, consider the following:

**Transaction Transformation** - Typically the two domains will be involved in different business processes. For example, Sales and HR. If integrating the two, the integration may involve considerable function string or stored procedure coding to accommodate the fact that a $5,000 sale in one translates to a $500 commission to a particular employee in the other.

**Number of Access Points** - If the domains intersect at multiple points, replication of aggregates could cause data inconsistencies as the same change may be replicated twice. This is especially true in hierarchical implementations.

**Messaging Support** - Replicating between domains may require adding additional tables simply to form the intersection between the two. For example, if Sales and Shipping were in two different domains, replicating the order directly - particularly with the amount of data transformation that may need to take place - may be impractical. Instead “queue” or “message” tables may have to be implemented in which the “new order received” message is enqueued in a more desirable format for replication to the other domain.

While some of this may be new to those who’ve never had to deal with it, particularly, any form of workflow automation involves some new data distribution concepts foreign to and in direct conflict with academic teachings. Since cross-domain replication is a very plausible means of beginning to implement workflow, some of these need to be understood. However, it is crucial to establish that cross-domain replication should not be used as a substitute for a real message/event broker system where the need for one clearly is established. Whether in a messaging system or accomplished else wise (replication), workflow has the following characteristics:

**Transaction Division** - While an order may be viewed as a single logical unit of work by the Sales organization, due to backorders or product origination, the Shipping department may have several different transactions on record for the same order.

**Data Metamorphism** - To the Sales system, it was a blue shirt for $39.95 to Mr. Ima Customer. To Shipping, it is a package 2x8x16 weighing 21 ounces to 111 Main Street, Anytown, USA.

**Transaction Consolidation** - To Sales, it is an order for Mrs. Smith containing 10 items. To credit authorization, it is a single debit for $120.00 charged to a specific credit card account.

And so forth. Those familiar with Replication Server’s function string capabilities know that a lot of different requirements can be met with them. However, as the above points illustrate, cross domain replication may involve an order of magnitude more difficult data transformation rules - spanning multiple records - not supportable by function strings alone. While “message tables” could be constructed to handle simpler cases, it increases I/O in both systems and may require modifications to existing application procedure logic, etc. Hence advent and forte of Sybase Enterprise Event Broker - also a topic better addressed individually. Having transitioned to here, it is best to stop.

Help stamp out ignorance - connect data islands by distributing data to where it is needed!!!