

GSSync



Users Guide

DISCLAIMER OF WARRANTIES. THIS SOFTWARE IS LICENSED AND PROVIDED "AS-IS" AND, TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, GOLDSTAR SOFTWARE INC, DISCLAIMS ALL GUARANTEES AND WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, REGARDING THE SOFTWARE AND RELATED MATERIALS, INCLUDING ANY WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE, TITLE, MERCHANTABILITY, AND NON-INFRINGEMENT. GOLDSTAR SOFTWARE INC. DOES NOT WARRANT THAT THE SOFTWARE IS SECURE OR FREE FROM BUGS, VIRUSES, INTERRUPTION, OR ERRORS, OR THAT THE SOFTWARE OR SERVICES WILL MEET YOUR REQUIREMENTS. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU. IN THAT EVENT, ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO 30 DAYS FROM THE DATE OF PURCHASE OR DELIVERY OF THE SOFTWARE, AS APPLICABLE. HOWEVER, SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY HAVE OTHER RIGHTS THAT VARY FROM STATE TO STATE.

Pervasive, Pervasive Software, Pervasive PSQL, Zen, Btrieve, and DataExchange are trademarks or registered trademarks of Pervasive Software Inc. or Actian Corporation.

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation.

All other company and product names are the trademarks or registered trademarks of their respective companies.

© Copyright 2009-2023 Goldstar Software Inc. All rights reserved. Reproduction of this software or documentation without expressed, prior, written permission from Goldstar Software Inc. is prohibited.

Portions of this product are © Copyright 2004 Nick Gammon, <http://www.gammon.com.au/>.

Portions of this product are © Copyright 2003 Daniel W. Howard.

Portions of this product are © Copyright 2007 Pervasive Software Inc.

This document is specifically for GSSync Version 2.11. If you are running a different version, please check for any changes to the settings or options.

Goldstar Software Inc.
1945 Maplewood Lane
Munster, IN 46321
<http://www.goldstarsoftware.com>

Contents

Chapter 1: Overview	1
Introduction	1
Features	1
System Requirements	2
Software Installation	2
Chapter 2: Understanding Replication.....	5
General Replication Concepts.....	5
Determining the Changed Data: Replication Metadata	5
Understanding the Source Data: Source Metadata	6
Working with the Target Database: Target Metadata.....	7
Unidirectional Replication	7
Bidirectional Replication	8
Action PSQL/Zen Database Concepts.....	8
Btrieve File Layout	9
System Data Values and Key.....	9
Data Dictionary Files	10
Variant Records.....	11
Named Databases	12
DataExchange Replication.....	12
Replication Metadata	12
Replication Process	13
Trade-offs with DataExchange Replication	14
GSSync Replication Using DX Metadata.....	14
Replication Metadata	15
Replication Process	15
Trade-offs with GSSync and DX Metadata	15
GSSync Replication Using Archive Log Metadata	16
Replication Metadata	16
Replication Process	18
Trade-Offs with GSSync and Archive Log Metadata	18
GSSync Replication Using GSSync Metadata	19

Replication Metadata	19
Replication Process	20
Trade-offs with GSSync and GSSync Metadata	21
Choosing a Replication Solution	21
Chapter 3: Running GSSync.....	23
Running GSSync	23
Using the Configuration File.....	25
Using the Configuration Database	26
Using the Command Line Options	26
Specifying Options at the Command Line.....	26
Overriding Configuration File Settings.....	26
Checking GSSync Return Codes	27
Scheduling GSSync	28
Chapter 4: Importing Data into GSSync	31
Specifying the Source Data	31
Btrieve File Name.....	31
Btrieve Owner Name	32
Btrieve Key Number.....	33
Specifying the Source Data Structure	34
Reading a DDF by Path.....	34
Reading a DDF by DBName	35
Providing DDF Access Rights.....	36
Indicating a Table Definition.....	36
Overriding the Btrieve File Name	37
Specifying a Variant Record Definition	37
Reading an XML Definition File.....	40
Specifying the Metadata	45
Specifying Archive Log Metadata.....	46
Specifying PDC Metadata.....	46
Indicating Starting and Ending Timestamp	47
Building GSSync Metadata	48

Replicating with GSSync Metadata	48
Exporting Without Metadata	50
Chapter 5: Exporting Data from GSSync	51
Target Data Formats	51
No Output	52
Adding Dates To File Names	52
Unformatted File.....	53
Btrieve File	54
Queue File	54
Comma-Delimited Text File.....	55
CSV Header Row	56
CSV Export Format	57
Vector VWLOAD Text File	57
XML File.....	58
Raw XML Format.....	59
Field XML Format	59
XML Tags	60
JSON File.....	60
Readable JSON Format	61
Compact JSON Format	62
JSON Names	62
SQL Script Export File	63
ODBC Database Target.....	64
Forked (SQL and ODBC Database) Target	65
SQL/ODBC Statement Definitions	66
Replacement Strings	66
DELETE Statements	68
INSERT Statements	68
UPDATE Statements.....	69
Using Multiple SQL Statements	69
Deleted Data Output.....	70

Ignoring Deleted Data	71
Additional Output Fields	71
Delta Flag	71
System Data Value	72
Update System Data Value	73
Btrieve Position Value	74
Last Change Timestamp	74
Export Timestamp	76
Ignoring Fields	78
Ignore Field List	78
Data Output Formatting Options	78
Boolean Export Format	78
Date Export Format	79
Time Export Format	80
Timestamp Export Format	80
Export Parallel Output	81
Data Output Filtering Options	82
Date Field Filtering	82
Character Field Filtering	83
Numeric Field Filtering	84
Retain Leading Zeroes in NUMERIC Fields	85
Record Filtering	86
Chapter 6: Interpreting the GSSync Log File	89
Log File Options	89
Specifying the Log File Location	89
Specifying the Log Level	90
Log File Details	90
Changing the Display Language	91
Chapter 7: Reviewing Replication Examples	93
Btrieve Replication with GSSync Metadata	93
Configuring Initial Replication	93

Updating the Data Set.....	93
CSV Replication with Archive Log Metadata.....	94
Configuring Initial Replication.....	94
Updating the Data Set.....	94
SQL Script Replication with PDC Metadata.....	95
Configuring Initial Replication.....	96
Updating the Data Set.....	97
ODBC Replication with GSSync Metadata.....	99
Configuring Initial Replication.....	99
Updating the Data Set.....	100
Archiving with Btrieve Replication with a Filtered Result Set to CSV	102
Purging Data from a Filtered Result Set.....	103
Chapter 8: GSSyncImport.....	105
Running GSSyncImport	105
Using the Command Line Options	105
Chapter 9: SyncAllTables.....	107
Configuring SyncAllTables.....	107
Running SyncAllTables	107
Using the Command Line Options	107
Extending This Tool	108
Chapter 10: The GSSync Scheduler	109
Preparing to Run GSSync Scheduler for the First Time.....	109
Configuring PowerShell to Run Unsigned Scripts	109
Creating the Configuration Database	109
Running the GSSync Scheduler the First Time	109
Understanding the GSSync Scheduler Database	110
The Jobs Table.....	110
The Log Table	112
Internal Data Definitions.....	113
Creating Jobs	113
Creating a Run-Once Job.....	114

Creating a Repeating Job	115
Defining the Job Schedule.....	115
Specifying Synchronization Jobs	115
Managing Jobs	117
Running the GSSync Scheduler After the First Time	118
Using GSSync Scheduler Off-Label	119
Reconfiguring GSSync Scheduler While Sleeping	119
Managing the Job Log	120
Appendix A: Sample XML Configuration File	121
Appendix B: Version History	129
Appendix C: Known Limitations	133

Chapter 1: Overview

Introduction

Thank you for your purchase or evaluation of GSSync, Goldstar Software's data replication solution for Actian PSQL/Zen databases! GSSync provides Actian database users with the ability to replicate data from their PSQL/Zen environment to several different target environments, including Btrieve files, comma-delimited files, SQL script files, and even target ODBC databases directly. This high-speed, low-overhead solution is designed from the ground up to be fast and flexible to meet your replication requirements, and is designed to work with several different types of metadata, from our own proprietary data to the PDC tables provided by DataExchange for the ultimate in flexible replication capabilities.

Features

The main features of GSSync include the following:

- Performs well from a high speed, low overhead command line environment
- Offers ease of scripting and utilization of multiple CPU or multi-core servers
- Configures easily with XML configuration files or command-line options
- Accesses Btrieve files directly for the best raw performance
- Runs with Pervasive.SQL 7 or above for maximum compatibility, but will leverage newer features up through Zen v15 if available
- Reads multiple types of file description metadata, including Actian Data Dictionary Files (DDF's) or XML file definitions
- Works without any metadata for pure Btrieve-to-Btrieve replication
- Supports multiple data change metadata options, including PDC tables from DataExchange, Archive Logs, and a proprietary GSSync format
- Writes data to multiple targets, including UNF files, Btrieve files, Btrieve queue files, comma-delimited (CSV) files, XML files, JSON files, SQL scripts, and direct-to-ODBC databases
- Corrects minor errors in common field types, including dates, strings, and more
- Handles variant record structure definitions for source files

- Filters records based on a user-defined formula with mathematical function support, randomization, and more

System Requirements

GSSync is a command line application that has no overhead associated with a GUI environment, so it runs very fast with very few resources required. GSSync requires a minimum of a Pentium CPU and 512MB of free memory to run correctly.

GSSync also requires a Pervasive.SQL 7 or higher database engine, although certain features may require a newer version. Using a Named Database to specify the data location requires the Pervasive PSQL v9 (or higher) database engine. Using the UpdateSystemData feature requires Actian Zen v14 (and v13 data files) or newer. (If you REALLY need GSSync to work with Btrieve 6.x, please contact us for a copy of v1.78.)

If GSSync is going to use PDC metadata, then DataExchange 2.5 or higher is also required. The DataExchange environment must be properly configured per the product instructions and activated on the First Site. No partner sites need be defined to use the data capture features of DataExchange with GSSync.

Exporting to an ODBC target database will require a suitable database engine and ODBC driver.

Software Installation

As GSSync utilizes the Actian PSQL/Zen database engine, you should have the database engine properly installed and activated before starting to work with GSSync. If you are running GSSync on the database server (which is highly recommended due to performance advantages), then no additional steps are required. If you are running GSSync on a client workstation, then you will need to install and configure the Actian PSQL/Zen Client for that environment. See the Actian manuals for more information on the installation process.

As a command line application, there is no formal installation for GSSync. To prepare your system for GSSync, simply copy the files from the ZIP download onto your computer in an accessible location (like C:\GSSYNC). If you want to be able to run GSSync from any directory, then including the files somewhere in your path is also recommended. (Copying the files to the C:\WINDOWS directory would provide the same result, but this practice is frowned upon by Microsoft. Instead, consider copying them to your Actian BIN folder.)

Once the files have been copied, you also need to apply your GSSync license. Download and run the GSLicKey application on the computer that will be running GSSync and enter your license keycode as provided by Goldstar Software. Once the license is applied to the computer, you will see serial number and registration information when you run GSSync.

If you intend to run GSSync from a scheduled task, be sure to specify the same user account for the task for which you installed the license. If you are running from an automated task scheduling application, such as AutoTask 2000, then you should modify the service account used for the scheduler to a specific login name, and to NOT use the "Local System" account. You can change the login information for the service by right-clicking the service in the Control Panel's Services applet, then selecting Properties and going to the Logon tab. If you need additional help with this, please check the manual that came with your scheduling application for more information about running with an explicit user account.

Chapter 2: Understanding Replication

The biggest problem with replication is not in the technology -- it is in the expectations that people have about the technology. At first glance by a novice, replication seems to be very easy: simply copy all data from one system to the other.

However, in the real world, replication solutions are almost never easy *or* simple. They are bogged down by poor or incorrectly defined source data definitions, poor adherence by the application to supported data types, and the almost universal lack of metadata to define the changed data set. They can also experience problems with the replication process itself, especially after a communications failure or some other problem occurs.

In order to understand how GSSync works, it is therefore critical to understand how replication works in general first, so that you can see the potential issues and design your replication solution to avoid them.

General Replication Concepts

Let's start with the general concept of replication. What is replication and how does it work?

In a nutshell, replication is the ability to maintain one or more copies of a block of information, allowing updates to propagate to those copies with a limited time lag. This means that the replication environment must be able to ascertain which data has changed, and thus determine which data must be copied. It must understand the source data that is being moved at some level or another. It must know where the data is moving to, and it must know how to work directly in that environment. If the data in the target layout is of a different format than the source layout, it must be able to translate the data from the source format to the target format. Finally, it has to do all of this quickly and efficiently to avoid dragging down performance on the original environment. On top of all of this, we also add the common requirement for bidirectional replication.

With all of these critical components, something can go wrong just about anywhere in this process. As they say, "the devil is in the details" in any replication environment.

Determining the Changed Data: Replication Metadata

Every replication environment needs some sort of data about the data, or *metadata*, to keep track of the various records as they are being changed. This replication metadata is therefore critical to the efficient operation of the replication environment, as it is how the replication environment determines which data must be replicated on any given cycle. Without metadata, we can only perform a full data copy every time, which is horribly inefficient.

As you may be aware, there are several different types of changes, and each must be tracked differently by the metadata.

- For database inserts, when new records are being created, you need to know the exact date and time of the new insert, so that you can track it and replicate the new insert accordingly. Luckily, it is relatively easy to add the timestamp of when a record was inserted to any data record and track it accordingly.
- For database updates, you may need to know the original record data AND the new record data in order to tell what has changed. In many cases, replication solutions don't worry about which fields were changed -- they just blindly replicate the entire record, which is usually sufficient. Again, the timestamp of the last change can be easily added to a record and tracked by the replication solution.
- For database deletes, things get a bit more complicated. We cannot store the timestamp of the record delete in the file -- because the record is no longer there! Instead, we must somehow track the deleted record through some other means, commonly known as an obituary, which represents a deleted record for some period of time. These obituaries then have to be managed, too, which adds to the complexity of the solution.

In some environments, the application explicitly updates a field in each record that indicates the last time the record was changed, so a replication solution is able to handle inserts and update, but not deletes. However, even this limited capability is quite rare, as most application developers do not build in the concepts of replication when they start writing a new application.

When this data is not stored and maintained by the application, the task falls to the database engine or some other tool to maintain this metadata. For many database engines, though, there is no natively-available and user-accessible metadata that can provide this information, and the Actian PSQL/Zen database is no exception.

Understanding the Source Data: Source Metadata

The next piece of the replication puzzle is to understand the source data. Obviously, this is a requirement because we need to read the source data and perform some actions on it. The ability to understand this source data is directly dependent on the existing of *source metadata*.

Source metadata, like the replication metadata, is simply data about our data. In this case, though, it is information about how to interpret the data that we are replicating. In many databases, the source metadata is built into the database itself. SQL Server, Oracle, and all other SQL databases contain a set of tables called *system tables*, or the *system catalog*, which defines how the data is to be interpreted.

In the Action world, we can get our source metadata from the same place -- the system catalog is called *data dictionary files* for PSQL/Zen environments. However, many applications work at the Btrieve interface, which is a low-level, high-throughput interface designed for performance. At the Btrieve level, records are merely bytes -- there is no metadata here! As a result, developers have a fantastic flexibility to do whatever they want with the data bytes in each record, and some application developers have certainly taken full advantage of this with custom data types, variant records, and more.

Obviously, if we lack true metadata, then we will not be able to understand the source data, and translation to the target environment will be a lot more challenging.

Working with the Target Database: Target Metadata

Know which records are changing and what they contain is the majority of the puzzle, but you must also understand the target environment. If you are writing to a specific environment, such as one SQL engine, you may have specific rules for statement syntax that you must adhere to, and these rules could be different from other SQL engines. As such, having *target metadata* available may also be critical.

In many systems, the target may be simple text files, making metadata less critical. However, you must still have the correct target file format if you expect someone else to read the database once it is replicated. No matter what your target, if you get data in the wrong format, then the odds of a successful replication are pretty slim.

Unidirectional Replication

With unidirectional replication, the replication solution is responsible for interpreting the metadata, determining the data that has changed, and then sending the data to the target environment. Data is always migrated from the primary database to the secondary database, in one direction only.

This one-way migration of data seems straightforward, but even it has some issues. What if a record is inserted, updated, and then deleted in the same replication cycle? If it is propagated as an insert, update, and delete in order, then you are replicating a lot of data for no reason. If you only do the last item, in this case the delete operation, then what should be deleted on the target database, if nothing was ever inserted? What if you have an insert and ten updates? You can just propagate the last record structure, but it must be replicated as an insert instead of an update, since the insert never took place. Ugh!

Luckily, with unidirectional replication, we can assume that the source database is always the master copy, and we can refresh the data at any time by deleting the target database and pushing a full replication of all data. This provides some measure of ensured data integrity that we can always fall back to in the event of a failure.

Bidirectional Replication

Bidirectional replication, on the other hand, is the ability to migrate data in both directions at the same time, retaining the "best" (or newest) values from both the primary and secondary databases.

Bidirectional replication retains all of the complexity of unidirectional replication, but it adds an entirely new set of complex issues. What happens when a record is updated on both sides at the same time? Should you just keep the latest version? Merge both sets of updates somehow? What about unique values (such as invoice numbers) that get duplicated, but the invoices are for different customers? Somehow, you must renumber one of the invoices, and adjust everything else downstream. What about when an insert happens on one side, and both an insert and delete happen on the other side, but AFTER the first one?

Complicating the matter further is the fact that many of the decisions that need to be made with respect to handling these replication conflicts are defined not by the replication environment, but rather by business rules surrounding the data being replicated. Oftentimes, the same conflict in two different files will be handled completely differently. Sometimes, data is so critical that you're better off NOT changing data, and you should notify someone to handle the replication manually, instead. This is why bidirectional replication solutions are incredibly complicated and require substantial configuration effort, and also a good reason why they almost never work as expected.

Another major issue with bidirectional replication is failure recovery. There is no easy way to recover from a failure, since you may not know what data was replicated and what data was not yet replicated. Any bidirectional solution will need to keep additional data on where it is in the replication cycle at every moment, so that it can recover gracefully and correctly.

As the problems of bidirectional replication often outweigh the benefits, most replication environments don't handle two-way synchronization like this. GSSync is no different, and we will not look at GSSync for use in two-way replication. If you need bidirectional capabilities, then you should look at implementing DataExchange first, or have the application developer create his or her own change tracking and replication solution.

For the remainder of this manual, we will assume unidirectional replication only.

Action PSQL/Zen Database Concepts

In the PSQL/Zen database world, there are a few more issues that we have to cover, too, including the Btrieve file layout, the benefits of System Data, Data Dictionaries, and more.

Btrieve File Layout

Btrieve files, by definition, contain record "blobs", or sets of bytes that are not interpreted by the database itself. Instead, all data interpretation and database linking is handled by the application code itself -- hard-coded into the source by the developer. In essence, the database acts as a "navigational record manager", storing and retrieving records on behalf of the application, but providing no other services. The application then links data by navigating from one record to another using known data from one table to read a record from another table. This navigational record manager concept known as Btrieve is the driving force behind the performance and stability of the PSQL/Zen database engine, and it provides a nearly-infinite level of flexibility for the developer to do all sorts of strange and wonderful things.

At its most basic level, Btrieve "blobs" have a given length -- and that is all. They are simply presented to the application as a set of bytes in a given order. As such, if the application decides to store variable blobs, or create custom-defined data types, this is transparent to the database engine itself. It has no bearing on the database at all.

If an application is developed with a relational data structure, on the other hand, then it *may* choose to follow the more rigid rules required by the SQL engine -- using well-defined data types, using well-defined record layouts, and so on. When this happens, we say that the data is "relationally-compatible". However, it is important to note that even following the more rigid rules, the database engine does not REQUIRE that the data follow the rules -- you can still put garbage into string fields, bad dates (like 02/176/2021) into date fields, and so on. If a developer takes any of these liberties, then any solution that relies on interpreting the data independent of the application is likely to fail. This is the primary reason why PSQL/Zen replication is so complicated.

System Data Values and Key

System Data is a set of special system-maintained database values (and their associated key) that is maintained internally by the database engine. A System Data Value is an 8-byte value that can be used to uniquely identify any one record in a given database file for the life of that record. It is tagged to a record when the record is created, and never modified over the lifetime of that record.

As you can see, System Data provides a nice way to identify any given record consistently. In fact, System Data is required by PSQL/Zen for its own replication solution, DataExchange, as this value is used to link the metadata back to each individual record -- without being dependent on any field within the data whatsoever.

One of the problems with using System Data for replication, though, is that while it is partially visible from the Btrieve interface, it is really a hidden value and Actian has only provided a viable mechanism for accessing this value from the SQL interface starting

with Actian Zen v15. The lack of this key feature makes any type of ODBC-based, stand-alone replication solution simply impossible to create.

System Data was introduced in the PSQL 7.x file format, so any replication solutions using System Data as part of the metadata will need all database files to be in at least 7.x format, and the database engine will need to be Pervasive.SQL 7 or newer, too. Further, System Data is an *optional* feature of the 7.x file format, so it may be necessary to perform some extra preparation on the database files to ensure that they have been rebuilt with the System Data value and key before enabling replication. This task is accomplished by the Rebuild utility and is fully documented in Actian's documentation.

A Special Note about System Data: *Officially, the System Data value is an 8-byte **unsigned** integer in PSQL/Zen. However, many databases, such as Vector and SQL Server, cannot handle this data type properly, forcing you to use a NUMERIC(20,0) or other oversized and inefficient data type. Since no calculations should ever be required on this value, we have made the decision to export this value as a SIGNED INTEGER instead. While this can have an impact on the ability to see the order in which records were created, it should have little other impact on the data migration process. If needed, we can add a flag in GSSync to make this value output as a signed or unsigned value, if this becomes an issue in the future.*

A new addition in Actian Zen v14 (but first documented with v15) is a second system data value, known as the UpdateSysKey value. When enabled on a data file, this second, internal, 8-byte value provides a true timestamp of the last UPDATE operation that completed against this record.

Data Dictionary Files

Data Dictionary Files, also known more succinctly as DDF's, are used by the SQL engine inside PSQL/Zen to help define the data layout of the Btrieve "blob", and can also be used by GSSync to understand the blob contents.

Data dictionaries are defined through a set of files that minimally includes FILE.DDF, FIELD.DDF, and INDEX.DDF. Other DDF files may exist as well, but these three files are the critical system tables that are needed to define the tables, columns, and indices for each table to the database engine.

Starting with Pervasive PSQL Summit v10, Actian added a newer DDF format, commonly called "V2 Metadata", that offers longer table and column names. These DDF's have physical file names of PVFILE.DDF, PVFIELD.DDF, and PVINDEX.DDF, in addition to the other DDF files. GSSync can work with either V1 or V2 metadata, but will always default to using V2 Metadata if it is available.

DDF file definitions are not provided by every application, as it is typically the task of the application developer to jump through the hoops to create them, and this is often

beyond the scope of the original application. Although some application developers go the extra mile to provide good DDF's, it is far more common to have poorly defined DDF's, or simply incomplete DDF's, for a given application. This makes it infinitely more challenging to get any replication solution working from a Btrieve database to a non-Btrieve file format.

As a rule, if you start working on a replication solution to CSV, SQL script, or ODBC targets and have any issues with data being interpreted incorrectly, the first step should be to validate your DDF's. Start by contacting the developer to get the "latest and greatest" version of the DDF's for your application. Then, use the Check Database Wizard (part of the PCC in Pervasive.SQL 2000i and Pervasive.SQL V8 and part of the DDF Builder in Pervasive PSQL v9 and newer) to check the definitions against the original Btrieve files. If they don't match, you will need to either go back to the developer again for a fix, or you may find that you need to fix the data definitions yourself. This is not a project for the novice!

If you find that you have too many problems with the data dictionary files, then you should consider an alternate file definition technology, such as the XML file definitions supported by GSSync. XML file definitions are really just text files that contain the same metadata information, but are far easier to build and edit.

Of course, if you need help to resolve the issues further, remember that Goldstar Software has been working with Btrieve files and DDF's for over 10 years and has expertise and tools that may help.

Variant Records

The PSQL/Zen database is well known and well respected by application developers because it puts them in charge of every aspect of the database design. One of the "features" of the Btrieve interface is that a developer can store whatever data he wants to store in any record in a file. This leads to the concept of *variant data records*.

A variant record is one in which the developer chose to store multiple data definitions in the exact same low-level database file. For example, there may be little difference between the data needed for customers and vendors – you need name, address, and other common data elements. A developer may elect to store both types of data in the same file, then indicate which record is which using a special *record type* field, which may simply be "C" for customer and "V" for vendor, or 0/1, A/B, or any other indication.

You can tell if you have variant record definitions in your database with a simple query:

```
SELECT Xf$Loc, COUNT(*) FROM X$File GROUP BY Xf$Loc HAVING COUNT(*) > 1
```

If this query returns any data records, then your database may be using variant records.

Variant records often cause huge issues for SQL data access, because all of the variant data records are visible on every SQL query. For example, if I issue the simple query:

```
SELECT * FROM Customer
```

then I will get BOTH customer and vendor records back. If the field definitions are slightly different for the Vendor table, then this query may return garbled data or other garbage for some of the fields for the vendor records.

The way to fix this is to alter the SQL query to take the record type field into account, like this:

```
SELECT * FROM Customer WHERE RecType = 'C'
```

Consequently, you would access the Vendor data with a similar query:

```
SELECT * FROM Vendor WHERE RecType = 'V'
```

As you can imagine, exporting data from a variant database is quite complicated as well. Luckily, GSSync supports variant records handling, simplifying this process.

Named Databases

A Named Database is a concept that exists logically in the database engine. The named database consists of a database name that the engine keeps track of which, in turn, contains related information such as the data dictionary directory, one or more data file directories, and a few other attributes.

Named databases are visible in the Control Center as the blue database icons in Pervasive PSQL v9 and newer, and you can get information about the named database by right-clicking the database and selecting **Properties**. The dictionary and data file locations can be easily changed from the resulting dialog box.

Named databases were also used in older versions, and they are visible in the PCC in Pervasive.SQL 2000i and Pervasive.SQL V8 by right-clicking the **Configuration** item in the PCC tree view, then selecting **Maintain Named Database**. Highlight the named database in the resulting dialog box to see and change the related properties.

DataExchange Replication

As the first replication technology to be examined, we will look at the functionality of DataExchange replication, designed to replicate a PSQL/Zen database to another similar database. It may very well be that you can use the DX replication environment to suit your entire needs, eliminating the need for any add-on tools.

Replication Metadata

The metadata for tracking changes in DataExchange is stored in a set of files called the PDC tables. These are actually Btrieve-based files that contain one metadata record for every real data record that exists in the live Btrieve file, as well as the deleted record obituaries, if any. Each PDC table is created to have the letters PDC at the beginning, followed by the SQL table name (from the DDF's) of the table being tracked.

Building Metadata

When you activate a file with DataExchange, the activation process first creates a new PDC table. Then, it reads every possible record from the source table, obtains its system

data value, and then inserts a new record into the PDC data file indicating that the record is "new" as of the start of the database. This process then loops until ALL database records are read and all PDC records are inserted, which can take a substantial amount of time on a large database.

When the activation step is completed, you will have a "live" file that contains your real data, as well as a metadata file (the PDC file) that contains the same number of records, each with a system data value that acts as a pointer to the corresponding record in your source database. This PDC record is a blob of its own, containing the system data value, the last change timestamp, and a deleted flag, among other fields.

Maintaining Metadata

After a file is activated, it is up to the change capture process (an integral part of the DataExchange product) to keep track of all changes to the data file in question. When a new record is inserted, the change capture mechanism inserts a new record into the PDC table with the system data value of that record and the timestamp of the insert. Every time a record is updated, the change capture updated the corresponding PDC record with the time of the last update. When a record is deleted, the change capture updated the last updated timestamp again, and it sets the deleted flag to create an obituary so that the database can keep track of the deleted record for a period of time.

Over time, deleted record obituaries will eventually fill up the PDC and bog down the system. Another process, known as the clean-up process, comes by periodically and removes the references to deleted records that are older than 90 days (by default). This allows replication to work normally as long as it replicates all data at least every 90 days. Of course, if replication is stopped for a period over 90 days, the obituaries will expire and get purged -- meaning that the metadata is no longer valid. A situation like this requires a complete re-replication of the entire database. This is not an impossible situation to recover from, but it can be definitely time-consuming.

Replication Process

Once the metadata challenges are overcome, the DataExchange replication process is actually quite straightforward.

First, the two DataExchange engines get together and agree on the time of the last replication. (They have to agree because of a problem known as clock drift, in which the times of the two servers can differ, possibly losing some updates.) Once they agree, they can start the replication process.

On the source server, replication process looks at each file separately, and if a file has been changed at any time since the last replication cycle, a database lookup is done in the PDC metadata records for all data records that have changed since the agreed-upon time. For each record that has changed, the system data value from the PDC record is

used to read the live data record as it exists right now, and both the PDC record and the live database record are sent to the target engine for processing.

Once the data gets to the target engine's DataExchange service, the live data record is written directly into the Btrieve data file as a blob, and the PDC data is updated accordingly in the PDC file. When a delete is replicated, the record is deleted from the target server, as expected. This process continues until all changed records have been replicated.

Trade-offs with DataExchange Replication

Since DataExchange relies on the PDC files to track changes, it is completely transparent to the application. Since DataExchange also works on the raw record blobs without regard to their contents, it is also completely transparent to the data itself, making it work with a wide variety of applications, even those for which data definitions are not available.

The ignorance of the data contents, though, makes it hard to deal with conflicts or to provide replication to other (i.e. non-Actian) environments. Without valid file definitions or source metadata, replication to a non-Actian target is simply not possible, and requiring this information is beyond the scope of DataExchange in the current versions.

Further, since the system data values are transparent to the database, we are also unable to build a pure SQL-based replication solution based on DataExchange metadata. This is because although we can read the metadata and determine which records changed (by finding the system data value), we have no mechanism to retrieve a record from the live data file by the system data value. The net result is that replication through SQL scripting, and thus "PSQL-to-Any" replication, is simply not possible.

GSSync Replication Using DX Metadata

Actian has spent years of research and development efforts on their own DataExchange replication solution. As such, it seems like such a shame to throw it away and build another solution, just because it doesn't work for all environments, such as replication to a non-PSQL/Zen target.

In fact, the maintenance of the DataExchange metadata is a perfectly functional feature, and it is well worth the cost of the DataExchange product. A proper DataExchange deployment will provide us with all of the metadata creation and maintenance that we need to replicate our data. We will just need to use a different tool to replicate the data, such as GSSync.

The primary advantage to using GSSync to replicate instead of DataExchange is that we can obtain some finer control over the data handling side of the fence. We can support a wider variety of target databases, including Btrieve, SQL, and ODBC databases. We

can work around other issues and problems and be compatible with a wider variety of database environments, too, since DX doesn't work with data records over 64K in size.

Replication Metadata

The metadata used by GSSync in its replication with DX metadata is, obviously enough, the same metadata that is created and maintained by DataExchange.

The metadata records used by DataExchange contain the following information:

- A global ID system data value
- A global site id where the record was created
- A local ID system data value (the system data value of the record on this server)
- The record's creation timestamp
- The record's last updated timestamp
- A deleted flag
- The record length

For additional information about the DX metadata, please refer directly to the DataExchange manuals.

Replication Process

The replication process is also similar to that of DataExchange. When a file replication is requested, you provide a starting timestamp to GSSync, which then starts to scan the PDC table for all records that have been updated or deleted since that time. Because the LastChangedDate field is a defined key in the PDC file, this process must ONLY read the number of records that corresponds to the number of records actually changed since the last replication cycle, dramatically cutting down on the work required.

When a record change is found, it is sent to the export handler and the record is written to the target database, in whatever format is appropriate. Record deletes are also processed as needed.

Trade-offs with GSSync and DX Metadata

It should be clear that for this feature of GSSync to work, you still need to purchase and deploy DataExchange onto your primary server. If your environment can be served by DataExchange replication alone, then the extra purchase of GSSync is clearly superfluous. This is especially the case when you are replicating as a fault tolerance solution. Why manage more complexity and another tool if you don't need to?

However, a real advantage to GSSync is that you need to ONLY set up a primary server -- you do NOT need to configure a secondary replication server (a partner in DX terminology) to receive the updates. This can reduce your hardware, OS, and database software license requirements. Additionally, if you want to replicate the data to a non-PSQL target, such as to a web server or other company-wide database, then GSSync will be just the ticket.

GSSync Replication Using Archive Log Metadata

The Btrieve/PSQL/Zen database has long had a built-in capability known as Archive Logging, which allows the database to track all changes going into a given database file into a separate log file. This archive log process captures each database insert, update, and delete operation as it occurs into a separate archive log file, which can then be interpreted by the Maintenance Utility to "roll forward" a database after a failure. This feature is closest to what other databases term a transaction log, and moving the data via this log is commonly called "log shipping" by other vendors.

This Archive Log information can be read by GSSync as a transcript of all database changes that have been processed in a file, which can then be formatted and written directly to the target database.

Replication Metadata

The archive log metadata is stored in a single log file that contains a small header, and then it contains information about each record change that occurred in the file.

The metadata records used by Archive Log contain the following information:

- The operation (insert, update, delete) that took place
- The key number which was being acted on
- The position of the record within the Btrieve file
- The client ID that caused this change
- The timestamp of this change
- The length of the record
- The actual record data being written
- Additional data needed for roll-forward purposes

It is *absolutely critical* to understand that the Archive Log metadata does not employ the System Data value to track a given record. Instead, it has exactly ONE value to refer to the database record -- the physical record position. In any given database, the physical record position is NEVER guaranteed to be unique. In fact, if you delete a record and

then insert a new record into the file, there is a good chance that the database will re-use the record slot you just deleted. This makes it look like a delete and an insert occurred to the same record in sequence, when in fact, they are completely different records. The ONLY way that an archive log replication can be considered "complete" is if all downstream replication targets contain this unique record position as a unique identifying key, and if all replication changes are committed, *in the exact order of occurrence*. This limitation of the Archive Log metadata is why Actian does not recommend nor use the archive log files for replication in DataExchange, and why they discourage its use for replication or for any other purpose than data recovery.

So, given this limitation, why is this even an option in GSSync? We are all often aware of the "best practices" in our daily life -- floss after meals, exercise daily, make and test backups, etc. However, sometimes events conspire to prevent these practices from being used in our real world. For example, if you have a Btrieve 6.x database engine and you would like to replicate data to SQL Server, but you cannot upgrade your old Btrieve engine or application, what can you do? You cannot use DX, since you cannot use System Data, so you need another option. Sometimes, archive logging may be your only viable option, and GSSync will offer you a possible solution, as long as you are aware of the limitations.

Building Metadata

Archive log metadata is created by defining the Archive Log configuration. To do this, create a file in the root of the drive that contains the database files, and call it BLOG.CFG. Inside this text file, create a line for each database file to be logged, containing the data file name (and its full path), an equal sign, then the full path to the log file. When you restart the engine, changes to these files will now be included in their respective log files. (For complete details on properly configuring the archival logging feature of the database engine, please refer to your product manuals.)

Maintaining Metadata

Periodically, you must move the log files from the archive log folder to the replication folder. This is one of the big downsides of the archive logging solution, and this is one reason why many users avoid it. The only way to ensure that the log files are all closed is to completely shut down the database engine, move the log files, and then restart the engine. Obviously, this is only ideal for a daily log file replication solution, as stopping the engine kicks all users out of the application.

Once the log files have been moved out and the engine restarts, new log files will be created, and the cycle restarts.

Replication Process

The replication process starts when the archive log files are moved out of the archive directory into a replication directory. Again, the database engine must be completely shut down for this to work properly.

Then, GSSync is run with each file's archive log. The archive log is read and parsed, one record at a time, and the corresponding changes are sent to the target, in the exact order in which they occurred. As the archive log files contain the entire database operation, there is no need to have access to the source database at this time. This allows GSSync to be used on the secondary server (i.e. the server on which you are writing the target data) instead of on the primary server.

Trade-Offs with GSSync and Archive Log Metadata

Archive Log metadata is ideal in a situation where you do not have access to DataExchange, due to either cost or compatibility issues (such as the database engine version), and you still need to replicate all changes very rapidly. Because the archive logs contain the data records themselves, replicating via the logs does not require access to the original source data, so it can easily be done off-line, in a completely standalone environment. The archive logs can easily be sent from one site to another for processing on a separate server, and it is ideal for monitoring a limited subset of your database for changes. It is also ideal for data sets that never see record deletes, such as history files.

Note that you should almost always include the record's Position Value in the export to serve as a primary key. Also, due to the re-use of the Position Value from one record delete to another insert, note that you **MUST** do each of these operations in order. Due to this limitation, exporting to a comma-delimited or UNF file is not viable, since the resulting split output files (one containing records, the other containing deleted records) will not provide the proper ordering. However, it **MAY** be possible to process deletes first, and then process inserts/updates, as long as the Position Value is used for the unique key. This may leave extra records in the file that should have been deleted, but will never result in a valid record being deleted inadvertently. Further, exporting to a Btrieve file will also not guarantee that the records are sent to the same physical location, so inserts and subsequent updates may impact the wrong records. Based on these restrictions, you should probably replicate using Archive Log data to SQL Script and ODBC target databases only. As you can expect, Actian and Goldstar Software provide no guarantee that this function will even work, let alone work well.

One other important note is required regarding the Archive Logs. The Btrieve interface defines a "partial record update", known by developers as an "Update Chunk" or operation code 53. This operation can be used to modify large records (i.e. those over 60K or so that don't fit into the data buffer) or can be used to efficiently modify small parts of even moderate records. At this time, the GSSync replication code is unable to

handle these chunk operations because they do NOT contain a complete image of the record being changed. This deficiency may or may not be addressed in a future version, due mainly to the limited usefulness of the Archive Log metadata option. If you are looking to use Archive Log Metadata and need tracking of Update Chunk operations, please contact Goldstar Software.

In short, Archive Log replication is possible, but it is not recommended unless you have no other options and you understand the exact benefits and limitations that this solution provides.

GSSync Replication Using GSSync Metadata

The final mechanism for storing metadata, if you cannot rely on the metadata provided by the database engine from Archive Logging or from DataExchange, is to track the changes through GSSync's own proprietary metadata format. This provides the flexibility to avoid the pitfalls of the other replication environments, at the expense of performance, especially for large data sets.

Replication Metadata

If we need to maintain the metadata external to the application and database engine, how do we find out when changes occur? The short answer is that we can't! The database engine does NOT keep track of what data has changed for us in any way! This is why the PDC and archive log metadata is so important, because it can provide some sort of change tracking. Obviously, we need to overcome this deficiency and come up with a way to maintain our own metadata instead.

With no way to know what has changed, GSSync (when using GSSync metadata) is forced to read *every record* in the database and build a corresponding metadata record for each one, akin to the PDC records that DataExchange creates. However, with no way to easily maintain the metadata records only when changes occur, it utilizes a simple, brute-force mechanism to update the metadata -- it re-reads EVERY record in the database on each replication cycle and compares it with the stored metadata. If a change is detected, then GSSync knows that the record is no longer the same, and it can replicate the data accordingly.

The metadata records used by GSSync contain the following information:

- The current timestamp
- The record's system data value (which may be 0 if system data is disabled)
- The record's position value
- The record's internal usage count
- A processed flag

Building Metadata

Building initial metadata for GSSync is done using the GSCREATE metadata build option. This option instructs GSSync to create a brand new metadata file first with no records in it. Then, it goes through the entire source Btrieve file, reading record by record from the beginning to the end. For each record found, GSSync creates a metadata record with the information noted above and writes it to the metadata file.

The process of building the metadata for the first time obviously will take some time on a larger database. You can export every data record at the same time, or you can skip the export, depending on your needs.

Maintaining Metadata

As previously mentioned, the engine does not automatically maintain the metadata, which would be helpful in finding out that records have changed. Instead, since there is no separate metadata maintenance process, we use a single replication pass that handles both the replication and metadata maintenance at the same time.

Replication Process

As the replication process starts running, it must read every record in the source Btrieve file, in order by the Btrieve key specified. For each record, the System Data value and the Position is also found. This gives us the information needed to find the associated metadata record. Note that there are TWO different options – and you must instruct GSSync which key to use to identify the data (though we recommend using the System Data value whenever possible).

If the metadata record is not found, then we can surmise that the record was recently inserted. To process an insert, GSSync creates a new metadata record with all of the information from the current record, inserts it into the metadata table, and then exports the newly inserted record normally.

If the metadata record is found, the record usage count (an internal value inside the database file that is rarely exposed) from the current record is compared to that saved within the metadata record. If they are the same, then this is an existing record that has not changed, and we can ignore it. If the usage count has changed, then the record has been updated one or more times. To process an update, the metadata record is updated with the new usage count and last changed timestamp, and the updated data record is exported as expected.

The replication process then reads the next Btrieve record and loops until all records currently in the file have been handled.

What is missing in this process is any way to catch deleted records, since they no longer exist. In order to handle this inevitability, GSSync uses a second pass through the metadata file to look for records that did NOT get updated by the first pass. These

metadata records are then processed as delete operations, and the metadata record is removed from the system. When all deletes have been processed, the pass is complete.

Trade-offs with GSSync and GSSync Metadata

The most obvious drawback to GSSync metadata is that every single record of the source database must be read during every replication cycle, and every single metadata record must be touched in some way or another. For very large databases, the sheer amount of work this imposes on the database engine system will be far too restrictive -- it will make GSSync replication a non-viable solution.

On the positive side, though, GSSync replication can be done on *any* database, with *any* version files and *any* database engine (although it really works best with System Data available on PSQL7 or higher), and without spending any money or efforts on another solution. It is a perfectly viable solution for replicating small databases, or even small portions of large databases, to an external SQL database. In fact, it is an ideal solution for such tasks as replicating an in-house inventory database to a web server for further user-side queries. This is especially useful when direct database access is not available, since GSSync can export scripts that can be uploaded into an FTP directory and run automatically by the web server's database engine.

Choosing a Replication Solution

At this time, there is no one clear winner in the replication world. As you can see, each solution has its advantages and disadvantages.

Users of Pervasive PSQL v9 and above are in the best shape for replication because you can use any of the supported options with ease, whereas users of older databases might be somewhat limited.

Small databases can probably be replicated with any of the solutions, but the GSSync metadata would likely be the most efficient way to collect the changes and replicate them, while avoiding the pitfalls of archive logs.

Very large databases will usually require the use of DataExchange to use the PDC metadata so that replication can run in somewhat real time, but the decision of using DataExchange replication or GSSync replication will then hinge on the target database environment.

If none of the other options is appealing to you, you can consider using the archive logs, which is the only option that bundles the entire data block into the log file, allowing you to package up all of the changes into a single, simple process.

GSSync is all about the options. For obvious reasons, YOU know your environment best. YOU know your data best. YOU know your data definitions best. YOU know your overall

needs best. As a result, YOU need to evaluate all the options available and select the best options to satisfy YOUR particular replication needs.

Chapter 3: Running GSSync

This chapter covers the basic background information on using GSSync.

Running GSSync

Launching the GSSync application must be done from the Windows command line interface. Attempting to run GSSync by double-clicking from a graphical (i.e. GUI) window will result in a quick flash of a black screen and no noticeable results as the help screen scrolls by and then closes.

You can start your own command line interface window by selecting Start/Run and entering CMD into the Run dialog box. From the command box, simply enter "GSSYNC" and you will see the default command syntax screen, as shown here:

```
GSSync Version 2.10: 11/29 (C)2022 Goldstar Software Inc.
```

```
The command syntax is as follows:
```

```
GSSYNC /CF<ConfigFile> [Options] or  
GSSYNC [Options]
```

```
Where [Options] includes the following:
```

```
/AD#: Appends the Datetime to the export file name in this format:
```

```
0 = None (Default)  
1 = _MMDD  
2 = _YYYYMMDD  
3 = _YYYYMMDD_HHMM  
4 = _YYYYMMDD_HHMMSS
```

```
/BE#: Sets the Boolean Export format to one of the following.
```

```
0 = 1 or 0 (Default)  
1 = TRUE or FALSE  
2 = YES or NO
```

```
/BF<Database>: Read this Btrieve File.
```

```
/BK#: Read the Btrieve File via this key (Default=Physical).
```

```
/CE#: Sets the CSV Export Format to one of the following.
```

```
0 = Do not quote any fields (Raw Comma File).  
1 = Quote all fields (Comma-Quote File).  
2 = Quote only String fields.  
10= Add to above to remove quotes from Header row.
```

```
/DD<Database>: Read file definitions from the given the dictionary directory.
```

```
/DE#: Sets the Date Export format to one of the following.
```

```
0 = mm/dd/yyyy (Default)  
1 = dd.mm.yyyy  
2 = yyyy-mm-dd  
3 = yyyyymmdd  
10= {d 'yyyy-mm-dd'}  
11= TO_DATE(Oracle)  
12= DATE 'yyyy-mm-dd' (Vector)
```

```
/DF#: Sets the Date Field Filter to one of the following.
```

```
0 = Export bad dates as-is.  
1 = Convert bad dates to NULL.  
2 = Convert bad dates to 01/01/1901.  
3 = Convert bad dates to 01/01/1980.  
10= Add to above to treat 00/00/0000 dates as bad.
```

```
/DL#: Sets Display Language (0=ENG, 1=ESP, 2=ITA, 3=FRA, 4=GER).
```

```
/DN<Database>: Read file definitions from this database name (PSQLv9 and up).
```

/DT<Table>: Use this Table name from the dictionary. (Requires /DN or /DD.)
 /DX<XMLFile>: Read file definitions from the given XML file.
 /ED#: 1 = Export the Delta Flag (IUD) with export. (Valid on CSV/UNF Only.)
 /EF#: 1 = Export the Position Value with the data. 0=No Export.
 /ES#: 0 = Don't Export System Data Value. See /TE for valid options.
 /ET<timestamp>: Ending timestamp for PDC/ArchLog sync optimization.
 (Only process data changed BEFORE this timestamp, as yyyyymmddhhmmss.)
 /EU#: 0 = Don't Export Update System Data Value. See /TE for valid options.
 /ID#: 1 = Ignore deleted records (for use with a data warehouse).
 /LA<Filename>: Appends to the output log file (Default is GSSYNC.LOG).
 /LF<Filename>: Creates a new output log file (Default is GSSYNC.LOG).
 /LL#: Specifies the Log Level to one of the following.
 1 = Minimal Logging
 2 = Error Messages and All Above
 3 = Warning Messages and All Above (Default)
 4 = Informational Messages and All Above
 5 = Debug Messages and All Above
 /LZ#: 1 = Retain Leading Zeroes in NUMERIC/DECIMAL Fields.
 /MA<ArchiveLog>: Obtain record metadata from the given archive log file.
 /MC<MetadataFile>: Create record metadata into a new GSSync metadata file.
 (This option implies a full export (/MN).)
 /MG<MetadataFile>: Export data by POSITION based on the given metadata file.
 /MN: Perform a full export (with No metadata).
 /MP<PDCFile>: Obtain record metadata from the given PDC (DataExchange) table.
 /MS<MetadataFile>: Export data by SYSKEY Value based on the metadata file.
 /NF#: Sets the export NUMERIC Field Filter to one of the following:
 0 = No Filtering (Default)
 1 = Change bad digits to 0
 2 = If bad digits, change value to 0
 3 = If bad digits, change value to NULL
 4 = If bad digits, change value to 9's
 5 = If bad digits, change value to -9's
 /OB<Filename>: Specifies output filename for the target Btrieve file.
 /OC<Filename>: Specifies output filename for the exported data in CSV format.
 /OD<Filename>: Specifies output filename for the deleted data keys.
 /OF<Filename>: Specifies the output SQL file forked with ODBC output.
 /OH#: Enables or Disables the Header Row for a CSV Export. (Default=1)
 /OJ<JSONFile>: Specifies the output JSON file for the exported data.
 /ON: Specifies that NO export output should be created (Build metadata only).
 /OQ<Filename>: Specifies output filename for the binary Queue file.
 /OR<Filename>: Read the ODBC Connection String from this file.
 /OS<Filename>: Specifies the output SQL file for the exported data.
 /OU<Filename>: Specifies output filename for the exported data in UNF format.
 /OV<Filename>: Specifies output filename for exported data in VWLOAD format.
 /OW<owner> specifies the Btrieve file owner name.
 /OX<XMLFile>: Specifies the output XML file for the exported data.
 /PA<password> specifies a Password for the DDF's.
 /PD#: Prepends the Datetime to the export file name (see /AD for formats).
 /PO#: Specifies # Parallel Output Files (Default 1).
 /RF"FilterExpr": Filter out records not returning TRUE for this expression.
 /SE#: Sets the timeStamp Export format to one of the following.
 0 = yyyy-mm-dd hh:mm:ss.ff (Default)
 1 = yyyyymmddhhmmss
 2 = 'yyyy-mm-dd hh:mm:ss.ff'
 10= {ts 'yyyy-mm-dd hh:mm:ss.ff'}
 11= TO_DATE(Oracle)
 12= TIMESTAMP 'yyyy-mm-dd hh:mm:ss.ff' (Vector)
 /SF#: Enables the export String Filter to the sum of the following:
 0 = No Filtering (Default)
 1 = Change CR/LF to Space
 2 = Change 0x00 to Space
 4 = Change Control Characters to Space


```

8 = Clear High Bit on Every Character
16 = Change Text to ALL UPPERCASE
32 = Eliminate Trailing Blanks from CHAR and VARCHAR Fields
64 = Change Vertical Bar to Space (needed for VWL Exports)
128= Change Double Quote to Space
256= Change Single Quote to Space
512= Change Backslash to Space
/ST<timestamp>: Starting timestamp for PDC/ArchLog sync optimization.
(Only process data changed AFTER this timestamp, as yyyyymmddhhmmss.)
/TE#: Exports the Export TimeStamp with the data in the given format.
0 = Do not send the Export TimeStamp
1 = Pervasive SeptaSecond Format (Default)
2 = Unix Format (Seconds Since 1/1/70)
3 = UTC Timestamp Format {ts 'yyyy-mm-dd hh:mm:ss.ff'}
4 = Local Timestamp Format {ts 'yyyy-mm-dd hh:mm:ss.ff'}
5 = UTC US String Format 'mm/dd/yyyy hh:mm:ss.ff'
6 = Local US String Format 'mm/dd/yyyy hh:mm:ss.ff'
7 = UTC European String Format 'dd.mm.yyyy hh:mm:ss.ff'
8 = Local European String Format 'dd.mm.yyyy hh:mm:ss.ff'
9 = UTC Oracle TO_TIMESTAMP Format
10= Local Oracle TO_TIMESTAMP
11= UTC Vector TIMESTAMP Format
12= Local Vector TIMESTAMP Format
13= UTC Big Numeric Format yyyyymmddhhmmssff
14= Local Big Numeric Format yyyyymmddhhmmssff
15= UTC YearFirst String Format yyyy-mm-dd hh:mm:ss.ff
16= Local YearFirst String Format yyyy-mm-dd hh:mm:ss.ff
17= UTC Quoted YearFirst String 'yyyy-mm-dd hh:mm:ss.ff'
18= Local Quoted YearFirst String 'yyyy-mm-dd hh:mm:ss.ff'
/TF#: Sets the export Time Format to one of the following.
0 = hh:mm:ss.ff (Default)
1 = hhmmss
2 = 'hh:mm:ss.ff'
10= {t 'hh:mm:ss.ff'}
11= TO_DATE(Oracle)
12= TIME 'hh:mm:ss.ff' (Vector)
/TL#: Exports the Last Change TimeStamp (PDC/ArchLog Only).
0 = Do not send the Last Change TimeStamp
1 = Pervasive SeptaSecond Format (Default)
[See /TE switch (above) for values 2 through 18.]
/US<username> specifies a User Name for the DDF's (Default=Master).
/VD<text>: Include only records including this text in the variant.
/VE#: 1 = Enable Variant record handling. 0=Disable variants.
/VL#: Specify the Variant field Length.
/VO#: Specify the Variant field Offset.
/VT#: Specify the Variant field data Type.
/XF#: Specifies the XML format: 0=Raw (Default), 1=Fields.

```

Whew! Now THAT'S a mouthful! Don't worry about how complicated this might look -- we're actually going to boil this down to a much simpler process with a configuration file that can provide most of these options for you.

Using the Configuration File

A configuration file is a special file that contains all of the configuration options for a given GSSync process in XML format. This XML file can be created with any text editor, such as Notepad or WordPad, but you must follow the XML standard layout.

A sample configuration file is provided with GSSync called GSSYNC.CFG. This XML file contains all of the various options, and is complete with in-line comments that serve as documentation for the file. Of course, all comments are optional and can be removed, but we do not recommend doing this, just in case you need to make changes later on.

The most common command line for GSSync will be very simple and look like this:

```
GSSYNC /CFGSSYNC.CFG
```

This command line will execute the GSSYNC.EXE program and cause it to read the configuration file called GSSYNC.CFG (in the current directory) and process the data based on the given settings in that configuration file. When completed, the application will exit and return you to the command line.

Using the Configuration Database

The configuration database is feature to be defined in a future version.

Using the Command Line Options

If you would like to review the syntax and option list for the GSSync application, just run GSSYNC.EXE by itself to see the help screen (shown above) which lists all of the command line options available to you. (You may need to either pipe the output through the MORE utility or define a command prompt that has a larger screen buffer size so that you can scroll back through the text.) These command line options will be covered individually as we review the various features of GSSync.

Specifying Options at the Command Line

If you would like to provide a specific option for GSSync, simply include it on the command line. If there is a parameter that goes with that option, such as a number or name, then you should include it immediately following the option (with no intervening space). If you have multiple command line options to provide, separate each by one or more blank spaces. Each option should be on the command line one time only. If you erroneously include an option two times, the SECOND occurrence of the setting will take precedence.

Note that you can utilize command line options exclusively for many of the basic replication and exporting functions, if you would like to avoid creating and using a configuration file. The only functions that cannot be controlled from the command line are related to the SQL and ODBC exports, which require some lengthy options that would not fit on a command line, and the IgnoreFieldList.

Overriding Configuration File Settings

In some cases, you may have a configuration file that specifies a set of configuration values, but you want to run a test process with different settings. One way to do this is to edit the configuration file manually, then run the process, then change the

configuration file back. However, this extra editing can subject the configuration file to typos, which can cause other problems.

The better solution is to provide the Configuration file FIRST on the command line, and then supply any overriding configuration settings directly on the command line following that parameter. Command line parameters are parsed in the order that they appear on the command line. Therefore, if you include any command-line settings BEFORE the configuration file, they will often be ignored. If an option normally requires some text (like a file name) and you omit the name, then the value from the configuration file will be used.

Some of the command line parameters are used to provide BOTH additional data AND a functional flag to change the way GSSync operates. For example, the /MC option includes a metadata file name immediately following the parameter, but it ALSO serves as a flag to force GSSync to perform a metadata “create” operation. If the metadata file name provided in the configuration file is correct, then you can specify only /MC (with no file name after it) to force the flag only. This should prevent typing errors and simplify testing of the replication environment.

Checking GSSync Return Codes

When GSSync completes, it completes with a status code that indicates if the run was successful or if an error was seen. This status code is logged in the log file on the last line, but it is also returned to the command processor as the return value from GSSync, making it possible to detect problems in a batch file and provide proactive error reporting and notification. You can check this value by evaluating the expression ERRORLEVEL from the batch file or command line like this:

```
IF ERRORLEVEL 1 GOTO <label>
```

This will go to the label in the batch file if ANY non-zero error code is found. (Note that the ERRORLEVEL command will branch on ANY value that is equal to or higher than the provided value.)

If you wish to check for a specific status code and take appropriate action, then you can use the %ERRORLEVEL% environment variable instead:

```
IF "%ERRORLEVEL%" == "1000" GOTO <label>
```

This line compares the ERRORLEVEL variable to the value 1000 and branches if they are equal.

The following Return Codes are currently defined by GSSync:

Return Code	Description
0	Success -- No Error
1000	Out of Memory

1001	Licensing Error
1002	Bad Command Line
1003	No Work To Do (Can be bad command line switches)
1004	Unsupported Option Selected
1005	Internal Coding Error
1006	Missing Required System Data
1007	Data Definitions Not Provided
1008	Unsupported Variant Record Definition
1100	Cannot Open Export File
1101	Cannot Write Export File
1102	Cannot Open Log File
1103	Cannot Write Log File
1104	Cannot Open Delete File
1105	Cannot Write Delete File
1106	Cannot Open Connection File
1107	Cannot Read Connection File
1108	Cannot Open Archive File
1109	Cannot Read Archive File
1110	Cannot Access ODBC Database
1111	Cannot Create Queue File
1112	Cannot Write Queue File
1113	Cannot Access Btrieve Target File
1200	Invalid SQL Statement

Other status codes may be added at any time.

Scheduling GSSync

GSSync includes a scheduler component written in PowerShell that is discussed in depth in **Chapter 10: The GSSync Scheduler** on page 109. The text in this section is provided in case you elect not to use the provided Scheduler or if you have additional issues.

It is possible to schedule GSSync by including the application in a batch file (or even in multiple batch files), and then by running those batch files from a built-in scheduler such as AT, Task Scheduler, or from an external schedule like AutoTask 2000.

If you do schedule a task, please remember that GSSync is licensed by the user account from which the license was entered. So, if you installed the license (with GSLicKey) as the Administrator user, then you should make sure that your scheduled task is also running as Administrator to avoid running as an evaluation license. If you must run as another user account, please use GSLicKey to enter the license under that account. If you need to run as the SYSTEM account user, please contact Goldstar Software for additional help in configuring this option.

When scheduling a recurring process with GSSync, you should be prepared to check the return value (in ERRORLEVEL, as described above) and provide some sort of intervention

if the process fails. You should also specify unique log files for each process you are scheduling, to facilitate a review of those processes in the event of a failure.

If you are running a large number of GSSync processes, you may also wish to run them concurrently by creating multiple batch files and then scheduling a series of GSSync processes in each batch file. By doing this directly on a server with multiple CPU's and a local PSQL/Zen database engine, you can see extensive parallelism, higher CPU utilization, and reduced cycle times (to run the entire set of processes). This is an effective way to leverage a multi-core server to its best potential.

You can also run GSSync from a workstation, but performance will likely be a bit slower than local server performance, due to the extra latency in each database request that must be made over the network wire.

Chapter 4: Importing Data into GSSync

One of the most important aspects of any replication project is to identify the source data -- that information which you need to replicate. GSSync is designed to be quite flexible in specifying the source data and accompanying metadata, allowing you to read raw Btrieve files as pure blobs without going through an ODBC driver. This provides a measure of performance that is unequalled by any other SQL-based tool out there.

GSSync can also leverage source metadata, if it is available, in either Data Dictionary or XML formats. As discussed in Chapter 2, we need source metadata to describe the database layout, or structure, of our source data, so that the application can understand each Btrieve blob. To avoid confusion between replication metadata and the metadata that is used to define the data structure, we have opted to use the term "data definition" to describe the latter information.

GSSync needs to know as much information about the source data as it can, so there are several options dedicated to this capability. All of the settings in this chapter refer to configuration settings found in the "SourceOptions" group in the XML file.

Specifying the Source Data

In its most raw elemental form, the source data is defined purely as an individual Btrieve database file. Btrieve files are individual database file names, which can be located in any directory on a database server.

When defining the source data, we need (at minimum) to provide the Btrieve file pathname. For databases that are protected by an owner name, you may also need to specify the data file owner name for GSSync. The only other attribute that may be required is the Btrieve key number to use when scanning the file for changes. With these three core attributes, we can effect a complete Btrieve-to-Btrieve replication solution with GSSync!

Btrieve File Name

Specifying the Btrieve file name will tell GSSync which database file to read during the replication process. You should use this option if you are replicating from a raw Btrieve file and you do not have data definitions available, or if you are using an XML file definition. You can also use this option to link a second Btrieve file loosely to an existing data definition. This can be helpful for some environments that leverage a single "history table" in the data dictionary, but which keep separate Btrieve files for each year of the history data.

Please note that newer Btrieve engines support path names and file names with spaces in them. GSSync will also support this option, should you choose to use it. If you do use this configuration, please verify in the Microkernel Router configuration that the "Use

Embedded Spaces" setting is checked. (It is checked by default in newer engines.) If this setting is unchecked and you use spaces in file names, then GSSync will not be able to access the data file.

The Btrieve File Name should be specified in the XML configuration file:

```
<BtrieveFile>C:\PVSW\DEMODATA\PERSON.MKD</BtrieveFile>
```

From the command line, the following switch should be specified:

```
/BF<Filename>
```

Included after the switch (with no space following it) should be the filename or pathname from which you want to read the source data. As with most switches, including the /BF switch AFTER the configuration file on the command line will override the configuration file setting with the one specified on the command line.

The use of absolute pathnames (as in the above example) is typically not recommended, since changes to the environment can require a substantial effort at fixing up the configuration files. Instead, you should use a simple file name or relative pathnames, such as "DATA.BTR", "..\DATA.BTR", or ".\FOLDER\DATA.BTR", which can then easily be moved to another folder if needed.

Btrieve Owner Name

If the Btrieve file has been protected by an owner name, either to protect against improper access or to provide data encryption capabilities for the file stored on disk, then you will need to provide the Btrieve file's owner name to GSSync. Failure to provide the owner name, or providing the incorrect owner name, will result in a Status 51 and a termination of the GSSync process.

Standard owner names are 8 bytes in length, but newer versions support a "long owner name" of 24 bytes in length, as well as a 32-byte owner name in v13 and newer. GSSync supports all of these formats for maximum compatibility with your application.

As the owner name is in essence the file's "password", you may not wish to include this directly in the configuration file or in a batch file which is accessible to all users. Please follow all necessary security precautions and protect the access to these critical files whenever necessary.

The Btrieve Owner Name should be specified in the XML configuration file:

```
<BtrieveOwner>owner</BtrieveOwner>
```

From the command line, the following switch should be specified:

```
/OW<ownername>
```


The owner name for the Btrieve file should be Included after the switch (with no space following it). If your configuration file specifies the Btrieve Owner Name and you wish to change it temporarily, include the /OW switch on the command line AFTER the configuration file with the desired owner name.

As the Btrieve owner name is provided by the application as a binary block of bytes, it is possible that some applications will use non-ASCII or control characters for an owner name. If you run into this issue, you can specify the owner name as a series of hexadecimal digits by prefixing the values with "0x". For example, to use an owner name of "007" in ASCII, you can use "007" or you can use "0x303037". With this feature, you can specify any of the 256 possible values for each byte in the owner name string. Note that a NULL byte terminates the owner name in any event. If your owner name happens to start with "0x" (i.e. the ASCII zero followed by a lowercase "x"), then you will be required to provide the owner name in hexadecimal format.

Btrieve Key Number

For PDC and Archive Log metadata types, the metadata drives how records are read from the Btrieve file. However, for GSSync metadata, the Btrieve file is traversed directly by the replication engine. This traversal can be done in physical order (which is typically the fastest), in logical order by a given application-defined key, or in order by the System Data key.

Traversing a file in physical order is typically the fastest and safest, because it ignores all of the key information in the file and it returns all possible records, even if the defined keys have Btrieve NULL values defined. To specify a traversal by physical order, use the special key number of -1. Keep in mind that a traversal in physical order can provide a randomly ordered result set that can even change the order of records between successive runs. If you are working with system data, GSSync must perform extra lookup operations to obtain the system data.

Although a bit slower, traversing a file by an application-defined key can ensure that the data is always output in the same order. Further, exporting the data by a known key value can make it possible to determine the point of failure of a downstream import process and make it possible to restart the import (after fixing the problem) after the point of the failure to avoid having to re-run the entire import process. To specify a traversal by a given key, use that key number. One possible problem with application-defined key values is that they may have unexpected attributes, such as NULL values or descending order, that may make working with them a bit more difficult. You may wish to use BUTIL -STAT to list the keys in the Btrieve file and verify their attributes before using those keys.

The fastest mechanism for reading Btrieve files when working with GSSync metadata is to use the System Data key, which is defined inside the Btrieve interface as Key Number

125. Using this key eliminates some additional overhead to read this value for each record. To specify a traversal by the System Data Key, use the special key number of 125. One drawback of this key, though, is that the data can come back in a seemingly random order. (In reality, the system key is generated based on the system datestamp, so records will typically come back in increasing order by the date/time when they were first inserted.)

The Btrieve Key Number should be specified in the XML configuration file:

```
<BtrieveKey>125</BtrieveKey>
```

From the command line, the following switch should be specified:

```
/BK125
```

The key number to use for traversing the Btrieve file should be Included after the switch (with no space following it). If your configuration file specifies the Btrieve Key Number and you wish to change it, include /BK# on the command line AFTER the configuration file with the desired key number.

Specifying the Source Data Structure

Btrieve files by themselves are simply storage containers for Btrieve records, and the Btrieve records themselves are simply "blobs" -- formless sequences of bytes, with no real data structure implied or interpreted by the database itself. The only form or structure is what the application itself puts onto the data when it reads it in.

Obviously, replicating formless blobs to comma-delimited files or to an ODBC target database is somewhat meaningless. As such, we need to define the proper source data structure definition to perform a replication to these targets. The source data definition gives GSSync enough information to interpret the blobs and properly export the data.

We can provide the data definition using a data dictionary. Data dictionary files (DDF's) are used by SQL engine (the SRDE) to interpret data directly for manipulation from native SQL. However, some environments have poorly defined data dictionaries, and this translates to some very convoluted layouts. Other environments have no dictionary provided by the application, making it all but impossible to replicate the data, so we can also use a special XML format to provide the needed metadata.

Reading a DDF by Path

The first and simplest way to provide GSSync with the data definition is to specify the path to the data dictionary files. You should first determine where the DDF files exist, and then provide GSSync with the pathname to those files.

The DDF path can be specified many different ways. The best way to define the path when you are running GSSync from the database directory itself is to use the current

directory, indicated with only a "." as the path. If the DDF's are in a subfolder of the current directory, or in a subfolder of the parent folder, then you should use a relative path (starting with ".\folder" for the current directory or "..\folder" to go one directory up the tree).

If absolutely necessary, a full pathname (starting with "\" or a drive letter "F:\") or a UNC pathname (with "\\server\share\path") can also be provided. However, the use of these latter options is discouraged as it reduces the flexibility of the environment, and a simple server name change can then require changes to hundreds of configuration files.

You must remember that DDF files are Btrieve files too. As such, it is critical that the files be located on a computer with an active database engine. If not, the system will not be able to read the definitions, and the process will fail.

Please note that newer Btrieve engines support path names and file names with spaces in them. If you have this configuration, please verify in the Microkernel Router configuration that the "Use Embedded Spaces" setting is checked.

The DDF Path should be specified in the XML configuration file:

```
<DatabaseDirectory>.</DatabaseDirectory>
```

From the command line, the following switch should be specified:

```
/DD.
```

The path to the DDF files, either in relative, absolute, or UNC format, should be included after the switch (with no space following it). If your configuration file specifies the Database Directory and you wish to change it, include /DD<path> on the command line AFTER the configuration file with the desired path.

Reading a DDF by DBName

For systems running Pervasive PSQL v9 and above, you can also indicate the data definitions by specifying the Named Database to access. The Named Database is the name that shows up in the Control Center when you view the list of databases, and it includes information about the location of the DDF's on the server for you.

While this option provides you with some flexibility for moving the database to a new folder, it ONLY works if you are running your application directly on the database server, and will not work from a client workstation.

The Database Name should be specified in the XML configuration file:

```
<DatabaseName>Demodata</DatabaseName>
```

From the command line, the following switch should be specified:

```
/DNDemodata
```

The database name as defined in the engine should be included after the switch (with no space following it). If your configuration file specifies the Database Name and you wish to change it, include /DN<name> on the command line AFTER the configuration file with the desired name.

Providing DDF Access Rights

Although many databases do not have security enabled, your system may have the additional SQL-level database security enabled. As such, you may need to provide access credentials to access the database structure information. (Note that these credentials are for DDF access, and not for native Btrieve file access, which is controlled by an owner name.)

Access credentials are provided with a login name and a password, both of which are case sensitive. The administrative username for a PSQL/Zen environment is "Master", so this is the default value for this setting, and it rarely needs to be changed. However, if your system administrator does not provide you with the Master password, you may need to set up a different user name and use that instead.

The database credentials should be specified in the XML configuration file:

```
<DatabaseUser>Master</DatabaseUser>
<DatabasePassword>password</DatabaseWord>
```

From the command line, the following switches should be specified:

```
/USMaster
/PApassword
```

The database user name or password, as defined by the database you are trying to open, should be included after the switches (with no space following them). If your configuration file specifies the credentials and you wish to change them, include these switches on the command line AFTER the configuration file with the proper data.

Indicating a Table Definition

Specifying the dictionary is the first half of the providing the data definitions. Because any given dictionary can define the data structures for any number of database files, you also need to tell GSSync which SQL table definition is the correct data definition for this specific Btrieve file.

The database table names can be easily viewed in the Control Center when you open up the database. A table name is either 20 characters in length (for older V1 DDF's) or 128 characters in length (for V2 DDF's), and both formats are handled by GSSync.

The Database Table should be specified in the XML configuration file:

```
<DatabaseTable>Person</DatabaseTable>
```

From the command line, the following switch should be specified:

```
/DTPerson
```

The database table name as defined in the DDF's should be included after the switch (with no space following it). If your configuration file specifies the Database Table Name and you wish to change it, include /DT<name> on the command line AFTER the configuration file with the desired name. If this option is specified and no DDF data definitions are provided, an error will result.

Overriding the Btrieve File Name

Defined inside the data dictionary, along with the table name and field definitions, is information that tells the database engine how to find the Btrieve file that contains the data for this table. If this information is properly stored in the DDF's, then GSSync can also make use of this information to determine the Btrieve File Name automatically for you. If you are specifying a dictionary and table name, you should leave the Btrieve File Name field empty, and the DDF definition will be used to access the file in the correct location.

However, sometimes the DDF's are incorrect, outdated, or you just want to avoid using this information for some other reason. This is not a problem for GSSync! If you specify the Btrieve File Name as indicated on Page 31, then this value will override the file name that is stored in the DDF's.

You should use this feature with some caution. Remember that a Btrieve file by itself has no structure -- only what we put on it at runtime. If you apply the wrong structure to a Btrieve file, the exported data will likely be meaningless.

Specifying a Variant Record Definition

When variant records are stored in a Btrieve file, you need a way to verify that only the proper Btrieve records are processed from the data source. GSSync includes a special ability (variant record handling) to examine each record encountered and ONLY process those insert and update operations for records that include the indicated value at the indicated position. For DELETE operations, GSSync will process every record, since there is no record type field available.

As of v1.20, GSSync provides the ability to specify a single variant record type using a single record type field. If you have a need for a more complicated way to specify variant record types, please contact Goldstar Software. Otherwise, you may be able to leverage the record filter expression (which works with only numeric values) instead.

If you need to handle variant record types, you first need to determine the field that tells the application which record type is which -- called the variant type field. Sometimes, this will be a field called "RecordType" or something similar, but this is not a

requirement. You may have to search through the data in the Control Center or contact the developer to determine the exact field that indicates the record type.

Once you know the field that specifies the type, you now need to obtain information about that field, namely the field offset, data type, and length. You can easily find this data using the Goldstar Software utility FPrint, which will show you these details very easily. If you don't have access to this tool, then you can do the same thing with a SQL query like this:

```
SELECT * FROM X$Field INNER JOIN X$File ON (Xf$Id = Xe$File)
WHERE RTRIM(Xf$Name) = 'tablename' AND RTRIM(Xe$Name) = 'fieldname'
```

Of course, be sure to use your own table name and field name. From the resulting data set, note the Xe\$Datatype field (which will be the Field Type value), the Xe\$Offset field (which will be the Field Offset value) and the Xe\$Len field (which will be the Field Length value), as we will need these to define the field to GSSync.

Once you have the variant field details, you can configure your system to handle these variants. First, you should enable variant handling in the XML configuration file:

```
<VariantRecordEnabled>1</VariantRecordEnabled>
```

From the command line, the following switch should be specified:

```
/VE1
```

Included after the switch (with no space following it) should be either a 0 (disable variant handling) or 1 (enable variant handling). If you include the switch with no value after it, 1 is assumed and variants are handled. If your configuration file specifies the VariantRecordEnabled flag and you wish to change it, include /VE# on the command line AFTER the configuration file with the desired value.

With variants enabled, you will also need to tell GSSync where to find the variant field, as it was determined using the process above. You can do specify all of these values in the XML configuration file using these lines:

```
<VariantRecordFieldOffset>17</VariantRecordFieldOffset>
<VariantRecordFieldType>0</VariantRecordFieldType>
<VariantRecordFieldLength>1</VariantRecordFieldLength>
```

You can also use command line switches, as expected:

```
/VO17
/VT0
/VL1
```

Include the proper value from the variant field definition after each switch. In this example, we are specifying a string field (type 0) at offset 17 with a length of 1 byte.

The VariantRecordFieldOffset must be located within the first 65000 bytes of the record. Attempting to specify a higher value will result in an error. Any variable length database records that do not include the variant record field (i.e. the record length is shorter than the specified VariantRecordFieldOffset) will be processed, and NOT skipped.

The VariantRecordFieldType field should indicate the Btrieve field type that should be checked against the filter. String and integer fields are the most common types used for variant record type data. Because of this, not every Btrieve field type is supported here, and you should only use the following data types (and type values):

Type Code	Btrieve Data Type
0	String (Space Padded)
1	Signed Integer (Valid Lengths are 1, 2, 4, and 8)
2	Floating Point Value (Valid Lengths are 4 and 8)
10	LString (Length-Prefixed String)
11	ZString (Null-Terminated/Padded String)
14	Unsigned Integer (Valid Lengths are 1, 2, 4, and 8)
15	Identity/AutoInc Field (Valid Lengths are 1, 2, 4, and 8)

Support for additional Btrieve data types may be added in the future. If you require a specific variant data type for your GSSync environment, please contact Goldstar Software to request that type to be added to the next release.

The VariantRecordFieldLength field should indicate the length of the Btrieve field that you are attempting to match, up to a maximum of 255 bytes. For example, integer values can be stored in one of four valid lengths: 1 byte, 2 bytes, 4 bytes, or 8 bytes. Specifying the proper field length is critical to ensuring that the comparison works properly.

Finally, you also need to specify the variant data that you need to match. You can specify the value in the XML configuration file using this line:

```
<VariantRecordData>A</VariantRecordData>
```

You can also use a command line switch to specify this value, as expected:

```
/VDA
```

In some cases, you may need to alter the value of the VariantRecordFieldLength with some variant data. For example, if the data type is 0 (a space padded string) and we specify a data value of "Hello" and a length of 10, then the actual comparison value is the space-padded string extended to 10 bytes (i.e. "Hello "), which ensures that we do not match the string "HelloThere". For many cases, the data and the data length will match the actual Btrieve field length, but this is NOT a hard requirement. For example, specifying a length of 1 and a value of "H" for the same string field will match string values of "Hello" and "Hi", but not "Bye" or "Good Bye". If you are matching a ZString

field, GSSync will pad the specified string with additional NULL bytes before making the comparison. However, if your application does not properly fill the extra bytes with NULL bytes, then it may cause records to NOT match the filter when they really should match. In this case, you may need to specify the length of the comparison data + 1 (to account for the null terminator) in order to filter the variant records properly.

Reading an XML Definition File

In some databases, you may not have a "good" DDF definition to read in order to determine the table structure, and creating the DDF's may not be a viable option. In other databases, the application developer has purposefully decided NOT to provide a data dictionary, but you still want to replicate via GSSync.

Luckily, there is another way to provide the data file definitions -- via an XML file. When Pervasive Software released PSQL v9.5, they included a new tool called COBOLSCHEMAEXEC that is designed to read an XML file in a "standard" format and then create data dictionaries directly from the XML definition. While this is typically only used for creating "non-relational" constructs for the support of COBOL applications, there are other uses for tools like this. In our case, we are going to appropriate the "standard" XML file format, and then provide some additional extensions to that standard to get some additional capabilities. To start understanding the XML format, you should first review the file COBOLSCHEMAEXEC.XSD, which is provided with the engine installation.

Here is an example XML Definition File showing the core structure and a few fields:

```
<SCHEMAEXEC xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <BTRIEVE FILENAME="Person.mkd"/>
  <MAINTABLE>
    <TABLEDETAILS>
      <TABLE NAME="Person"/>
      <FIELDS>
        <FIELD NAME="ID" Offset="0" Precision="8" Scale="0"
          BtrieveType="Unsigned(8) BINARY" CASESENSITIVE="true"
          NULLABLE="false" />
        <FIELD NAME="First_Name" Offset="9" Precision="16" Scale="0"
          BtrieveType="ZString" CASESENSITIVE="false" NULLABLE="true" />
        <FIELD NAME="Last_Name" Offset="26" Precision="26" Scale="0"
          BtrieveType="ZString" CASESENSITIVE="false" NULLABLE="true" />
      </FIELDS>
      <INDICES>
      </INDICES>
    </TABLEDETAILS>
  </MAINTABLE>
</SCHEMAEXEC>
```

Note that the <FIELD> blocks define each field in the database, including a field name, precision (used for some data types to provide the field length), scale (used to indicate digits to the right of the decimal point for some data types), Btrieve data type, and more. This is exactly the same information that is commonly stored in the data

dictionary files, but this XML file can be built with any text editor (like Notepad) by following a simple template and providing as much information as you have, without trying to learn how to use the COBOLSCHEMAEXEC tool or the DDF Builder application. The XML definition supports table and column names up to 128 characters in length, just like V2 metadata, and the same 1500-field limit applies to each table.

So one question may arise as you are reading this: If you can create an XML file that is compatible with COBOLSCHEMAEXEC for GSSync, and if you can use the COBOLSCHEMAEXEC tool to load the XML file definition into a data dictionary file set, which should you use? Honestly, it doesn't matter -- both are handled equally well by GSSync. If you can load the data and create DDF's, then you can easily access the data from SQL as well as Btrieve, which makes available a host of other features in the database engine, including reporting from any ODBC-compliant report writer. However, this accessibility can also present a security problem if not carefully protected, so you may prefer to leave data definitions in XML format.

GSSync is also mostly compatible with XML files generated by the utility RecordEdit, by Bruce Martin. You can find this tool at <https://sourceforge.net/projects/record-editor/>. Because of this:

- POSITION can be used instead of OFFSET. (Offset is zero-relative, Position is 1-relative.)
- LENGTH can be used interchangeably with PRECISION.
- DECIMAL can be used interchangeably with SCALE.
- TYPE can be used interchangeably with BTRIEVETYPE.

Additionally, several of the custom TYPE definitions used by RecordEditor have been added, with the following type mappings:

- Char -> Char (right justified) -> Character
- Binary Integer->Integer
- Single -> Float(4)
- Double -> Float(8)
- Mainframe Packed Decimal (+ve) -> Decimal
- Mainframe Packed Decimal (comp-3) -> Decimal
- Num (Right Justified zero padded positive) -> Numeric

- Num Assumed Decimal (+ve) -> Numeric
- Fujitsu Zoned Numeric -> Numeric (May change to FJNUMERIC Later)
- Char Null terminated -> Char Null padded -> ZString
- Binary Integer (only +ve) -> Unsigned
- Postive Binary Integer -> Unsigned [This is a typo in RecordEditor v0.92]
- Positive Binary Integer -> Unsigned [For when they fix the typo]
- Num Sign Separate Trailing -> NumericSTS
- Num Sign Separate Leading -> NumericSLS

Note, though, that the header definitions are *very* different, and the data provided in the RecordEditor header is insufficient to use as-is, so you should expect to replace at least the header to make it compatible with GSSync.

The following Btrieve Types and lengths are supported via the XML data definition:

Btrieve Type	Precision	Data Type Description
AutoTstamp		8-byte Integer Holding the Number of Nanoseconds Elapsed Since 1/1/1970
<i>4Float</i>	x	Floating Point value stored as x-byte Decimal Value but with first char="4" and second equal to the number of chars left of the decimal. For example, "42543200" is 54.32
<i>4Float(4)</i>		4Float Value of Length 4
<i>4Float(8)</i>		4Float Value of Length 8
AutoInc	x	x-Byte AutoInc Field (x must be 2, 4, or 8)
AutoInc (2)		2-Byte AutoInc (SMALLIDENTITY) Field
AutoInc (4)		4-Byte AutoInc (IDENTITY) Field
AutoInc (8)		8-Byte AutoInc (BIGIDENTITY) Field
BFloat	x	x-Byte BASIC Float
BFloat(4)		4-Byte BASIC Float (Single)
BFloat(8)		8-Byte BASIC Float (Double)
<i>Binary</i>	x	x-byte hexadecimal string representing each byte in exact order, prefixed with "0x"
Bit		Single Bit Field (Scale=Bit# to Evaluate)
BLOB		True BLOB Field (4-byte Length, 4-Byte Offset to Bytes)
Character	x	Space-padded String of x Bytes
CLOB		True CLOB Field (4-byte Length, 4-Byte Offset to Bytes)

<i>Comp3</i>	x	An alias for the DECIMAL Type of x bytes, included for convenience. Includes 2x-1 digits plus a sign.
<i>Comp6</i>	x	Similar to the DECIMAL type of x bytes, a COMP-6 field has no sign information, so it represents 2x digits.
<i>CTime</i>		4-Byte Integer Holding the Number of Seconds Since 1/1/1970 00:00:00, Exported as TIMESTAMP.
Currency		8-Byte Currency Field
Date	x	If x=4, this is a 4-Byte Btrieve Date Field. <i>If x<>4, see DATE Options Below.</i>
<i>Date(2)</i>		2-byte date field, stored as an integer where the first two character positions are the Year-1980, and the remaining 3 positions are the number of days into the year. For example, 26002 is 2006-01-02.
<i>Date(3)</i>		3-byte YMD Field, stored as one Byte Each for Year-1900, Month, and Day, Exported as DATE.
<i>Date(4)</i>		4-Byte Btrieve Date Field.
<i>Date(6)</i>		6-Byte String in YYMMDD format, Exported as DATE. Years >= 70 are 20th Century.
<i>Date(8)</i>		8-Byte String in YYYYMMDD format, Exported as DATE.
DATETIME		DATETIME Field, Exported as TIMESTAMP.
Decimal	x	COBOL Decimal Type
<i>FJNumeric</i>	x	x-Byte Numeric Value in Fujitsu COBOL Internal Format
<i>Float</i>	x	x-Byte IEEE Float (x must be 4, 6 or 8)
Float(4)		4-Byte Floating Point (Single)
<i>Float(6)</i>		6-Byte Floating Point (Pascal)
Float(8)		8-Byte Floating Point (Double)
GUID		GUID Field
<i>HexBytes</i>	x	Duplicate of <i>Binary</i> type
<i>Integer</i>	x	x-Byte Signed Integer (x must be 1,2,4, or 8)
Integer(1)		1-Byte Signed Integer
Integer(2)		2-Byte Signed Integer
Integer(4)		4-Byte Signed Integer
Integer(8)		8-Byte Signed Integer
Logical	x	x-Byte Logical Value (x must be 1 or 2)
<i>Logical(1)</i>		1-Byte Logical Value
<i>Logical(2)</i>		2-Byte Logical Value
<i>LongDate</i>		4-Byte Integer Holding Date as a Number, e.g. 20,090,422, Exported as DATE.
LString	x	Length-Prefixed String of x Bytes
LVar		LVAR Field

<i>MagicDate0001</i>		4-Byte Integer Holding the Number of Days Since 1/1/0001, Exported as DATE.
<i>MagicDate1901</i>		4-Byte Integer Holding the Number of Days Since 1/1/1901, Exported as DATE.
<i>MagicTime</i>		4-Byte Integer Holding the Number of Seconds Since 00:00:00, Exported as TIME.
Money		Money Data Type
Note		Note Field
Numeric	x	x-Byte Numeric Value
NumericSA	x	COBOL Signed-ASCII Value of x Bytes
NumericSLB		COBOL Signed Leading with Compiler Option -dcb Value
NumericSLS		COBOL Signed Leading Separate Value
NumericSTB		COBOL Signed Trailing with Compiler Option -dcb Value
NumericSTS		COBOL Signed Trailing Separate Value
String	x	Space-padded String of x Bytes
Time		4-Byte Btrieve Time Field
Timestamp		8-Byte Timestamp Field (in Septaseconds)
Timestamp2		8-byte Integer Holding the Number of Nanoseconds Elapsed Since 1/1/1970
Unsigned	x	x-Byte Unsigned Integer (x must be 1, 2, 4, 8)
Unsigned(1)		1-Byte Unsigned Integer
Unsigned(2)		2-Byte Unsigned Integer
Unsigned(4)		4-Byte Unsigned Integer
Unsigned(8)		8-Byte Unsigned Integer
Unsigned(1) Binary		1-Byte Unsigned Integer (Same as above)
Unsigned(2) Binary		2-Byte Unsigned Integer (Same as above)
Unsigned(4) Binary		4-Byte Unsigned Integer (Same as above)
Unsigned(8) Binary		8-Byte Unsigned Integer (Same as above)
<i>VarBinary</i>	x	Duplicate of <i>Binary</i> type
<i>WString</i>	x	Wide Space-padded String of x Bytes
<i>WZString</i>	x	Wide Null-terminated String of x Bytes
ZString	x	Null-Terminated String of x Bytes

Note that for some data types, the Precision field should be used to specify the field length. If you do not set the length properly, the system may be unable to interpret your values properly.

For the COBOL data types (i.e. NUMERIC and DECIMAL), you should also indicate the scale of the number, which indicates how many digits are stored from the right of the decimal point. A scale value of 0 would represent an integer field. GSSync also supports the scale value from the XML definition for INTEGER and UNSIGNED field types, which can be used when a decimal point is implied in an integer field. Note that Actian does NOT support a non-zero scale in the DDF definitions, so if you need this feature, you MUST use an XML definition.

The data types that are shown in **bold** lettering in the chart are considered "standard" PSQL/Zen data types, as defined by the current XSD file as of this writing. If you intend on using your XML file to create DDF's, then you should use these types only.

Several of the data types in the above chart are shown in *italics*. Developers who have been around PSQL/Zen databases for some time may not recognize these data types as standard types -- you are not mistaken. Remember that the GSSync application is accessing the database structure at the Btrieve level (not via SQL calls), so we can also handle and interpret certain unsupported data types very easily. Some older applications may use these non-standard data types, like 6-byte floating-point values or custom date fields. While attempting to access these fields from SQL/ODBC would return unusable values, we can properly define the fields in the XML layout and allow GSSync to access them with its custom exporting module. Note that some of the fields, such as DATE(3) and DATE field with a Precision value of 3 map to the same result. Either is exported as a native DATE field to the target database.

The current GSSync code design is quite expandable to include other custom data types. If you have requirements for an additional data type, please let us know!

If you have DDF's but would like to use the XML feature to access some of the expanded data type support, there are shortcuts to creating the XML files. Goldstar Software has a tool called DDF2XML that can read data dictionary files and build an XML file for a table that is compatible with COBOLSCHEMAEXEC from the schema definition using a single operation. You can then modify the XML structure from there to make any changes that you might need.

The XML Definition File should be specified in the XML configuration file:

```
<XMLFileDefinition>PersonDef.XML</XMLFileDefinition>
```

From the command line, the following switch should be specified:

```
/DXPersonDef.XML
```

Included after the switch (with no space following it) should be the XML definition file name that you wish to read. If your configuration file specifies the XML Definition File and you wish to change it, include /DX<filename> on the command line AFTER the configuration file with the desired name.

Specifying the Metadata

Now that we have the source data specified in the configuration, we must define the metadata options to use for the replication window. The metadata options include the Archive Log Metadata, DataExchange (PDC) Metadata, and GSSync Metadata. See Chapter 2 for more information about selecting the best replication metadata option for your environment.

Specifying Archive Log Metadata

If you are using the Archive Log Metadata, then you will be collecting one or more Archive Log files that will contain the changes captured by the database engine. These changes will be in files known as Archive Log Files, or more succinctly, *archive logs*. Each archive log will contain the sequence of database changes that took place within the environment during that period for which archiving was enabled and since the log was last cleared.

To process this archive log data, you must inform GSSync that it should expect the Archive Log metadata for this process. This is done by setting the Metadata Type in the configuration file to "ARCHIVE".

```
<MetadataType>ARCHIVE</MetadataType>
```

Once the metadata type has been set, you must then inform GSSync which archive log file should be processed. Again, be VERY careful to associate an archive log with the correct file. A mix-up here can create a file structure that is completely mangled, or it could corrupt the target database. Specifying the Archive Log File is done with another configuration setting in the XML file:

```
<ArchiveLogFile>PERSON.LOG</ArchiveLogFile>
```

From the command line, these two settings can be handled by a single switch:

```
/MAPERSON.LOG
```

Included after the switch (with no space following it) should be the archive log file that you wish to read. You can include an entire path, if appropriate. If your configuration file specifies the Archive Log File and you wish to change it, include /MA<filename> on the command line AFTER the configuration file with the desired archive log file name.

Specifying PDC Metadata

If you are using the PDC Metadata, then your activated DataExchange environment is already collecting change requests into the PDC tables. You now need to link up the PDC metadata, which contains the last updated timestamps for each record in the file, with the original Btrieve file.

To process PDC metadata, you must inform GSSync that it should expect the PDC metadata for this process. This is done by setting the Metadata Type in the configuration file to "PDC".

```
<MetadataType>PDC</MetadataType>
```

Once the metadata type has been set, you must then inform GSSync which PDC file should be processed. Again, be VERY careful to always associate a PDC file with the correct Btrieve file. A mix-up here can replicate the wrong records, or it could corrupt the target database. Specifying the PDC File is done with another configuration setting:

```
<PDCMetadataFile>DX_DATA\PDCPERSON.MKD</PDCMetadataFile>
```

From the command line, these two settings can be handled by a single switch:

```
/MPDXDATA\PDCPERSON.MKD
```

The PDC file that you wish to read should be included after the switch (with no space following it). You can include an entire path, absolute or relative, if appropriate. If your configuration file specifies the PDC File and you wish to change it, include /MP<filename> on the command line AFTER the configuration file with the desired PDC file name.

Indicating Starting and Ending Timestamp

When using PDC metadata, the database engine and change capture routines maintain the Last Modified Date for every record in the table. Based on this information, we can simplify our exporting work if merely skip all records that have NOT changed since our last replication window. We do this by specifying the Starting Timestamp for this replication cycle, and ONLY those records that have changed since this time will be replicated. We can also optionally specify the ending timestamp, if we want to try to ensure that related data changes are always kept together.

With this option, we can perform a Btrieve-to-Any replication efficiently and effectively, using the PDC metadata only. We can then build this into a script that would run periodically (say, every 5 minutes) to push all changes to the target database. As it would be almost impossible to edit the configuration file for each process run, these settings are ONLY available from the command line, using the following switches:

```
/STyyyyymmddhhmmss  
/ETyyyyymmddhhmmss
```

For each Timestamp value, you can provide as much of the timestamp as you wish to provide, but you MUST start from the left side. For example, if your switch is "/ST2009", then all changes since 1/1/2009 @ 00:00:00:00 will be processed. If you wish to export all record changes for a specific month, provide the year and month only, as in "/ST200904", and indicate the NEXT month for the Ending Timestamp value, as in "/ET200905". When a record is found that meets or exceeds the ending time, i.e. records changing in May 2009 or later, the process will stop looking for data. If you are exporting changes on a daily basis for replication across a WAN link, consider putting the year, month, and day in there. To replicate once per hour, consider a timestamp like "2009042908", which will replicate all records changed after 8AM on 4/29/2009.

You should also keep in mind that the replication process does take a certain amount of time to process. For best results when replicating, you should specify the STARTING timestamp of the *previous replication* for the /ST option, and the STARTING timestamp of the *current replication* for the /ET option. If you use the time that the previous replication completed, you may miss a small number of records that were changed

during that replication cycle. You may also find it helpful to include a small "fudge factor", or a few extra seconds, to provide just a bit over overlap, capturing all changes, at the risk of replicating some records multiple times.

Building GSSync Metadata

If you are using the GSSync Metadata, the first step is to perform a full export, which reads every record in the source data set, and create the starting metadata file. This will determine the current "version" of each record so that GSSync can detect later changes in the Btrieve file.

To generate the GSSync metadata for the first time for a table, you instruct GSSync to create the GSSync metadata for the original data set by setting the Metadata Type in the configuration file to "GSCREATE".

```
<MetadataType>GSCREATE</MetadataType>
```

Once the metadata type has been set, you must then inform GSSync which metadata file should be created. This process will be creating the file, so it does not need to exist. If the file does exist, it will be overwritten with no warning. Specifying the GSSync File is done with another configuration setting:

```
<GSMetadataFile>PERSON.GS</GSMetadataFile>
```

From the command line, these two settings can be handled by a single switch:

```
/MCPERSON.GS
```

The GSSync metadata file name that you wish to create should be included after the switch (with no space following it). You can include an entire path, if appropriate. If your configuration file specifies the GSSync File and you wish to change it, include /MC<filename> on the command line AFTER the configuration file with the desired file name.

In some cases, you may wish to run the GSCREATE option without actually using the exported data. If this situation arises, set the Export Option to NONE or provide the /ON switch. This will allow GSSync to skip the record export processing, which will allow it to create the GSSYNC metadata records very rapidly.

Replicating with GSSync Metadata

Once you have created the initial GSSync metadata for a table, you have all of the information that you need to detect and replicate changes. GSSync will read every record, check to see if it has been changed, and if so, export it. When the first scan completes, GSSync will then search for any "old" records that still exist in the metadata table, processing them as record deletes.

However, things can get a bit more complicated here, because there are actually two different record locator values that GSSync can employ to track changes. The first value is the *record position*. A record position is a simple 4-byte value that indicates the exact logical location in the data file at which the record exists. Like someone's house address, the record which lives at any position can completely change. If one record leaves (via a delete operation) and another record moves into the same slot (via an insert operation), GSSync will see this as an *updated* record, not as a delete and insert combination. (The same can occur if the file is rebuilt – as each record will get shuffled into a new record position.) Because of these issues, we normally do not recommend using the record position value to track changes in any environments – but it is an available option simply because EVERY Btrieve file format supports record positions, and EVERY Btrieve record will have a unique value at any point in time.

The second possible record locator value is the *System Data* value. As previously indicated, PSQL7 and above support adding a hidden, 8-byte field and index to any database file called System Data. This value is assigned when a record is created, and it is static for the life of the record. This makes it an ideal candidate for replication tracking, which explains why DataExchange uses it for its own metadata tracking. Even better, whereas a file rebuild operation can completely shuffle the record position values, the Rebuild Utility can actually retain the existing System Data values during a rebuild, reducing the load on the replication system. As you can see, we strongly recommend using the 8-byte System Data value for your replication record locator.

To use GSSync to replicate based on the GSSync metadata for a table using the record position value as a primary key, you set the Metadata Type in the configuration file to "GSSYNC".

```
<MetadataType>GSSYNC</MetadataType>
```

To use GSSync to replicate based on the GSSync metadata for a table using the system data value as a primary key, set the Metadata Type to "GSSYNCSYS".

```
<MetadataType>GSSYNCSYS</MetadataType>
```

Once the metadata type has been set, you must then inform GSSync which metadata file should be checked. This file must exist, and it must have been created by GSSync using the GSCREATE option at some point in the past. Specifying the GSSync File is done with another configuration setting:

```
<GSMetadataFile>PERSON.GS</GSMetadataFile>
```

From the command line, these two settings can be handled by a single switch. To replicate data changes based on the record position value, use:

```
/MGPERSON.GS
```

To replicate data changes based on the System Data value, use:

/MSPERSON.GS

The GSSync metadata file name that you wish to read and update should be included after the switch (with no space following it). You can include an entire path, if appropriate. If your configuration file specifies the GSSync File and you wish to change it, include /MG<filename> or /MS<filename> on the command line AFTER the configuration file with the desired file name.

WARNING: NOTE THAT THE /MC OPTION WILL CREATE THE METADATA FILE WITH BOTH SETS OF DATA (RECORD POSITION AND SYSTEM DATA). HOWEVER, YOU SHOULD USE ONLY ONE OF THE SWITCHES FROM THAT POINT FORWARD – THE ONE THAT RELATES TO THE UNIQUE KEY IN THE TARGET DATABASE. IF YOU USE THE WRONG SWITCH OR ALTERNATE BETWEEN BOTH SWITCHES ON SUCCESSIVE SYNCHRONIZATION RUNS, YOU MAY EXPERIENCE MASSIVE DATA LOSS IN THE TARGET DATABASE!

Exporting Without Metadata

In some cases, it may be appropriate to export all records from a given file, regardless of any metadata. This is suitable to export a smaller database system completely every time without having to go through the process of setting up metadata for each table. This produces similar results to a straightforward export via SQLExec, the Export Data Wizard (in the PCC), or even a tool like Microsoft's DTS/SSIS, but you can define the source data using an XML file instead of requiring DDF's.

To use GSSync to replicate all records, set the Metadata Type in the configuration file to "NONE".

```
<MetadataType>NONE</MetadataType>
```

From the command line, include the following switch:

/MN

If your configuration file specifies metadata and you want to test a full export instead, include /MN on the command line AFTER the configuration file.

Chapter 5: Exporting Data from GSSync

Target Data Formats

GSSync provides the capability of exporting data into a number of target formats. These include:

- **No Output:** This option can be used when creating initial metadata structures or for testing purposes.
- **Unformatted File:** The Btrieve UNF file format is ideal when you are replicating data to an offsite location running PSQL/Zen and you want to manually load the record set at high speed into the database. It is designed as a data carrier, for which a reading application is required on the target database.
- **Btrieve File:** If you are replicating to another Btrieve file (on the same server or on a locally-accessible server also running PSQL/Zen), then the direct Btrieve replication is your best option, as it directly writes the target database at high speed with no interpretation overhead.
- **Comma-Delimited Text File:** Long a staple of data translation tools, the ubiquitous CSV file allows you to export the data as pure ASCII text and later read it into just about ANY target database. Several CSV formats are supported, including full quotes, no quotes, and quoted strings only.
- **Vector VWLOAD Text File:** When transferring data to Actian Vector, the VWLOAD process can be substantially faster than other data import options. This option creates a native VWLOAD file.
- **XML File:** Some newer environments allow you to read an XML file and directly import data, retaining some additional information about the data types and the like. This is the area that holds the most promise for future data integration needs.
- **JSON File:** Similar to the XML format, but with a much more compact representation, JavaScript Object Notation (JSON) is another new, promising technology for automating data access from text files.
- **SQL Script File:** The data can be exported as a series of SQL statements written to a text file. These statements can then be archived, compressed, and transmitted to the target database, where they can be executed directly by any SQL command processor (like Goldstar Software's SQLExec) to replicate the data.

- **ODBC Database:** When a direct ODBC target database is accessible, this option can provide an immediate import into the target database at the same time the data is being exported.
- **Forked Output:** This is not really a separate output format, but instead is a combination of a SQL Script File and ODBC Database. The data is written to both targets at the same time to facilitate archiving the changes, just in case any of the database changes are lost due to a rollback in the ODBC target.

The remainder of this chapter discusses these target data formats in more detail.

No Output

In certain cases, it may be advantageous to run a process without actually exporting any data. For example, this can save lots of processing time if you are using Btrieve-to-Btrieve replication with GSSync metadata and want to initialize the starting metadata database. In this case, you can simply copy over the Btrieve file in question to the target location, and skip the export/import steps completely.

The Export Type of NONE should be specified in the XML configuration file:

```
<ExportType>NONE</ExportType>
```

From the command line, the following switch should be specified:

```
/ON
```

This switch may not be not commonly used. However, it can be helpful in certain situations, so it has been left in as an option for the product.

Adding Dates To File Names

While exporting data, it can be sometimes helpful to be able to build up a library of exports automatically, one from each database run. This can be useful to archive a series of changes for auditing or warehousing purposes. While it is certainly possible to specify unique file names on every run, the scripting to achieve this can become tedious. Luckily, GSSync provides two solutions to add local datetime information to the export file names for you.

If you want the date to come first (allowing the target files to be sorted in datetime order), then you need to prepend the date information to the filename. This is done through the PrependDatetimeToFileName value in the XML Configuration file:

```
<PrependDatetimeToFileName>0</PrependDatetimeToFileName>
```

From the command line, the following switch should be specified:

```
/PD0
```

If you prefer that the date information be AFTER the base filename (so that you can organize your files by base filename instead), then use the AppendDatetimeToFileName value in the XML configuration file:

```
<AppendDatetimeToFileName>0</AppendDatetimeToFileName>
```

Or, from the command line, use this switch:

```
/AD0
```

For these two switches, specify a value from the following table:

Type Code	Datetime Format
0	None (Default)
1	MMDD
2	YYYYMMDD
3	YYYYMMDD_HHMM
4	YYYYMMDD_HHMMSS

The datetime information will be separated from the base filename by an underscore character (_) to make it easier to see. If the target file has an extension, the append date will be BEFORE the final "." in the filename. If no dot or extension is present in the filename, the data will be appended to the end of the file name.

Unformatted File

Exporting to a UNF file was originally designed as an interim solution to be only used during the development and testing of GSSync. However, we found that additional value could be had by leaving this feature in the system, as it provides a way to replicate data at the Btrieve level without requiring direct Btrieve access. It also provides a way to evaluate the EXACT record structure being exported, which can be useful in analyzing SQL or ODBC export problems caused by invalid bytes in the source data.

Importing the UNF data into the target database requires an additional import module that has not yet been built. If this feature is important to you, please contact Goldstar Software for information about the import module.

The Export Type of UNF and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>UNF</ExportType>  
<ExportUNFFile>PERSON.UNF</ExportUNFFile>
```

From the command line, the following switch should be specified:

```
/OU<Filename>
```

Included after the switch (with no space following it) should be the filename to which you would like to export the data in UNF format. This file name may be modified by the AppendDatetimeToFileName switch.

Btrieve File

An export to a target Btrieve file provides direct, Btrieve-to-Btrieve replication when the target file is either on the same server (and thus using the same database engine) or on another server with its own database engine. Direct replication is the simplest and fastest replication possible, as it requires only a source and a target file and no database definitions (i.e. DDFs) of any kind. Writing the target data is faster than other target data types since there is no interpretation of the bytes to be done -- the raw data blobs are merely sent into the target database as-is.

With Btrieve File replication, the original Btrieve file should be identical to the target Btrieve file (which can be created with a simple BUTIL -CLONE command). When establishing starting data, you can also just use a simple file copy, which seeds the initial data as identical to the source data file, too. One important requirement is that both the source and the target Btrieve files MUST be configured to include System Data, which also implies that the files be at least 7.x file format. Failure to do this can result in problems with the replication.

The Export Type of BTRV and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>BTRV</ExportType>
<ExportBTRVFile>PERSON.BTV</ExportBTRVFile>
```

From the command line, the following switch should be specified:

```
/OB<Filename>
```

The Btrieve filename to which you would like to export the data in Btrieve format should be included after the switch (with no space following it). This file must be created already to avoid returning a processing error, so the AppendDatetimeToFileName option will not affect this file name.

Queue File

An export to a binary Queue file is very similar to a Btrieve file target. However, instead of applying the changes directly to the target Btrieve file, GSSync instead writes the changed data to an intermediary binary file. The resulting file format can then be read by the GSSyncImport utility to apply the changes to a target Btrieve file on a different server or environment.

Why would you use this type of solution? In some cases, direct Btrieve-to-Btrieve replication is simply not available, perhaps because the database may be located on a remote LAN or other server that cannot be accessed by the primary environment. The Queue file would allow you to replicate the data to this intermediary format for transfer, and when the files arrive at the target server, they can be applied from there. Another use case could be when you want the data on the target server to lag behind

the source server for some reason – i.e. you want more fine-grained control over the replication process.

The binary Queue file is very similar to the Archive Log file, in that it simply contains a list of changes that must be applied to the target database in a specific order, and because it runs at the Btrieve layer, it does NOT require file definitions. Since the queue file leverages the System Data value as its primary key (in order to maintain the proper record status on the target), it is a bit more forgiving and more efficient when used with DX Metadata. However, like an archive log, the files MUST be applied in the exact same order in which they are generated, or inconsistent results may be seen. Because of this possibility, performing a periodic full synchronization (which can be done simply by copying the files over again) may be an important step for long-standing replication sessions.

As with Btrieve File replication, the original Btrieve file should be identical to the target Btrieve file (which can be created with a simple BUTIL -CLONE command). When you establish your starting data, you can perform a simple file copy, which seeds the initial data as identical to the source data file, too. One important requirement is that both the source and the target Btrieve files MUST be configured to include System Data, which also implies that the files be at least 7.x file format. Failure to do this can result in problems with the replication.

The Export Type of QUEUE and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>QUEUE</ExportType>
<ExportQueueFile>PERSON.BQF</ExportQueueFile>
```

From the command line, the following switch should be specified:

```
/OQ<Filename>
```

The Queue filename to which you would like to export the data in binary format should be included after the switch (with no space following it). This file name can (and probably should) be modified by the AppendDatetimeToFileName switch.

Comma-Delimited Text File

The comma-delimited export file, also known as comma separated value or CSV, is the most common format used for heterogeneous replication. Just about every major database environment provides direct support for CSV files, as do many data import/export and analysis tools (including Microsoft Excel and Access).

A comma-delimited (CSV) file is a text file (that can be opened by Notepad or any other text editor) that contains a header line with the field names for each field in the data file, followed by one line for each record in the database. Within each text line, the

individual fields are separated by commas (thus the name *comma-delimited*). String fields are often contained in quotes, which gives rise to the other common name for this format, the *comma-quote-delimited file*. An example of a CSV file is shown here:

```
"ID","First_Name","Last_Name","Date_Of_Birth"  
"100062607","Janis","Nipart","10/22/1962"
```

The "official" specification for a CSV file does not require quotation marks unless the data contains commas, so numeric fields are typically represented just as the numeric value itself. However, some tools want to see quotes around all string fields. GSSync supports several formats, in the hopes that it will match whatever import application you are using.

The Export Type of CSV, the Export File Name, and the Export CSV Format should each be specified in the XML configuration file:

```
<ExportType>CSV</ExportType>  
<ExportCSVFile>PERSON.CSV</ExportCSVFile>
```

From the command line, the following switch should be specified:

```
/OC<Filename>
```

The filename to which you would like to export the data in CSV format should be included after the switch (with no space following it). This file name may be modified by the AppendDatetimeToFileName switch.

CSV Header Row

The CSV file format typically includes a header row that indicates each field name in the exported data set. This is critical for some applications to ensure that field names line up properly, and it can be used when importing into applications like Microsoft Excel to ensure that the data is readable by humans. Including the CSV header is particularly important when you are exporting additional fields, so that you don't get any of them confused.

However, some CSV environments may not need nor want the header row. This is especially true of systems that are further importing the data using some other tool. For example, the Goldstar Software tool SQLExec supports running a single INSERT statement from a raw CSV data set, but the first line can NOT contain the field names.

While the default is to include the CSV header row, you can control this feature with this setting in the XML configuration file:

```
<ExportCSVHeaderRow>0</ExportCSVHeaderRow>
```

From the command line, the following switch should be specified:

```
/OH0
```


By default, the header row is included in all CSV files. If your configuration file specifies the CSV header row and you wish to switch it off, include /OH0 on the command line AFTER the configuration file. Conversely, use /OH1 to switch it on from the command line.

CSV Export Format

While the CSV format is considered a “standard” in the industry, several different implementations of the standard have been used in the way quotes are included (or not). If you do not specify any specific format, GSSync will include quotes around each and every field in the output stream, as well as the column names in the header row. This is *almost* always safe to do, and most import programs should be able to handle this format. If your import application doesn’t like the default format, you can change the way quotes are added with this setting in the XML configuration file:

```
<!--
    Let's define the CSV Export Format Here
    Set to 0 to quote none of the fields (Raw Comma File)
    Set to 1 to quote all fields (Comma-Quote File))
    Set to 2 to quote only String fields
    Add 10 to eliminate quotes from the header row field names
-->
<ExportCSVFormat>1</ExportCSVFormat>
```

From the command line, the following switch should be specified:

```
/CE1
```

By default, every field of every row (and the header row) are quoted, and we recommend starting here for each new project. If you change the value of this setting to 0, then none of the fields are quoted. This is often called the “raw comma” format. If you also want to eliminate quotes from the header row, use the value of 10 instead. Some import programs will want quotes only on string fields, and you can enable this functionality by using the setting value of 2 (or 12 to eliminate header row quotes).

If your configuration file specifies the CSV Export Format and you wish to change it temporarily, include /CE<#> on the command line AFTER the configuration file, where <#> is the desired option from the list above.

Vector VWLOAD Text File

The Actian Vector database provides a columnar-store database structure that is ideal for data analytics and reporting. While the CSV data format above can indeed be used by VWLOAD, the native format eliminates much of the overhead of the CSV format by using vertical bars to delineate fields and foregoing the quotation marks.

A VWLOAD (VWL) file is a text file (that can be opened by Notepad or any other text editor). Unlike a CSV, there is no header row, so you must ensure that you have the correct data fields available before starting. Each line in the file represents a record in

the database. Within each text line, the individual fields are separated by vertical bars (|). No quotation marks or other delineators are required. An example of a VWL file is shown here:

```
100062607|Janis|Nipart|1962-10-22
```

When you force the output data into the VWL format from the command line (with the /OV switch), GSSync is smart enough to make several other configuration changes for you:

- BooleanExportFormat (/BE) gets set to 1 (TRUE/FALSE).
- DateExportFormat (/DE) gets set to 2 (yyyy-mm-dd).
- TimeStampExport (/SE) gets set to 0 (yyyy-mm-dd hh:mm:ss.ff)
- TimeExportFormat (/TF) gets set to 0 (hh:mm:ss.ff)

These changes ensure that the data is most compatible with VWLOAD. If you require a different value for any of these switches, then you must set these values AFTER the /OV switch on the command line.

The Export Type of VWL and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>VWL</ExportType>  
<ExportVWLFile>PERSON.VWL</ExportVWLFile>
```

From the command line, the following switch should be specified:

```
/OV<Filename>
```

The filename to which you would like to export the data in VWL format should be included after the switch (with no space following it). This file name may be modified by the AppendDatetimeToFileName switch.

XML File

The eXtensible Markup Language, or XML, format has seen a surge in popularity in recent years. XML files are simple text files that follow a straightforward pattern or structure, using tags (strings enclosed in angle brackets) to identify data. XML files can be viewed and edited with any text editor, such as Notepad, which makes them very easy to use when data correction or other modifications may be required. XML files also provide a way to include additional metadata (such as field names) in the data structure. They are gaining in popularity as computers get faster and data storage gets cheaper, since the additional overhead in storage is a small cost to gain additional flexibility in the data itself.

Currently, GSSync supports two different XML output formats, currently called "raw" and "fields" format. These two output formats will be described in the following sections. As a younger technology brimming with flexibility, the XML formats don't seem to have many well-defined standards yet. As such, it is likely that the number of supported XML formats will be extended in future releases. If you have a need for a specific XML layout, please contact Goldstar Software.

The Export Type of XML and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>XML</ExportType>
<ExportXMLFile>PERSON.XML</ExportXMLFile>
```

From the command line, the following switch should be specified:

```
/OX<Filename>
```

The filename to which you would like to export the data in XML format should be included after the switch (with no space following it). This file name may be modified by the AppendDatetimeToFile Name switch.

Raw XML Format

The Raw XML Format is a straightforward table layout which consists of a table tag, named by the table name (or Btrieve file if a table name is not available), with a number of child "row" elements. Each field or column in the database is provided as a distinct attribute for the <row> element. Here is an example of an XML output in raw format:

```
<?xml version='1.0' ?>
<Person.MKD xmlns:sql="urn:schemas-microsoft-com:xml-sql">
  <row ID="100062607" First_Name="Janis" Last_Name="Nipart"
Date_Of_Birth="10/22/1962" />
  <row ID="100285859" First_Name="Lisa" Last_Name="Tumbleson"
Date_Of_Birth="09/25/1948" />
</Person.MKD>
```

The Export XML Format should be specified in the XML configuration file:

```
<ExportXMLFormat>0</ExportXMLFormat>
```

From the command line, the following switch should be specified:

```
/XF0
```

By default, the Raw XML Format is selected. If your configuration file specifies the Field XML Format and you wish to switch it back to Raw XML Format, include /XF0 on the command line AFTER the configuration file.

Field XML Format

The Field XML Format is slightly different in that each row is provided as a separate element tagged with the table name (or Btrieve file name, if a table name is not

available). Within each row, each field is defined as a distinct child element. Here is an example of an XML output in field format:

```
<?xml version='1.0' ?>
<Root>
  <Person.MKD>
    <ID>100062607</ID>
    <First_Name>NULL</First_Name>
    <Last_Name>Nipart</Last_Name>
    <Date_Of_Birth>10/22/1962</Date_Of_Birth>
  </Person.MKD>
  <Person.MKD>
    <ID>100285859</ID>
    <First_Name>Lisa</First_Name>
    <Last_Name>Tumbleson</Last_Name>
    <Date_Of_Birth>09/25/1948</Date_Of_Birth>
  </Person.MKD>
</Root>
```

The Export XML Format should be specified in the XML configuration file:

```
<ExportXMLFormat>1</ExportXMLFormat>
```

From the command line, the following switch should be specified:

```
/XF1
```

By default, the Raw XML Format is selected. If you wish to use the Field XML Format, include /XF1 on the command line AFTER the configuration file. Additional XML formats may be added in subsequent versions.

XML Tags

It is important to note that the naming conventions allowed for XML tags is similar to, yet slightly different from, that for SQL tables and columns. An XML tag cannot contain any special characters other than period (.), underscore (_) and hyphen (-). If any invalid characters are found, they will be replaced with underscore (_) in the output stream. Further, an XML tag cannot start with "XML", a hyphen (-) or numeric digit (0-9). If a column tag tries to start with one of these, the text "C_" will be prepended to the tag. For row tags, "R_" is prepended instead.

JSON File

The JavaScript Object Notation, or JSON format, is also getting more popular with data transfer environments. JSON files are simple text files that follow a straightforward pattern or structure, using name/value pairs to identify data. The power of JSON comes in its ability to represent nested data structures (though the design of GSSync limits this benefit, since no nesting will ever occur). JSON files can be viewed and edited with any text editor, such as Notepad, which makes them easy to use when data correction or other modifications may be required. Like XML files, JSON files also include additional metadata (such as field names) in the data structure.

Currently, GSSync supports two different JSON output formats, currently called “readable” and “compact” formats. These two output formats will be described in the following sections. If you have a need for a specific JSON layout, please contact Goldstar Software.

The Export Type of JSON and the Export File Name should both be specified in the configuration file:

```
<ExportType>JSON</ExportType>
<ExportJSONFile>PERSON.json</ExportJSONFile>
```

From the command line, the following switch should be specified:

```
/OJ<Filename>
```

The filename to which you would like to export the data in JSON format should be included after the switch (with no space following it). This file name may be modified by the AppendDatetimeToFileName switch.

Readable JSON Format

The Readable JSON Format is the default output format, and it uses whitespace (CR/LF pairs and indentation) to provide a visual representation of the data in the output text file, suitable for human review and correction. Here is an example of an JSON output in Readable format:

```
{
  "Root": {
    "Person": [
      {
        "ID": 104101361,
        "First_Name": "James"
      },
      {
        "ID": 104321686,
        "First_Name": "Kanagae"
      }
    ]
  }
}
```

The Export JSON Format should be specified in the configuration file:

```
<ExportJSONFormat>0</ExportJSONFormat>
```

From the command line, the following switch should be specified:

```
/JF0
```

By default, the Readable JSON Format is selected. If your configuration file specifies the Compact Format and you wish to switch it back to Readable Format, include /JF0 on the command line AFTER the configuration file.

Compact JSON Format

The Compact JSON Format contains the exact same data, but most whitespace is removed to provide the smallest possible output file. This saves space and processing overhead when a human will not be examining the data, but rather a computer will be processing it directly. Here is the same JSON output as above in the Compact format:

```
{"Root":{"Person":[{"ID":104101361,"First_Name":"James"}, {"ID":104321686,"First_Name":"Kanagae"}]}}
```

The Export JSON Format should be specified in the configuration file:

```
<ExportJSONFormat>1</ExportJSONFormat>
```

From the command line, the following switch should be specified:

```
/JF1
```

By default, the Readable XML Format is selected. If there is a configuration file entry, that will change the default. Including /JF1 on the command line AFTER the configuration file will override this further.

Line-Oriented JSON Format

In some cases, it may be desirable to process each record as its own JSON data, which can allow you to read one line at a time and read it with a JSON parser, instead of having to read the entire data set at once. The line-oriented JSON format has been defined to satisfy this need, where each row is a single JSON data set, separated by CR/LF pairs. Here is the same JSON output as above in the line-oriented format:

```
{"ID":104101361,"First_Name":"James"}  
{"ID":104321686,"First_Name":"Kanagae"}
```

The Export JSON Format should be specified in the configuration file:

```
<ExportJSONFormat>2</ExportJSONFormat>
```

From the command line, the following switch should be specified:

```
/JF2
```

By default, the Readable XML Format is selected. If there is a configuration file entry, that will change the default. Including /JF2 on the command line AFTER the configuration file will override this further.

JSON Names

Naming conventions allowed for JSON names are quite limited. Allowable characters in a name can be a letter, underscore (_), dollar sign (\$), or a numeric digit. If any invalid characters are found in a field name, they will be replaced with underscore (_) in the output stream. Further, a JSON name cannot start with a numeric digit (0-9). If a

column name tries to start with one of these, the text "C_" will be prepended to the name. For row names, "R_" is prepended instead.

SQL Script Export File

Another capability of GSSync is the ability to export data records and changes as native SQL statements. This one capability alone places GSSync far above most other replication and data manipulation tools available today.

Exporting data as a SQL script allows you to replicate a PSQL-based data set to ANY SQL-compatible format quite easily. You want to move data from Btrieve to SQL Server? No problem! Oracle? Easy! MySQL? Done! In essence, you can now replicate PSQL/Zen data to any SQL database that can execute simple script statements in a text file. Of course, to replicate this data via SQL properly, you **MUST** have data definitions available, through either DDF files or an XML definition of the data to be replicated. Failure to provide the file definitions will result in an error.

Even more powerful, the SQL script files are straightforward text files, meaning that you can review and modify your data prior to importing if your needs dictate. This makes troubleshooting problems even easier, since you can clearly see and adjust the SQL statements that GSSync is generating, if needed. Script files are easy to compress with compression (i.e. ZIP) utilities for permanent archiving, and the high level of compression means that they can be easily transmitted to an off-site location for import into a database, making them uniquely suitable for exporting data to off-site web servers.

It is also important to note that a SQL Script export generates the exact same SQL statements as what is pushed through the ODBC driver during an ODBC export. This makes the SQL script a great way to troubleshoot the ODBC export functionality, too.

SQL scripts are a VERY complicated feature, so you should expect to expend considerable effort in carefully building your replication environment.

When a SQL script is written, an "empty" SQL statement is read from the Configuration file (in XML), and GSSync builds the final statement using a series of replacement patterns to specify the data to be passed. See the section below titled **SQL/ODBC Statement Definitions** for more details about these statements and replacement patterns.

The Export Type of SQL and the Export File Name should both be specified in the XML configuration file:

```
<ExportType>SQL</ExportType>
<ExportSQLFile>PERSON.SQL</ExportSQLFile>
```

From the command line, the following switch should be specified:

/OS<Filename>

Included after the switch (with no space following it) should be the filename to which you would like to export the data in SQL script format. This file name may be modified by the AppendDatetimeToFileName switch.

ODBC Database Target

If you are able to define the requirements for the SQL script export, but you would like to load the data into a database that is directly accessible to the computer running GSSync, then you are a candidate for using the ODBC Database Target. This option allows the SQL statements to be sent directly to any accessible ODBC data target for execution in real time, eliminating the need for an extra processing step to run the scripts.

Using an ODBC target requires that the ODBC driver for the given database be installed and properly configured. Once this is completed, you build a connection string, which defines to the ODBC environment how the database connection should be made, and provide that connection string to GSSync so that it can connect to the database. You can use the Microsoft test utility, ODBCtest, to work with your ODBC connection strings to identify the exact format required for your database.

As with the SQL Script export, you MUST have data definitions available, through either DDF files or an XML definition of the data to be replicated. Failure to provide the file definitions will result in an error. Further, the target table MUST exist in the target database. Because you may be adding or removing fields from the target data via the SQL statements, GSSync will not attempt to create the table for you.

Once connected, each SQL statement that is built (as in the SQL Script Export described above) is sent immediately to the ODBC driver for processing by the target database. Again, this can be a complicated solution, so see the next section for information about writing your source SQL statements.

The Export Type of ODBC and *either* the ODBC Connect String or the ODBC Connect String File should be specified in the XML configuration file:

```
<ExportType>ODBC</ExportType>
<ExportODBCConnectString>DSN=DEMODATA</ExportODBCConnectString>
<ExportODBCConnectStringFile></ExportODBCConnectStringFile>
```

As you can see, GSSync allows you to specify the connection string directly in the configuration file, but you can also reference a separate text file that contains the connection string. Goldstar Software recommends that **ONLY** the connection string file be used, and not the connection string option. While specifying the connection string may be easier for one replication object, it also means that if you have hundreds of files (and thus hundreds of configuration files) that changing the connection string would

involve hundreds of individual edits. You can avoid this extra work by creating a text file with your connection string and referring to that only. (If the connection string contains a user name and password, it is also critical to protect this data with appropriate security.)

To specify the Connection String File from the command line, use this switch:

```
/OR<Filename>
```

The filename that contains the ODBC Connection String should be included after the switch (with no space following it).

If you must specify the connection string itself on the command line, use this switch:

```
/OO<connectionstring>
```

After the parameter, include the ODBC Connection String you need after the switch (with no space following it). This works well only for short strings, such as “DSN=dsnname”, but it can get complicated for longer connection strings. You may need to wrap the entire connection string in double quotes to bypass the command line parser, and you may need to otherwise escape other double quotes contained in that string. If you have issues with this, use the option to Read the connection string from a file (/OR) instead.

Forked (SQL and ODBC Database) Target

Sometimes it may be helpful to have the data written directly to an ODBC target, but a rollback or recovery in that environment can result in lost data. In that case, you may wish to use the FORK output option, instead. Essentially, this combines the ODBC output format with the SQL text file format and writes both formats simultaneously. Please read the two above sections for more specific details.

The Export Type of FORK, the SQL File Name, and the ODBC connection information should be specified in the XML configuration file:

```
<ExportType>FORK</ExportType>
<ExportSQLFile>PERSON.SQL</ExportSQLFile>
<ExportODBCConnectionString>DSN=DEMODATA</ExportODBCConnectionString>
<ExportODBCConnectionStringFile></ExportODBCConnectionStringFile>
```

From the command line, the following switch should be specified:

```
/OF<Filename>
```

Included after the switch (with no space following it) should be the filename to which you would like to export the data in SQL script format. If blank, the data from the configuration file will be used. Note that if you are ALSO specifying the ODBC connection string on the command line, the /OO or /OR option should be included FIRST on the command line, with /OF option following.

SQL/ODBC Statement Definitions

When SQL and ODBC targets are being used, GSSync must be able to construct a correct SQL statement to replicate the database changes to the target database. This is done using special configuration options that are used to build SQL statements with the help of *replacement strings*, or special strings that indicate a field or other metadata information should be provided in the SQL statement. You can picture the replacement strings as a series of "search & replace" operations on the original SQL statement before it is submitted to the target.

Three SQL statements are defined in the configuration file, one each for the possible changes detected in the source database -- DELETE, INSERT, and UPDATE. Note that some operations are applicable to only specific metadata sources. For example, Archive log metadata tracks deletes, inserts, and updates separately, so each statement type will be used and must be defined. However, PDC-based metadata is only able to properly track records that have been deleted and records which currently exist (i.e. they were either inserted or updated, or possibly both), so this export would only use the DELETE and UPDATE statements.

Due to command line length limitations, it is simply impractical to specify a command line with the SQL statements included on it. For this reason, GSSync requires that these options be specified in the XML configuration file ONLY.

Replacement Strings

Replacement strings are special sequences of characters delineated by double-curly brackets, as in the example "{{SKEY}}". When these replacement strings are found by the GSSync SQL preprocessor, they are replaced in the final SQL statement with the text to which they are referring. These fields then act as simple placeholders for real data in the SQL statement definitions in the configuration file.

There are eight metadata replacement strings defined by GSSync to refer to the database metadata for the record change being replicated. These strings are:

Token	Replacement String
{{SKEY}}	The S ystem KEY data value of the record
{{UKEY}}	The U ppdate system KEY data value of the record
{{POSN}}	The B trieve record PO sition of the record
{{TYPE}}	The change TYPE , which can be either D, I, or U
{{PROC}}	The timestamp of the GSSync PRO Cessing run (i.e. the time of the start of the GSSync process)
{{LUPD}}	The L ast UP Dated timestamp of the record (valid for Archive Log and PDC Metadata only)
{{TNAM}}	The T able NA Me of the source table (valid only when using /DT or DatabaseTable setting)
{{BNAM}}	The B trieve file NA Me of the source Btrieve file, which may be from the /BF switch, the BtrieveFile

config setting, or extracted from the DDF's.
{{DNAM}} The **Database NAME** of the source database (valid only when using /DN or DatabaseName setting)

These metadata replacement strings allow us to access the record's position or system data as a unique record identifier, which makes true replication possible. The named fields can help you to simplify massive scripting into more generic config files.

If you are using GSSync metadata, please note the warnings indicated above – you should **ONLY** use the record identifier (SKEY or POSN) that matches your export type. Having access to the timestamp fields also makes it possible to replicate data directly into a data warehouse, where records are always inserted as NEW data with the associated timestamp.

In addition to the metadata replacement strings, we can also use replacement strings to indicate the various fields within the database by name, number, or with a shortcut for all fields.

To use a replacement string by field name, simply provide the source field name inside the double curly brackets. For example, accessing the field name "ID" can be done by simply including "{{ID}}" in your SQL script. In the rare event that a field name is a duplicate of one of the pre-defined metadata replacement strings (like {{TYPE}}), the metadata replacement string will take precedence.

You can also refer to fields in the source data record by their ordinal field number. Field #1 is the first field in physical order in the database -- i.e. the field that would come back first if you issued a "SELECT * FROM Table" statement. To specify the first field, therefore, you would provide a replacement string of "{{1}}". If you wanted to include the first three fields as separate fields, your replacement sequence would look like this:

```
{{1}}, {{2}}, {{3}}
```

Note that replacement strings are very simplistic to use, because they work just like a search and replace function in the SQL script, but they can be very powerful as well. For example, you could concatenate the first two fields during the export by specifying a replacement sequence like this:

```
{{1}}+{{2}}, {{3}}
```

One more replacement string is defined as a shortcut that refers to "all fields in a comma-delimited list". To use this shortcut, use a replacement string wildcard of "{{***}}", and each field from the source file will be included in the order in which they appear in the source data table. In the case of a variant data definition, your source definition may include "filler" fields that are not part of the actual data set. This can cause all sorts of problems exporting data when the data types of the filler data fields do

not match the data itself. In this case, avoid using `{{****}}` and specify only the fields that are properly defined for this variant record type.

The GSSync SQL preprocessor can handle SQL statements, after all replacement strings have been replaced with their final text, of up to 64K in size.

DELETE Statements

A DELETE statement is the easiest case. Assuming that we know what the unique identifier is on the target data set, we can replicate a delete with a replacement string like this:

```
DELETE FROM MyTable WHERE SystemData = {{SKEY}};
```

This statement assumes that the System Data Value is also tracked in the replication database as the primary key -- a good way to ensure that replication works correctly. Although you may want to try it, you should NEVER use any fields from the record itself, because on a delete operation, the actual fields no longer exist in the source data! Instead, you should only use metadata fields (like `{{SKEY}}` or `{{POSN}}`).

The DELETE statement should be specified in the XML configuration file:

```
<ExportODBCDeleteQuery>statement</ExportODBCDeleteQuery>
```

The DELETE statement cannot be set from the command line.

INSERT Statements

An INSERT statement is a bit more problematic, because we need to provide the entire data set to the target database. Again, the use of the replacement strings, and especially the replacement string wildcard, makes this quite easy.

```
INSERT INTO MyTable VALUES ( {{SKEY}}, {{****}} );
```

This statement assumes that the first field of the target database has been defined to hold the 8-byte System Data Value, and that the remaining fields have been defined in the target database in the exact same order as the source database. The use of the `{{****}}` replacement string becomes a great time-saver, and it ensures that if the source data ever changes, that all of the new fields (after the change) will still be exported. Note again that tables with variant records may not be able to use this simplification feature.

Of course, if you are working with partial replication of data, exporting only the First_Name, Last_Name, and Date_Of_Birth fields, then you could build a more formal INSERT query with a column list like this:

```
INSERT INTO MyTable ( SKey, LName, FName, DOB)
VALUES ( {{SKEY}}, {{Last_Name}}, {{First_Name}}, {{Date_Of_Birth}} );
```

The INSERT statement should be specified in the XML configuration file:

```
<ExportODBCInsertQuery>statement</ExportODBCInsertQuery>
```

Like the other SQL statements, the INSERT statement cannot be set from the command line.

UPDATE Statements

The most complicated statement to write is an UPDATE statement. This is especially complicated because when using PDC-based metadata, an UPDATE and an INSERT are both treated identically -- there is simply no way to tell which is which.

To make the UPDATE query easier, GSSync has the capability of running multiple statements, separated by a semicolon. This allows us to simplify the UPDATE statement into two separate statements, namely a DELETE followed by an INSERT. Therefore, a reasonable UPDATE statement could be the combination of the two above statements:

```
DELETE FROM MyTable WHERE SystemData = {{SKEY}};  
INSERT INTO MyTable VALUES ( {{SKEY}}, {{****}} );
```

Alternatively, if you are using Archive Log or GSSync metadata, or some other method to know when you have a true UPDATE change, you can use a true SQL UPDATE statement like this:

```
UPDATE MyTable SET  
LName = {{Last_Name}}, FName = {{First_Name}}, DOB = {{Date_Of_Birth}}  
WHERE SystemData = {{SKEY}};
```

Of course, a table with 50 or so fields will have a MUCH longer statement. It should now be clear why these statements should not be provided on the command line.

It is also possible to use a plain INSERT statement instead of an UPDATE statement, when a data warehouse solution is being utilized. In this case, there should be a combination key in the target database on the {{SKEY}} field and the {{LUPD}} or {{PROC}} timestamp field, so that a simple INSERT statement would then suffice.

The UPDATE statement should be specified in the XML configuration file:

```
<ExportODBCUpdateQuery>statement</ExportODBCUpdateQuery>
```

The UPDATE statement cannot be set from the command line.

Using Multiple SQL Statements

If you need to specify multiple SQL statements in any of the SQL/ODBC Statement Definitions, as in the example of the UPDATE statement above which is comprised of separate DELETE and INSERT statements, you should separate each statement with a semicolon.

If multiple statements are sensed, they are normally sent to the database as a single coherent whole, where both statements are sent together on the same command line.

Most SQL environments will properly execute multiple statements sent to the database at the same time if they are properly separated by semicolons. This is why a series of hacking attacks known as SQL Injection Attacks can be so successful.

Some SQL environments, however, may not allow this multi-statement execution properly and will return execution errors. To address this, another switch has been added to the Configuration file that splits statements on the semicolon separator:

```
<ExportODBCStripSemicolons>1</ExportODBCStripSemicolons>
```

Enabling this switch (which, like the SQL statements themselves, is not available from the command line) will force GSSync to break up each multiple statement at the semicolon, handle the replacement, and then submit the statement to the SQL engine. This will allow the SQL engine to handle only one statement at a time, at the extra cost of doubling the number of ODBC calls and with an expected decrease in performance.

Deleted Data Output

For some types of replication targets, notably UNF, XML, and CSV files, we run into one more major issue. These formats are suitable for specifying the data that we are replicating, but they have no method of specifying the data that has been deleted, even though most metadata options allow us to detect the delete operations that occur. Because there is no way to represent the deletes, we can never have a true replication when replicating to these formats.

As a way of capturing this information, GSSync will create a Deleted Data File when the export type matches one of these types, and it will write the system data or record position of each deleted record to the file.

The format of the Deleted Data File will be the same as the export target format. So, if you are exporting data in CSV format, you will get a Deleted Data File in CSV format. If you are exporting in UNF format, then you will get a UNF file that includes ONLY the additional output fields (that are specified by the provided flags) for the deleted record. At this time, there is no automated tool for processing the Deleted Data Files to effect a "true" replication, but the data is captured for your manual review, and it could be read by an ETL-style application for automated processing as-is.

The pathname to the Deleted Data File should be specified in the config file in XML:

```
<ExportDeletedDataFile>ODBC</ExportDeletedDataFile>
```

From the command line, the following switch should be specified:

```
/OD<Filename>
```

Included after the switch (with no space following it) should be the filename to which you would like to export the deleted data keys.

If no Deleted Data File is specified and the export format is UNF, XML, or CSV, then a Deleted Data File name will be constructed automatically by GSSync by prepending the letters "Del_" in front of the defined export file name, and this file name will be used to contain the deleted records.

Ignoring Deleted Data

In certain situations, namely when using GSSync to assist in building a data warehouse where only changes to the system are tracked, it may be important to NOT propagate any DELETE operations to the target database. You can use the Ignore Deletes Flag to tell GSSync to skip all processing of deleted records. Note that the metadata scan (if using GSSync metadata) will still work as expected – this switch ONLY affects the exported data of the given replication cycle.

The Ignore Deletes Flag can be used with any export type. It is specified in the configuration file in the following XML element:

```
<IgnoreDeletes>1</IgnoreDeletes>
```

The Ignore Deletes Flag is specified on the command line with the following option:

```
/ID1
```

By default, delete operations are replicated normally, and you must manually enable the option to skip them. If your configuration file specifies the Ignore Deletes Flag and you wish to disable it, include /ID0 on the command line AFTER the configuration file.

Additional Output Fields

Several output fields are optional in the export. These fields can provide you with additional information about the exported data, which can be useful as you handle your own synchronization of the data on the target system.

Delta Flag

The Delta Flag is a separate output field that provides information on the type of record change that was detected by the change detection algorithm. If the change was detected as new record, then an "I" (for INSERT) is exported for this field. If the change was detected as an updated record, then a "U" (for UPDATE) is exported. If the change was detected as a deleted record, then a "D" (for DELETE) is exported. Note that some metadata options (such as PDC metadata) cannot distinguish between a new record and an update, so all existing records are considered updates.

The Delta Flag can be added to the UNF, XML, JSON and CSV export formats only and is exported with a column header of "DeltaFlag". This flag has no impact on the Btrieve export option. It can be accessed from a SQL or ODBC export via the {{TYPE}} replacement string.

The Delta Flag is specified in the configuration file in the following XML element:

```
<ExportDeltaFlagFormat>1</ExportDeltaFlagFormat>
```

It can be specified on the command line with the following option:

```
/ED1
```

By default, the Delta Flag is NOT exported. You must manually enable the option to export the flag. If your configuration file specifies the Delta Flag and you wish to disable it, include /ED0 on the command line AFTER the configuration file.

If you need to rename the field name from the default of “DeltaFlag”, you can do this with the following configuration file option:

```
<ExportDeltaFlagFormatName>DeltaFlag</ExportDeltaFlagFormatName>
```

This column name cannot be changed from the command line.

System Data Value

The System Data Value is a separate output field that provides information about the Btrieve record's system data. As described in detail in Chapter 2, system data can be used to identify every unique record in an environment, and exporting this data value can provide a unique indicator for your target database as well.

This configuration setting provides the ability to either NOT export the System Data Value (the default 0), or to include that column as an 8-byte *signed* integer value (1). [Note: This value is formatted as a signed value because it may be required as a unique identifier in the target database, and not all databases support the UNSIGNED(8) data type, such as SQL Server.] If your environment is using v2 System Data (i.e. it uses v13 data file formats and includes UpdateSysKey values), then this value is a true septasecond timestamp, and you can specify the export format as well. See the section below for **Last Change Timestamp** for possible display formats. ***Note that attempting to export older System Data values as formatted timestamp may result in errors or invalid values, as the older System Data did NOT use a true timestamp.*** (Also, note that, unlike the other timestamp fields, the system data value field header does not change with the format.)

The System Data Value Flag can be added to the UNF, XML, JSON and CSV export formats and is exported with a column header of “SystemData”. The data will be maintained automatically for Btrieve or Btrieve Queue File export options. It can be accessed from a SQL or ODBC export via the {{SKEY}} replacement string.

The System Data Value is specified in the config file in the following XML element:

```
<ExportSystemDataField>1</ExportSystemDataField>
```

It can be specified on the command line with the following option:

```
/ES1
```


By default, the System Data Value is NOT exported. You must manually enable the option to export the system data value. If your configuration file specifies the System Data Value and you wish to disable it, include /ESO on the command line AFTER the configuration file.

If you need to rename the field name from the default of “SystemData”, you can do this with the following configuration file option:

```
<ExportSystemDataFieldName>SystemData</ExportSystemDataFieldName>
```

This column name cannot be changed from the command line.

Update System Data Value

The Update System Data Value is a new component from the V2 System Data definition, first available with Actian Zen v15. This metadata element contains the timestamp of the last update of the record and is maintained continually by the engine as a true septasecond timestamp value.

This configuration setting provides the ability to either NOT export the Update System Data Value, or to export it in one of several formats. See the section below for **Last Change Timestamp** for possible display formats. (Note that, unlike the other timestamp fields, the update system data value field header does not change with the format.)

The Update System Data Value Flag can be added to the UNF, XML, JSON and CSV export formats and is exported with a column header of “UpdateSystemData”. The data will be maintained automatically for Btrieve or Btrieve Queue File export options. It can be accessed from a SQL or ODBC export via the {{UKEY}} replacement string.

The Update System Data value is specified in the config file in the following XML element:

```
<ExportUpdateSystemDataField>1</ExportUpdateSystemDataField>
```

It can be specified on the command line with the following option:

```
/EU1
```

By default, the Update System Data value is NOT exported. You must manually enable the option to export the Update System Data value. If your configuration file specifies the Update System Data Value and you wish to disable it, include /EU0 on the command line AFTER the configuration file.

If you need to rename the field name from the default of “UpdateSystemData”, you can do this with the following configuration file option:

```
<ExportUpdateSystemDataFieldName>UpdateSystemData</ExportUpdateSystemDataFieldN  
ame>
```

This column name cannot be changed from the command line.

Btrieve Position Value

The Btrieve Position Value is a separate output field that provides information about the Btrieve record's position within the database file. As described in detail in Chapter 2, positional data is NOT guaranteed to be unique at all times, and if a database file is rebuilt or otherwise reorganized, the position of all records can change, rendering this position data useless. Based on this, we do not recommend using this option unless you cannot otherwise use the System Data Value.

The Btrieve Position Value Flag can be added to the UNF, XML, JSON and CSV export formats only and is exported with a column name of “BtrievePosition”. This flag has no impact on the Btrieve export option. It can be accessed from a SQL or ODBC export via the {{POSN}} replacement string.

Note that this field is always exported as an UNSIGNED value, which is a 4-byte value for older files (9.5 or older), or an 8-byte value for v13 format data files over 256GB or 4B rows. If you are storing this data in the target database, be sure to allow for the maximum possible value and use a larger data type, if needed.

The Position Value is specified in the config file in the following XML element:

```
<ExportPositionDataField>1</ExportPositionDataField>
```

It can be specified on the command line with the following option:

```
/EP1
```

By default, the Position Value is NOT exported. You must manually enable the option to export the position data. If your configuration file specifies the Position Value and you wish to disable it, include /EP0 on the command line AFTER the configuration file.

If you need to rename the field name from the default of “BtrievePosition”, you can do this with the following configuration file option:

```
<ExportPositionDataFieldName>BtrievePosition</ExportPositionDataFieldName>
```

This column name cannot be changed from the command line.

Last Change Timestamp

The Last Change Timestamp is a separate output field that provides information about the time and date of the last time that this record was changed. This information is only provided if the selected metadata provides it, which occurs for DataExchange/PDC and Archive Log metadata only. If you use this option with any other type of metadata, it defaults to the Export timestamp.

This configuration setting provides the ability to either NOT export the Last Change Timestamp, or to export it in one of several formats. At this time, the following values are used to indicate the export format:

Value	Heading	Format
0	None	Timestamp is not exported.
1	LastUpdatedTimestampSepta	Pervasive Septasecond Format
2	LastUpdatedTimestampUnix	Unix Format (Seconds since 1/1/1970)
3	LastUpdatedTimestampTS	UTC Time in ODBC Format ({ ts ... })
4	LastUpdatedTimestampTS	Local Time in ODBC Format ({ ts ... })
5	LastUpdatedTimestampTS	UTC Time in US String Format ('mm/dd/yyyy hh:mm:ss:ff')
6	LastUpdatedTimestampTS	Local Time in US String Format ('mm/dd/yyyy hh:mm:ss:ff')
7	LastUpdatedTimestampTS	UTC Time in European String Format ('dd.mm.yyyy hh:mm:ss:ff')
8	LastUpdatedTimestampTS	Local Time in European String Format ('dd.mm.yyyy hh:mm:ss:ff')
9	LastUpdatedTimestampTS	UTC Time in Oracle TO_TIMESTAMP Format
10	LastUpdatedTimestampTS	Local Time in Oracle TO_TIMESTAMP Format
11	LastUpdatedTimestampTS	UTC Time in Vector TIMESTAMP Format
12	LastUpdatedTimestampTS	Local Time in Vector TIMESTAMP Format
13	LastUpdatedTimestampNum	UTC Time in Big Numeric Format (yyyymmddhhmmssff)
14	LastUpdatedTimestampNum	Local Time in Big Numeric Format (yyyymmddhhmmssff)
15	LastUpdatedTimestampTS	UTC Time in YearFirst String Format (yyyy-mm-dd hh:mm:ss:ff)
16	LastUpdatedTimestampTS	Local Time in YearFirst String Format (yyyy-mm-dd hh:mm:ss:ff)
17	LastUpdatedTimestampTS	UTC Time in Quoted YearFirst String Format ('yyyy-mm-dd hh:mm:ss:ff')
18	LastUpdatedTimestampTS	Local Time in Quoted YearFirst String Format ('yyyy-mm-dd hh:mm:ss:ff')

If a value of 1 is selected, the timestamp will look like 633755029020000000, and a header will be LastUpdatedTimestampSepta. If a value of 2 is selected, the timestamp will look like 1239906153 and a header will be LastUpdatedTimestampUnix. If a value of 3 is selected, the timestamp will appear as {ts '2009-04-16 18:23:17:00'} and have a header of LastUpdatedTimestampTS. If a value of 4 is selected, the timestamp will appear as {ts '2009-04-16 13:23:17:00'} and have a header of LastUpdatedTimestampTS.

You can add the Last Change Timestamp field to the UNF, XML, JSON and CSV export formats, but not to the Btrieve Export format, and the field is exported with a column header of “LastUpdatedTimeStamp”. It can be added to a SQL or ODBC export via the replacement string {{LUPD}} (signifying "Last Updated").

The Last Change Timestamp is specified in the config file in the following XML element:

```
<ExportLastChangeTimeStamp>1</ExportLastChangeTimeStamp>
```

It can be specified on the command line with the following option:

```
/TL1
```

By default, the Last Change Timestamp is NOT exported. You must manually enable the option to export the timestamp value. If your configuration file specifies the Last Change Timestamp and you wish to disable it, include /TLO on the command line AFTER the configuration file.

If you need to rename the field name from the default of “LastUpdatedTimeStamp”, you can do this with the following configuration file option:

```
<ExportLastChangeTimeStampName>LastUpdatedTimeStamp</ExportLastChangeTimeStampName>
```

This column name cannot be changed from the command line.

Export Timestamp

The Export Timestamp is a separate output field that provides information about the time and date of the actual export itself. This information is available on every export and is the time of the START of the export, not the actual time when the record was exported. This allows you to identify the set of updated records in a data warehouse as a consistent whole.

This configuration setting provides the ability to either NOT export the Export Timestamp, or to export it in one of several formats. At this time, the following values are used to indicate the export format:

Value	Heading	Format
0	None	Timestamp is not exported.
1	ExportTimestampSepta	Pervasive Septasecond Format
2	ExportTimestampUnix	Unix Format (Seconds since 1/1/1970)
3	ExportTimestampTS	UTC Time in ODBC Format ({ ts ... })
4	ExportTimestampTS	LocalTime in ODBC Format ({ ts ... })
5	ExportTimestampTS	UTC Time in US String Format ('mm/dd/yyyy hh:mm:ss.ff')
6	ExportTimestampTS	Local Time in US String Format ('mm/dd/yyyy hh:mm:ss.ff')
7	ExportTimestampTS	UTC Time in European String Format

		('dd.mm.yyyy hh:mm:ss.ff')
8	ExportTimestampTS	Local Time in European String Format ('dd.mm.yyyy hh:mm:ss.ff')
9	ExportTimestampTS	UTC Time in Oracle TO_TIMESTAMP Format
10	ExportTimestampTS	Local Time in Oracle TO_TIMESTAMP Format
11	ExportTimestampTS	UTC Time in Vector TIMESTAMP Format
12	ExportTimestampTS	Local Time in Vector TIMESTAMP Format
13	ExportTimestampNum	UTC Time in Big Numeric Format (yyyymmddhhmmssff)
14	ExportTimestampNum	Local Time in Big Numeric Format (yyyymmddhhmmssff)
15	ExportTimestampTS	UTC Time in YearFirst String Format (yyyy-mm-dd hh:mm:ss.ff)
16	ExportTimestampTS	Local Time in YearFirst String Format (yyyy-mm-dd hh:mm:ss.ff)
17	ExportTimestampTS	UTC Time in Quoted YearFirst String Format ('yyyy-mm-dd hh:mm:ss.ff')
18	ExportTimestampTS	Local Time in Quoted YearFirst String Format ('yyyy-mm-dd hh:mm:ss.ff')

If a value of 1 is selected, the timestamp will look like 633755029020000000, and a header will be ExportTimestampSepta. If a value of 2 is selected, the timestamp will look like 1239906153 and a header will be ExportTimestampUnix. If a value of 3 is selected, the timestamp will appear as {ts '2009-04-16 18:23:17:00'} and have a header of ExportTimestampTS. If a value of 4 is selected, the timestamp will appear as {ts '2009-04-16 13:23:17:00'} and have a header of ExportTimestampTS.

You can add the Export Timestamp field to the UNF, XML, JSON and CSV export formats, but not to the Btrieve Export format, and the field is exported with a column header of "ExportTimestamp". It can be added to a SQL or ODBC export via the replacement string {{PROC}} (signifying "Processed Timestamp").

The Export Timestamp is specified in the config file in the following XML element:

```
<ExportExportTimeStamp>1</ExportExportTimeStamp>
```

It can be specified on the command line with the following option:

```
/TE1
```

By default, the Export Timestamp is NOT exported. You must manually enable the option to export the timestamp value. If your configuration file specifies the Export Timestamp and you wish to disable it, include /TE0 on the command line AFTER the configuration file.

If you need to rename the field name from the default of “ExportTimestamp”, you can do this with the following configuration file option:

```
<ExportExportTimestampName>ExportTimestamp</ExportExportTimestampName>
```

This column name cannot be changed from the command line.

Ignoring Fields

In some cases, it may be advantageous to ignore one or more fields on the data export side. This can be helpful when fields are not properly defined, custom data types are implemented by the developer, or when various “filler” fields are used, but the data is not easily exported using a standard data type definition (like CHAR). Ignoring these fields can not only reduce the export data size, but it can improve performance as well, or it can allow you to create partial data extracts for testing purposes.

Ignore Field List

The Ignore Field List is a comma-delimited list of field names which will be ignored by the export function. When the table definition is being loaded, which can be from a data dictionary or XML DDL definition, any field names that appear in the Ignore Field List will be skipped in the output stream.

The Ignore Field List has no impact on BTRV, QUEUE or UNF export models, which always work on complete Btrieve-level records. It is, however, supported with CSV, FORK, JSON, ODBC, SQL, VML and XML export formats.

The Ignore Field List is specified in the configuration file in the following XML element:

```
<IgnoreFieldList></IgnoreFieldList>
```

Fields in this list do NOT have to be valid for the current table, which allows you to create a baseline configuration for multiple tables with fields like Filler, Filler2, Unused, and so on. Spaces around commas should not be included but will be removed if found.

Due to the potential length of the element and required parsing of the string value, this function cannot be specified on the command line at this time.

Data Output Formatting Options

Date and time values are two data types that are widely supported by various database environments, but it seems that everybody likes to get their dates or times in a specific format, depending on their target database. As a result, when GSSync writes date and time values, it allows you to specify the exact format to be used in the output.

Boolean Export Format

GSSync currently supports several boolean format options. These options, along with the format description and an example of each output is listed in the table below.

Value	Format	Example Output
-------	--------	----------------

0	1 or 0	1
1	TRUE or FALSE	TRUE
2	YES or NO	YES

If you need support for an additional export format for your specific environment, please contact Goldstar Software for an enhancement request.

The Boolean Export Format is specified in the config file in the following XML element:

```
<BooleanExportFormat>1</BooleanExportFormat>
```

It can be specified on the command line with the following option:

```
/BE1
```

By default, the Boolean Export Format is 0, using 1 and 0, but some databases (like Vector) may require TRUE and FALSE. (If you specify the /OV option on the command line, this value is set to 1 by default.) If your configuration file specifies a Boolean Export Format and you wish to change it, include /BE# on the command line AFTER the configuration file and provide the desired value.

Date Export Format

GSSync currently supports several date format options. These options, along with the format description and an example of each output is listed in the table below.

Value	Format	Example Output
0	US: mm/dd/yyyy	04/16/2009
1	European: dd.mm.yyyy	16.04.2009
2	String: yyyy-mm-dd	2009-04-16
3	Integer: yyyymmdd	20090416
10	ODBC: {d 'yyyy-mm-dd'}	{d '2009-04-16'}
11	Oracle: TO_DATE()	TO_DATE('2009-04-16','YYYY-MM-DD')
12	Vector: DATE	DATE '2009-04-16'

Values 10 and above can be used with any type of export, but they only have real value when used with SQL or ODBC exports. If you need support for an additional format for your specific environment, please contact Goldstar Software for an enhancement request.

The Date Export Format is specified in the config file in the following XML element:

```
<DateExportFormat>1</DateExportFormat>
```

It can be specified on the command line with the following option:

```
/DE1
```

By default, the Date Export Format is 0, or the standard US format. (If you specify the /OV option on the command line, this value is set to 2 by default.) If your configuration

file specifies a Date Export Format and you wish to change it, include /DE# on the command line AFTER the configuration file and provide the desired value.

Time Export Format

GSSync currently supports several time format options. These options, along with the format description and an example of each output, are listed in the table below.

Value	Format	Example Output
0	Default: hh:mm:ss:hh	14:40:32.00
1	Integer: hhmmss	144032
2	Quoted Time String	'14:40:32.00'
10	ODBC: {t 'hh:mm:ss.ff'}	{t '14:40:32.00'}
11	Oracle: TO_DATE()	TO_DATE('14:40:32','HH24:MI:SS')
12	Vector: TIME	TIME '14:40:32.00'

Values 10 and above can be used with any type of export, but they only have real value when used with SQL or ODBC exports. If you need support for an additional format for your specific environment, please contact Goldstar Software for an enhancement request.

The Time Export Format is specified in the config file in the following XML element:

```
<TimeExportFormat>1</TimeExportFormat>
```

It can be specified on the command line with the following option:

```
/TF1
```

By default, the Time Export Format is 0, or the standard format. (If you specify the /OV option on the command line, this value is set to 0 by default.) If your configuration file specifies a Time Export Format and you wish to change it, include /TF# on the command line AFTER the configuration file and provide the desired value.

Timestamp Export Format

GSSync currently supports several timestamp format options which is used to output data from TIMESTAMP and DATETIME fields. These options, along with the format description and an example of each output, are listed in the table below.

Value	Format	Example Output
0	Default: yyyy-mm-dd hh:mm:ss.ff	2017-12-05 14:40:32.00
1	Big Integer: yyyyymmddhhmmss	20171205144032
2	Quoted Timestamp String: 'yyyy-mm-dd hh:mm:ss.ff'	'2017-12-05 14:40:32.00'
10	ODBC: {ts 'yyyy-mm-dd hh:mm:ss.ff'}	{ts '2017-12-05 14:40:32.00'}
11	Oracle: TO_TIMESTAMP()	TO_TIMESTAMP('2017-12-05 14:40:32','YYYY-MM-DD HH24:MI:SS')

Values 10 and above can be used with any type of export, but they only have real value when used with SQL or ODBC exports. If you need support for an additional format for your specific environment, please contact Goldstar Software for an enhancement request.

The Timestamp Export Format is specified in the config file in the following XML element:

```
<TimestampExportFormat>1</TimestampExportFormat>
```

It can be specified on the command line with the following option:

```
/SE1
```

By default, the Timestamp Export Format is 0, or the standard format. (If you specify the /OV option on the command line, this value is set to 0 by default.) If your configuration file specifies a Timestamp Export Format and you wish to change it, include /SE# on the command line AFTER the configuration file and provide the desired value.

Export Parallel Output

The Action Vector VWLOAD utility has an option (--cluster) that allows it to import multiple files in parallel with a single command, providing substantial performance gains over a single-threaded import. The trick to making this efficient is to avoid having to manually split the input file immediately before processing, which will only lengthen the total time for the import. However, if the data can be exported in parallel with little extra cost, exploiting this parallelism makes a lot of sense. Other environments may also benefit through the parallel importing of data, so we have enabled a parallel output format feature on several export formats, including UNF, CSV, VWL, XML, JSON, SQL, and FORK (SQL only). This option has no meaning when used with BTRV, ODBC, or QUEUE export formats.

Enabling the Export Parallel Output setting tells GSSync to create “n” separate files (by appending an underscore and a two-digit number to the end of the filename) and export data to each in a round-robin fashion. The value “n” can be anywhere from 0 to 100, with 0 or 1 indicating that you want just ONE file out, and any number of 2 or above indicating that you want multiple files. This option makes each of the files approximately equal in size, with the exception of Deleted Record headers, which are always sent to the first export segment, if no Deleted Records File is indicated.

Use this switch in the XML configuration file to enable this feature:

```
<ExportParallelOutput>#</ExportParallelOutput>
```

From the command line, the following switch should be specified:

`/PO#`

In both cases, the “#” provided should be a value from 0 to 100. As with most settings, you can override the configuration file setting as well by including `/PO#` on the command line AFTER the configuration file.

Data Output Filtering Options

GSSync supports additional processing of the replication data set that allows you to change the data as it is being exported, in something known as an *output filter*. Output filters are processed on the data as it is being exported to change the data to a standard, or more meaningful, representation. An example of this is handling invalid date fields, which may be perfectly acceptable in Btrieve, but which can cause some environments (notably SQL/ODBC) some real issues.

There are two types of filters. The first type is called field filters, as they apply to the data within a particular field only, specified by their data type, such as dates and character fields. The second type is called a record filter, as it is used to determine if a particular record should be included in the output stream or not.

Date Field Filtering

Btrieve date fields are a classic trouble point for many environments. In the Btrieve world, a "date" field is simply defined as a 4-byte value, where one byte is used each for the day and month, and two bytes are used for the year. The data type is defined so that it collates as an unsigned integer, and the database never attempts to interpret these values as anything meaningful. Based on this, if an application erroneously stores four spaces (ASCII value 32) into a date field, the resulting date will be 32/32/8224. Exporting this date to an ODBC data target, which must interpret the date value as a "real" date, will result in an error.

Other bad dates can also cause problems with the ODBC drivers and other SQL engines, and this setting allows you to fix up those values on the fly as the data comes across. Month values outside the range of 1-12 are considered bad. Day values outside the range of 1-31 are also bad, with additional handling for shorter months. Leap years are calculated as well, so February 29 is allowed only on the correct years. Finally, years above 9999 may cause havoc in some SQL implementations, so these are also flagged as bad.

To fix up your dates, GSSync provides several methods of filtering the date fields.

Value	Format
0	No Filtering: Export Invalid Dates As-Is
1	Output Invalid Dates as NULL Values

- 2 Output Invalid Dates as 01/01/1901
- 3 Output Invalid Dates as 01/01/1980
- +10 Treat 00/00/0000 Dates as Invalid Dates

Note that the last option is a modifier that allows you to also treat the so-called "zero dates", or dates in which the day, month, and year are all zero, as bad dates as well. Commonly, Btrieve applications use a zero date as a flag for "no date", and these values may permeate a data set, but other database environments may not understand this notation and return errors. To fix the zero dates as well, add 10 to the fix option that you wish to use. For example, if you wanted to output all bad dates as NULL values and include zero dates in that list, your setting would be 1+10 or 11.

The Date Field Filter is specified in the config file in the following XML element:

```
<DateFieldFilterOptions>1</DateFieldFilterOptions>
```

It can be specified on the command line with the following option:

```
/DF1
```

By default, the Date Field Filter is 0, which provides no filtering. If you require filtering, you must provide a value for the filter type. If your configuration file specifies the Date Field Filter and you wish to disable it, include /DF0 on the command line AFTER the configuration file.

Character Field Filtering

Like date fields, Btrieve-based character fields can also be used to store a wide variety of data, including invalid data that cannot be displayed as a regular character field, like ASCII NULL values or control characters.

To address this potential problem, GSSync provides several methods of filtering the character or string fields.

Value	Format
0	No Filtering: Export Character Fields As-Is
+1	Change CR/LF Values to Spaces
+2	Change Null Bytes (0x00) to Spaces
+4	Change Control Characters to Spaces
+8	Clear the High Bit on Every Character
+16	Convert all text to UPPERCASE
+32	Eliminate Trailing Blanks from Character Fields
+64	Convert Vertical Bars to Spaces (needed for VWL Exports)
+128	Convert Double Quote (") to Space
+256	Convert Single Quote (') to Space
+512	Convert Backslash (\) to Space

With strings, the data can easily encompass one or more of these anomalies, so this filter is specified a bit differently than the date filter above. In this case, you can specify one or more of the character filters by *adding together* the individual values and providing that value for the switch. For example, if you wanted to convert CR/LF values and ASCII NULL bytes to spaces and truncate any trailing spaces from each field, your setting would be 1+2+32 or 35. Again, this design was chosen for its expandability, so if you have other character field filtering requirements, please contact Goldstar Software.

The Character Field Filter is specified in the config file in the following XML element:

```
<CharacterFilterOptions>1</CharacterFilterOptions>
```

It can be specified on the command line with the following option:

```
/SF1
```

By default, the Character Field Filter is 0, which provides no filtering. If you require filtering, you must provide a value for the filter type. If your configuration file specifies the Character Field Filter and you wish to disable it, include /SF0 on the command line AFTER the configuration file.

Numeric Field Filtering

Some applications, such as those written in COBOL, may use one of the NUMERIC data types (NUMERIC, NUMERICSA, NUMERICSTS, etc.). These data types are supposed to only contain numeric digits (with a separate or embedded sign flag), but they are stored such that they look just like string fields, and some may try to convert strings to NUMERIC values with this tool. Unfortunately, some applications elect to store invalid bytes in these field, and exporting these fields may yield unintelligible or incorrect data instead of a proper numeric value.

GSSync can detect and correct these bad digits in your NUMERIC data types.

Value	Format
0	No Filtering: Export NUMERIC Fields Including any Bad Bytes
1	Change any Bad Digits to "0"
2	If there are any bad digits, change the field value to 0.
3	If there are any bad digits, change the field value to NULL.
4	If there are any bad digits, change the field value to 9's.
5	If there are any bad digits, change the field value to -9's

The first option attempts to correct each invalid digit. In many cases, applications will erroneously store an ASCII space (0x20) in the numeric field instead of a leading 0. In other cases, an ASCII null (0x00) may be used. Using a Numeric Filter of 1 will fix this, so a 5-digit value of " 34" will be corrected to "00034". (Such leading zeroes will be removed by default, but see the next section if you need to retain them.)

The other 4 options are a bit more draconian, as they assume that ANY bad bytes make the entire field invalid and therefore will replace the invalid value with a known “flag” value, such as 0 or NULL. With the last two options, the length of the field is dependent on the length of the NUMERIC field itself. Thus, a 5-digit numeric field of “00 00” will be converted to either “99999” or “-99999”.

If you can justify the need for a different filtering option or need different logic for this filter, please contact Goldstar Software.

The Numeric Field Filter is specified in the config file in the following XML element:

```
<NumericFilterOptions>1</NumericFilterOptions>
```

It can be specified on the command line with the following option:

```
/NF1
```

By default, the Numeric Field Filter is 0, which provides no filtering. If you require filtering, you must provide a value for the filter type. If your configuration file specifies the Numeric Field Filter and you wish to disable it, include /NF0 on the command line AFTER the configuration file.

Retain Leading Zeroes in NUMERIC Fields

When applications use the NUMERIC or DECIMAL data types, the data format allows for the existence of leading zeroes in the data. When exporting these data types to the JSON export format, these leading zeroes will break the JSON parser, so they are automatically removed. However, with other export formats, these leading zeroes may be important to keep. One example of this is when using a DECIMAL field to store a human-readable date that starts with the month. If you need to parse out the month as the first two bytes in a SQL statement, the leading zero is *very* important. A similar issue can be seen when using a NUMERIC value in one table to link to a string in another.

To facilitate these use cases, GSSync has an option to retain the leading zeroes in the NUMERIC and DECIMAL data types. This cannot be set per field, but rather is an on/off type switch for all such fields that defaults to OFF. You can enable the retention of leading zeroes in these fields with the following XML element:

```
<RetainLeadingZeroes>1</RetainLeadingZeroes>
```

It can be specified on the command line with the following option:

```
/LZ1
```

By default, the Retain Leading Zeroes flag is 0, which removes any leading zeroes or spaces. If you require these characters to be maintained, enable this setting. If your configuration file specifies the Retain Leading Zeroes and you wish to disable it, include /LZ0 on the command line AFTER the configuration file.

Record Filtering

Record filtering includes the concept of including or skipping records in a replication cycle based on the content of one or more fields, specified by a filter expression. One use of record filters is to synchronize only newer data records (i.e. skip all records older than 2 years). Another use of record filters is to export data from a central database with only information specific to a single location, allowing you to replicate location-specific data from the central database to a database at each location -- without the overhead of replicating every change.

The *record filter expression* is a numeric expression that can be evaluated into a true or false value. If the expression evaluates to a non-zero value (i.e. TRUE) for a given record, the data record is exported. If the expression evaluates to 0 (i.e. FALSE), the data record is skipped. Please note that the current version of GSSync supports ONLY numeric expressions -- there is no support for string comparisons or expressions at this time. This may be considered for a future enhancement.

Record filter expressions can include any numerical computation (+, -, *, /), as well as parentheses to specify order of operations, but they can also include comparison operations (<, <=, >, >=, ==, and !=), logical functions like AND (&&) and OR (||), mathematical functions, randomization functions, and even a tertiary if(,,) function. Any numerical field value from the record itself can also be specified by including the SQL field name in the expression. (Data definitions are obviously required for this feature to work.)

Note that in an XML file, the characters "<" and ">" must be avoided. To facilitate the use of the XML code, GSSync has defined the following constants:

Comparison Operator	Specified in the XML File
>	>
>=	≥
<	<
<=	≤
<>	Use "!=" Instead

This can be seen in this example that exports only records where the field "ID" is greater than or equal to the value 200000000:

```
ID &ge; 200000000
```

Note that the command line filter option can be used with the XML formats or the original operator, but the filter MUST be enclosed in double quotes, as in this example:

```
GSSYNC /CFgssync.cfg /RF"ID >= 200000000"
```

In addition to specifying field names and mathematical operations, the following functions are defined and supported within GSSync's record filter expressions:

Function	Description
Abs(x)	The absolute value of x
Acos(x)	The arccosine of x
Asin(x)	The arcsine of x
Atan(x)	The arctangent of x
Ceil(x)	The next highest whole integer of x
Cos(x)	The cosine of x
Cosh(x)	The hyperbolic cosine of x
Exp(x)	The result of e to the x power
Floor(x)	The next lowest whole integer of x
Log(x)	The logarithm (base e) of x
Log10(x)	The logarithm (base 10) of x
Sin(x)	The sine of x
Sinh(x)	The hyperbolic sine of x
Sqrt(x)	The square root of x
Tan(x)	The tangent of x
Tanh(x)	The hyperbolic tangent of x
Int(x)	The integer portion of a floating point value x
Rand(x)	A random number between 0 and (x - 1)
Percent(x)	Returns true x% of the time, else false
Min(x,y)	The minimum value of x and y
Max(x,y)	The maximum value of x and y
Mod(x,y)	The remainder left over after the division of x by y
Pow(x,y)	The value of x raised to the power of y
If(expr,x,y)	If expr evaluates to true, return x, else return y

Most of these functions are clear, but a few could use some additional explanation. The Percent() function can be used to select a random subset of your data, useful for extracting test data. To export approximately 5% of the data set at random, use the simple filter "Percent(5)". The Rand() function can be used to provide pseudo-random values in the export stream. The in-line If() statement can be used to convert one true/false expression into two different values, the first if the expression evaluates to true and the second if it evaluates to false.

In addition to the numeric fields, there are two "special" data types that can be used by GSSync in the filter, namely the DATE and TIME types. Regardless of any export formatting, dates and times are always converted to numeric values ("05/06/2009" is converted to 20090605, and "2:56 PM" is converted to 145600) and these numeric values are used in the expressions. As such, to compare with a date field, you should always compare with an 8-digit value. You can also determine JUST the year of a date field with the function "Int(DateField / 10000)".

The Record Filter is specified in the configuration file with two separate XML elements, where the first one specifies the filter, and the second one specifies whether the filter is active or not.

```
<RecordFilterExpression>ID &ge; 200000000</RecordFilterExpression>  
<RecordFilterActive>1</RecordFilterActive>
```

The record filter can be easily turned on or off using the RecordFilterActive switch. Change this switch in the configuration file to 0 to disable the filter, which is easier than removing or commenting out the filter string while testing.

The same Record Filter can be specified on the command line with one of the following two options:

```
/RF"ID &ge; 200000000"  
/RF"ID >= 200000000"
```

By default, the Record Filter is undefined, which provides no filtering. If you require filtering, you must provide a value for the filter expression. If your configuration file specifies the Record Filter and you wish to disable it, include /RF"" on the command line AFTER the configuration file.

Chapter 6: Interpreting the GSSync Log File

The GSSync Log File is used to keep track of everything that happens during a single run of GSSync. This provides you with the ability to review the processing run after it completes to verify that no unusual errors were seen, or to determine the cause of an error if a failure does occur.

Log File Options

There are a few options controlling the log file contained in an XML group called "LogOptions". The log file location simply specifies where the log data for this run should be stored. The Log Level is a number from 1 to 5 that tells GSSync how much data should be logged, based on the following chart:

Log Level	Logged Data
1	Required Messages Only
2	All Above and Error Messages
3	All Above and Warning Messages
4	All Above and Informational Messages
5	All Above and Debugging Messages

Specifying the Log File Location

The default log file name is GSSYNC.LOG, and a new file will be created in the current working directory of GSSync when the application starts. To specify a different log file location in the configuration file, use the following XML elements:

```
<LogFileName>GSSYNC.LOG</LogFileName>  
<LogFileAppend>0</LogFileAppend>
```

For the LogFileName value, you can specify either an absolute path (e.g. C:\LogFiles\GSSync.LOG) or a relative path (e.g. ..\Logs\MyFile.LOG). For the LogFileAppend value, the default of 0 will create a new log file, while changing this to 1 will append the log information to an existing log file (if one exists).

The Log File Location can be specified on the command line with the following option:

```
/LFcreateanew.LOG
```

If you prefer to append the log messages to an existing log file, then use /LA instead:

```
/LAappendtothis.LOG
```

If you have already set the log file name in the configuration file, you can also use the /LF or /LA switches on the command line (without a filename) to turn off or on the append functionality on the existing log file.

Specifying the Log Level

The Log Level is specified in the configuration file in the following XML element:

```
<LogLevel>3</LogLevel>
```

It can be specified on the command line with the following option:

```
/LL3
```

By default, the log level is 3, which records errors and warnings only. If you want to increase the log level, you can specify a level of 4 to include informational messages, or 5 to include all debugging messages. Please note that running GSSync in Debug mode will include a lot of information and will reduce performance substantially.

Remember that the command line options will override the configuration file options when both are present. This is an ideal reason to leave the Log Level set to 3 inside the configuration file, as you can always add /LL5 to the end of your command to run a replication cycle with debugging messages enabled.

Important Note: If you are having issues with parsing the configuration file, then include the /LL5 switch on the GSSync command line **before** the /CF option. This will enable debugging messages to the command window for the configuration file parser.

Log File Details

Each line of the log file will consist of several items, separated by commas. This should allow you to read the log file into a spreadsheet or other database application for further analysis. Here's an example of a brief log:

```
1,2018-12-18,17:59:45,Starting Process GSSync Version 1.60: 12/17 (C)2018  
1,2018-12-18,17:59:45,Created By Goldstar Software Inc.  
1,2018-12-18,17:59:45,GSSync is performing a full export only.  
1,2018-12-18,17:59:48,Processed 1500 records.  
1,2018-12-18,17:59:48,Process Terminating Normally With Status 0.
```

The first value indicates the severity level of the message. Messages of Level 1 are always reported in the log. Other messages are reported ONLY if the value meets or exceeds the current log level.

The second and third values represent the date and time of the message. This provides you with an idea of exactly when each item occurred, and it allows you to calculate cycle times easily by comparing the starting and finishing times.

The last value is the log message itself, which can contain any readable text.

The very last line in the GSSync Log File will always indicate the return code if the code terminated normally (either successfully or with an error). If you do not see a "Terminated Normally" line with the status code, then the GSSync application failed to complete normally.

If you experience any problems with GSSync and need to contact Goldstar Software for help, please be sure to run the process with a Log Level of 5 (Debugging) and be ready to provide that log file upon request.

Changing the Display Language

GSSync supports displaying status and error messages in several different languages. English (0) is used by default, but you can also select Spanish (1), Italian (2), French (3), or German (4) at this time.

The Display Language can be specified in the XML configuration file:

```
<DisplayLanguage>1</DisplayLanguage>
```

From the command line, the following switch can also be specified:

```
/DL1
```

The value should be used to indicate the language you wish to use.

Chapter 7: Reviewing Replication Examples

With all of the possible options for handling a replication data set with GSSync, where should you start? To answer this question, we have created a few sample replication scenarios that should provide you with a baseline to get started.

The database that we are using for this replication is the DemoData database. For Pervasive PSQL v9 and earlier, this database is installed in the C:\PVSW\DEMODATA directory. For Pervasive PSQL Summit v10 and v11, this database is installed to the C:\Documents and Settings\All Users\Application Data\Pervasive Software\PSQL\Demodata folder or to the C:\ProgramData\Pervasive Software\PSQL\Demodata folder. For Actian PSQL v12 and newer, this database is installed to the C:\ProgramData\Actian\PSQL\Demodata folder. For Actian Zen v14 and newer, this database is installed to the C:\ProgramData\Actian\Zen\Demodata folder. This sample database allows you to leverage these tests immediately and see the same results with minimal effort. If you are not familiar with the DemoData database, you may wish to explore it with the Control Center before going through these examples.

For the purpose of these examples, we will assume that you are running these commands from a command line window set to the Demodata folder as your current directory, and the same folder will be used to store all configuration and other files.

Btrieve Replication with GSSync Metadata

For our first example, we will start with a very simple replication environment. In this case, we are running a simple Btrieve application for which we have no file definition metadata, and we don't want the added expense of using DataExchange. The database file that we wish to replicate is called PERSON.MKD, and we want to replicate it with GSSync metadata to another copy of the same file in the directory C:\BACKUP.

Configuring Initial Replication

We can configure the initial replication environment through two separate steps, namely copying over the initial data file and initializing the GSSync metadata. Both of these can be easily done from the command line:

```
COPY PERSON.MKD C:\BACKUP\PERSON.MKD
GSSYNC /BFPERSON.MKD /MCPERSON.GS /ON
```

Updating the Data Set

Now that the initial seed data has been generated, we can make changes to the Person table in the PCC. When we are ready to replicate those changes, we only need one more command:

```
GSSYNC /BFPERSON.MKD /MSPERSON.GS /OBC:\BACKUP\PERSON.MKD
```

We are assuming that the file already contains system data, but it might be safest to confirm this also by executing a `BUTIL -STAT PERSON.MKD` command and checking for the presence of the System Data field. If there is no system data in the Btrieve file, then you can use the record position value by changing the `/MS` switch to `/MG`.

We can continue to make changes and replicate data as needed. As you can see, this is a very simplistic replication environment, using only command line options to specify the parameters. This table, since it contains only 1500 records by default, is very small and quite amenable to using the GSSync metadata option to replicate as frequently as needed.

CSV Replication with Archive Log Metadata

In our second example, we're going to replicate the Person table from the Demodata database to a comma-delimited file, but we're going to use the Archive Log information that the database can create for us to detect changes.

Configuring Initial Replication

The initial replication configuration consists of setting up the Archive Log on the PERSON.MKD file. Remember that when you replicate data via a CSV file, it creates the text data, but does not send that data anywhere, so it is really just the first part of any real "solution" for replication.

To set up the archive log, we first create a file `C:\BLOG.CFG` and put one line in it:

```
C:\PVSW\DEMODATA\PERSON.MKD
```

This line tells the database engine to start logging changes on that file. Note that we are using the simpler path from the older engines. If you are using PSQlv10 or newer, be sure to specify the complete path to the file. Since we did NOT specify the log file, the database engine automatically assumes that the log file is PERSON.LOG -- the same base filename plus the LOG extension. Restart the engine, and the database will be ready to start tracking changes to this file.

Updating the Data Set

Before running the update, you may wish to use the PCC to make some changes to the PERSON table, perhaps adding a record, inserting a new record, and deleting a record or two. As we'll need access to the archive log file, be sure to exit the PCC when you are done and make sure that all users are out of the PERSON.MKD file. This should ensure that the archive log file is properly closed.

When we are ready to run the replication process, we can again specify all of the options from the GSSync command line or use a configuration file. We're going to opt for a command line for this example, since it's still fairly straightforward.

```
GSSYNC /DD. /DTPerson /MAPerson.LOG /OCPerson.CSV
```

Note that we are now specifying the database and table name instead of only the Btrieve file name, because source metadata is required for the CSV export option. The "/DD." option indicates that we should use the DDF's in the current directory, and the /DT option indicates which table metadata that should be used to interpret the data blocks.

The net result from this command is that the archive log file (PERSON.LOG) is read one entry at a time, and for each insert or update, the record data is sent to the export file PERSON.CSV, following the metadata definitions in the DDF's.

What about the deleted records? As no deleted export table was explicitly specified on the command line, the file Del_Person.CSV is created, and the deleted records are written to that location by default.

For a "real" replication solution, a more likely scenario would be to shut down the database engine, move (not copy) all of the log files from the log directory to a new folder, and then restart the engine. This allows the engine to restart with new log files, allowing access from users again. You can then safely start the replication process to replicate all changed records without worrying about somebody accessing the log file at the wrong time.

SQL Script Replication with PDC Metadata

In this example, we are going to use DataExchange metadata to replicate a database file to a remote database server to which we do not have direct access. To make this happen, we will use a SQL script file that can easily be transferred to the other server and executed in an Oracle environment.

The first prerequisite, of course, is to have DX properly installed, configured, and activated for the database. The steps for this are outside the scope of this manual, but are covered in both the *Getting Started with Actian DataExchange* product manual as well as Goldstar Software's *Implementing Actian DataExchange* course manual. We completed, we should have the activated database sitting in the C:\PVSW\DEMODATA folder, and the replication data sitting in the subdirectory called DX_DEMODATA. DataExchange "change capture" should be active and collecting changes into the PDC files in the latter directory.

The second prerequisite is to have good data file definitions. As with the previous example, we will be using the DDF definitions for the Person table.

The final prerequisite is that we need to have the target database set up already. In the interest of saving space, we have created the Oracle database on the web server to include five fields only, including the System Data value and four columns from the data structure: ID, Last_Name, First_Name, and Date_Of_Birth. The primary key should be

set up as the System_Data column, as this is how we will be updating it. Supplemental keys on other fields, such as ID, Last_Name may be present to make data retrieval easier, but are not required.

Configuring Initial Replication

As previously indicated, we cannot do this type of replication from the command line only, due to the length of the SQL statements. Instead, we must use a configuration file for this replication cycle. Instead of listing the entire file, we are going to ONLY list those elements that must be set up for this to work.

As DataExchange is going to be maintaining the metadata, there are two ways of doing the initial replication. We can either do a full export of the data from GSSync and import the data by just running the script, or we can do a simple ODBC data extraction and import directly into the target database. We're going to use the former option here, as this is also a good way to verify that all of the data currently in the database will properly pass to Oracle. (Remember that dates in PSQL/Zen databases are not always perfect.)

In order to make all of this happen, we need to define each of the three sections -- the Source Options, the Metadata Options, and the Export Options. We will create these in a file called PERSONSTART.XML, indicating that this is the XML file to start up the Person replication.

We'll start with the Source Options group to define where we are getting the data from. Again, we have DDF's in the current directory, so let's use those. We're going to add one more option here -- I really want the data to be imported into the target database in order by Last_Name/First_Name, so I want to specify the use of Btrieve Key #1 when reading the data file.

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
  <BtrieveKey>1</BtrieveKey>
</SourceOptions>
```

For the Metadata Options group, we want to specify NO replication metadata, because we are replicating ALL of the data, and we don't want to deal with the extra overhead of looking at each PDC record.

```
<MetadataOptions>
  <MetadataType>NONE</MetadataType>
</MetadataOptions>
```

The last section that we must define is the Export Options. Again, we are targeting a SQL script file, so we'll need to provide all of the information needed to provide the script, including where to save it. On any full export like this, there is no metadata, so

there is no way to detect changed or deleted records. As such, we will save some time (and space) by not specifying the SQL statements for those operations.

```
<ExportOptions>
  <ExportType>SQL</ExportType>
  <ExportSQLFile>PERSONSTART.SQL</ExportSQLFile>
  <ExportODBCInsertQuery>
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
      {{First_Name}}, {{Date_Of_Birth}} );
  </ExportODBCInsertQuery>
  <DateExportFormat>11</DateExportFormat>
  <DateFieldFilterOptions>11</DateFieldFilterOptions>
</ExportOptions>
```

You will also notice a few other options have been included here. We are explicitly selecting the date format to use the Oracle function TO_DATE() so that the fields are stored as real dates when they get to the target database. We are also filtering the dates so that any invalid dates (including dates or 00/00/0000) get exported as NULL values.

We can save this into our configuration file, PersonStart.XML, and then run the process with the following command:

```
GSSYNC /CFPersonStart.XML
```

When this process finishes, we will have a new SQL Script file (PERSONSTART.SQL) which contains all of the starting data for the database in a series of INSERT statements, one after the other. We can then send that file over to the target database and run it, verifying that all data is imported into the database correctly.

Updating the Data Set

Updating the result set is a bit more complicated, since we need to also define the PDC metadata options, as well as define how to handle both DELETE operations and UPDATE operations. We'll create a new configuration file called PERSONDELTA.XML, as this sample will replicate changes (or deltas) to the Person table.

The source data does not change in this example from the starting data source, but the Btrieve key number is no longer required, since we will be replicating from the PDC change logs and reading the records based on the last changed timestamp:

```
<SourceOptions>
  <DatabaseDirectory>.</DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
</SourceOptions>
```

For the Metadata Options group, we now specify PDC replication metadata, which also requires the specification of the correct PDC table.

```
<MetadataOptions>
  <MetadataType>PDC</MetadataType>
  <PDCMetadataFile>DX_Demodata\PDCTPerson.MKD</PDCMetadataFile>
```

```
</MetadataOptions>
```

Finally, we see what additional settings need to be in the Export Options. All of the details from above are copied, but we also need to specify the DELETE and UPDATE scripts, and we need to change the target SQL file to PERSONDELTA.SQL.

```
<ExportOptions>
  <ExportType>SQL</ExportType>
  <ExportSQLFile>PERSONDELTA.SQL</ExportSQLFile>
  <ExportODBCInsertQuery>
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
      {{First_Name}}, {{Date_Of_Birth}} );
  </ExportODBCInsertQuery>
  <ExportODBCDeleteQuery>
    DELETE FROM Person SystemData = {{SKEY}};
  </ExportODBCDeleteQuery>
  <ExportODBCUpdateQuery>
    DELETE FROM Person SystemData = {{SKEY}};
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
      {{First_Name}}, {{Date_Of_Birth}} );
  </ExportODBCUpdateQuery>
  <DateExportFormat>11</DateExportFormat>
  <DateFieldFilterOptions>11</DateFieldFilterOptions>
</ExportOptions>
```

As with the initial replication, running the process from the command line is now very simple, but we DO need to provide one additional element, the time of the last replication. At this time, there is no way for GSSync to track this information itself, so you must maintain this information externally and provide the value on the command line.

```
GSSYNC /CFPersonDelta.XML /ST2009050608
```

When this process finishes, we will have a new SQL Script file (PERSONDELTA.SQL) which contains all of the data for each changed record in the database since 8AM on May 6, 2009. We can then send that file over to the target database and run it, allowing the changes to propagate into the target database.

For ongoing replication, it is important to automate some of these steps. For example, if you expect to automatically replicate all data every hour, then you'll need a way to generate the 24 command line options for each hour's replication cycle. This can be done manually by creating 24 separate batch files that work with each hourly value, but this seems a bit tedious and error-prone. A much better solution is to build a batch file and maintain the time on the last synchronization in an external file. This can be done in a batch file by extracting the date and time and storing the result into a text file on the disk, as in the example shown below:

```
REM Read in the previous replication timestamp
REM This may need to be manually set for the first run
SET /P LastSync=<PersonLastSync.TXT

REM Get the Current Hour and Minutes Values
```

```

FOR /F "tokens=1,2 delims=:" %%A IN ("%Time%") DO (
    SET Hours=%%A
    SET Minutes=%%B
)
SET /A Hours = 100%Hours% %% 100
REM If the AMPM switch is set, adjust for PM
ECHO.%Minutes% | FIND /I "P" >NUL && SET /A Hours += 12
SET Minutes=%Minutes:~0,2%
REM Set up the leading zero, if needed
if "%Hours:~1,1%"==" " set Hours=0%Hours%
REM Build the final Date and Time Strings
set datestr=%date:~-4,4%%date:~-10,2%%date:~-7,2%
REM Now run GSSync with the previous value
GSSYNC /CFPersonDelta.XML /ST%LastSync%
IF %ERRORLEVEL% == 0 goto Success
:Failed
REM Insert your error handling code here...
Goto End
:Success
REM Write the last sync timestamp to the file for next time.
echo %datestr%%Hours% > PersonLastSync.TXT
:End

```

Your own implementation may vary, especially for environments that do not use the "PM" designation for times after Noon. You may want to add ECHO statements to understand better how this script works before using it. Additionally, you should also include your error handling code after the ":Failed" label to address the case of a failed replication cycle.

Note that you can simplify the process (at the expense of additional data transfers) by simply replicating ALL records that have changed on a given day by providing ONLY the date information (and excluding the Hours value). Of course, if a record was already replicated and had not changed again that day, it will be re-replicated on every subsequent cycle that day, but for databases with minimal changes taking place, this extra overhead should be negligible.

ODBC Replication with GSSync Metadata

In this final example, we will look at a complete ODBC replication process to an ODBC data source. This would commonly be used for moderately-sized data sets which do not have DataExchange available, but which still need to export data to a target database. Since the ODBC driver is available on the server on which GSSync is running, we can eliminate the separate script import step that is needed on the SQL Script export option. As previously indicated, the ODBC replication and the SQL replication are just about identical -- the main difference is that we need to provide a valid ODBC connection string to connect to the target database.

Configuring Initial Replication

Again, we cannot use a simple command line replication command, because of the need for the SQL statements, so we will build a configuration file called PERSONSTART.XML and use that file to define the Source Options, Metadata Options, and Export Options.

We will use the same Source Options that were used in the previous example -- since this information has not changed.

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
  <BtrieveKey>1</BtrieveKey>
</SourceOptions>
```

For the Metadata Options group, we again want to replicate ALL data to get started, and we want to write the data into a GSSync Metadata file called PERSON.GS.

```
<MetadataOptions>
  <MetadataType>GSCREATE</MetadataType>
  <GSMetadataFile>PERSON.GS</GSMetadataFile>
</MetadataOptions>
```

As with the SQL export, this export will create INSERT results only, so we can take some shortcuts in defining the scripts. Our target DSN has already been created and called TargetDemo, and the target table Person has already been created inside that database. As previously noted, it is usually best to link to an external ODBC Connection String File, but this is not being done for the sake of simplicity in the example.

```
<ExportOptions>
  <ExportType>ODBC</ExportType>
  <ExportODBCConnectionString>DSN=TargetDemo</ExportODBCConnectionString>
  <ExportODBCInsertQuery>
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
      {{First_Name}}, {{Date_Of_Birth}} );
  </ExportODBCInsertQuery>
  <DateExportFormat>11</DateExportFormat>
  <DateFieldFilterOptions>11</DateFieldFilterOptions>
</ExportOptions>
```

As with the SQL export, a few additional options have been specified to handle the date fields properly. We can then run GSSync with the PersonStart.XML file with the following command:

```
GSSYNC /CFPersonStart.XML
```

When this process finishes, the target database should be fully populated with the starting data for the database, and the GSSync Metadata table (PERSON.GS) should be completely initialized for each record in the primary database.

Updating the Data Set

Updating the result set is again similar to the process discussed above. We are again creating a new XML file, called PERSONDELTA.XML, to handle replicating the changes in the environment.

The source data does not change in this example from the starting data source, but now we want to specify the Btrieve Key for the System Data (Key 125), since it can improve the performance of the GSSync metadata scanning:

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
  <BtrieveKey>125</BtrieveKey>
</SourceOptions>
```

For the Metadata Options group, we now change it to GSSYNCSYS replication metadata so that we replicate only the changed data:

```
<MetadataOptions>
  <MetadataType>GSSYNCSYS</MetadataType>
  <GSMetadataFile>Person.GS</GSMetadataFile>
</MetadataOptions>
```

Again, if the file does not have System Data enabled, you can enable a slightly more tenuous replication by using the record position, which is a MetadataType of GSSYNC.

The Export Options also remain mostly the same, with the addition of the DELETE and UPDATE scripts. We also add one more switch for stripping the semicolons, which is needed for some databases.

```
<ExportOptions>
  <ExportType>ODBC</ExportType>
  <ExportODBCConnectionString>DSN=TargetDemo</ExportODBCConnectionString>
  <ExportODBCInsertQuery>
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
    {{First_Name}}, {{Date_Of_Birth}} );
  </ExportODBCInsertQuery>
  <ExportODBCDeleteQuery>
    DELETE FROM Person SystemData = {{SKEY}};
  </ExportODBCDeleteQuery>
  <ExportODBCUpdateQuery>
    DELETE FROM Person SystemData = {{SKEY}};
    INSERT INTO Person VALUES ( {{SKEY}}, {{ID}}, {{Last_Name}},
    {{First_Name}}, {{Date_Of_Birth}} )
  </ExportODBCUpdateQuery>
  <ExportODBCStripSemicolons>1</ExportODBCStripSemicolons>
  <DateExportFormat>11</DateExportFormat>
  <DateFieldFilterOptions>11</DateFieldFilterOptions>
</ExportOptions>
```

The replication updates are then handled with a simple command:

```
GSSYNC /CFPersonDelta.XML
```

When this process finishes, the ODBC target database should have been completely updated with all of the changes from the source database!

Archiving with Btrieve Replication with a Filtered Result Set to CSV

For this example, we will perform a simple Btrieve-to-Btrieve replication once again, but we will add the complexity of an export filter. We are going to replicate only those records that have a date older than a specific value, which would be handy for use in an application that is archiving data. In the Person table that comes in Demodata, the only date field is the Date_Of_Birth field, so this example is somewhat contrived, but you should be able to extend this to just about any other environment.

Again, we start by reviewing the parameters for running this tool. We know that we want to do a Btrieve replication, but because we are using filters, we are required to provide the source data definitions. This means that we will provide the data dictionaries and table name for the source data.

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
</SourceOptions>
```

We're going to ignore the metadata for this example, opting instead to push ALL records to the target database.

```
<MetadataOptions>
  <MetadataType>NONE</MetadataType>
</MetadataOptions>
```

For the target data, we'll use CSV for simplicity, and then we'll adding a record filter that will exclude all records where the Date_Of_Birth is in 1970 or newer. We'll also disable the CSV Header Row, since we'll be importing the data directly into the data warehouse using a tool that doesn't need it. We are also going to add the timestamp of when this record was exported, so that we can keep track of when the record was archived from the database.

```
<ExportOptions>
  <ExportType>CSV</ExportType>
  <ExportCSVFile>PERSON.CSV</ExportCSVFile>
  <ExportCSVHeaderRow>0</CSVHeaderRow>
  <ExportExportTimeStamp>3</ExportExportTimeStamp>
  <RecordFilterExpression>Date_Of_Birth &lt; 19700101</RecordFilterExpression>
  <RecordFilterActive>1</RecordFilterActive>
</ExportOptions>
```

Notice that we are NOT specifying the DateExportFormat option here. When this option is omitted, the default date export format of "mm/dd/yyyy" is used. As you might notice, though, this format is NOT valid for a numeric comparison by the record filter. When a record filter acts on a date field, it ALWAYS handles the date as a numeric date value with the format YYYYMMDD for comparison purposes.

To run the command, we simply use the normal configuration file format:

```
GSSYNC /CFPersonDelta.XML
```

Of course, we can ALSO do this completely from the command line if we wanted to do so. Applying the configuration settings into a single command line would look like this:

```
GSSYNC /DD. /DTPerson /MN /OCPerson.CSV /RF"Date_Of_Birth < 19700101" /TE3 /OH0
```

When deciding between a configuration file and a command line, you should always weigh the benefits of both options. Replication processes that may need to be run again in the future would make a good candidate for the configuration file, whereas one-time exports might be best for the command line.

Purging Data from a Filtered Result Set

Let's now think a bit outside of the box. Say that we have an application that has been running for 14 years, and the data all goes back to when it was first started. However, the data from everything older than 7 years ago is no longer needed on-line, and we want to move it into an archive database and delete it from the production database in order to save disk space and improve system performance. Unfortunately, the application developer never created an archive/purge module to perform this clean-up task for us. Can we leverage GSSync to create an archive/purge module for our database environment?

Of course, the answer is a qualified YES. Deleting data from a working system can always create bad data links and other problems in the data set, so you may have to carefully work within the database environment to make sure that such a purge won't cause irreparable harm to the data. If the app will work, though, then it should be safe to continue.

In order to make this magic happen, we are going to run TWO separate extracts. The first extract will be identical to the one above to get the data out that matches the "purge" criteria. The second extract is going to handle getting rid of the records for us automatically.

Let's examine the Archive Step first with our PersonArchive.XML file:

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
</SourceOptions>

<MetadataOptions>
  <MetadataType>NONE</MetadataType>
</MetadataOptions>

<ExportOptions>
  <ExportType>BTRV</ExportType>
  <ExportBTRVFile>PERSONArchive.MKD</ExportBTRVFile>
  <RecordFilterExpression>Date_Of_Birth &lt; 19700101</RecordFilterExpression>
  <RecordFilterActive>1</RecordFilterActive>
</ExportOptions>
```

Before we run this process, we need to create the new "archive" file, and then we'll push ONLY those records that match the filter into the newly-created table:

```
BUTIL -CLONE PERSONArchive.MKD PERSON.MKD
GSSYNC /CFPersonArchive.XML
```

Once the archive is complete, we can now do the purge with PersonPurge.XML:

```
<SourceOptions>
  <DatabaseDirectory>./DatabaseDirectory>
  <DatabaseTable>Person</DatabaseTable>
</SourceOptions>

<MetadataOptions>
  <MetadataType>NONE</MetadataType>
</MetadataOptions>

<ExportOptions>
  <ExportType>ODBC</ExportType>
  <ExportODBCConnectionString>DSN=DEMODATA</ExportODBCConnectionString>
  <ExportODBCInsertQuery>
    DELETE FROM Person WHERE ID = {{ID}};
  </ExportODBCInsertQuery>
  <RecordFilterExpression>Date_Of_Birth < 19700101</RecordFilterExpression>
  <RecordFilterActive>1</RecordFilterActive>
</ExportOptions>
```

So how is this going to work? The Archive step that we just did identified all of the records that we want to delete. As such, we make NO changes to the source, metadata, or filtering information. What we DO change is the export side. Instead of writing the data to a new Btrieve file as would *normally* be done, we instead connect to the ODBC database "Demodata" -- which happens to be the same database we just connected to for reading the data -- and instead of inserting the data records that match the criteria, we use the primary key (the ID field, in this case) to DELETE the records. The net result is that each matching record is deleted as it is found -- resulting in a purge!

Obviously, this is a contrived example, but imagine using it on the InvoiceDate field for an invoice database, LastOrderDate for a customer database, or another such fields, and you can see how flexible this solution really can be.

Chapter 8: GSSyncImport

If you decide that you need to use a binary Queue file for your data transfer, then you will need to leverage a second utility (called GSSyncImport) to import the data into your target database. At this time, GSSyncImport only supports importing queue files directly to a target Btrieve file. If you need to migrate data into a different target format, then you should modify your GSSync output format accordingly.

Running GSSyncImport

Launching the GSSyncImport application must be done from the Windows command line interface. Attempting to run GSSyncImport by double-clicking from a graphical (i.e. GUI) window will result in a quick flash of a black screen and no noticeable results as the help screen scrolls by and then closes.

You can start your own command line interface window by selecting Start/Run and entering CMD into the Run dialog box. From the command box, simply enter "GSSYNCIMPORT" and you will see the default command syntax screen, as shown here:

```
GSSyncImport Version 1.66: 06/27 (C)2019 Goldstar Software Inc.
```

```
The command syntax is as follows:
```

```
GSSYNCIMPORT [Options]
```

```
Where [Options] includes the following:
```

```
/DN<DBName>: Override DBName (Default is read from file).  
/DT<Tablename>: Override Table Name (Default is read from file).  
/LA<Filename>: Appends to the output log file (Default is GSSYNCIMPORT.LOG).  
/LF<Filename>: Creates a new output log file (Default is GSSYNCIMPORT.LOG).  
/LL#: Specifies the Log Level to one of the following.  
    1 = Minimal Logging  
    2 = Error Messages and All Above  
    3 = Warning Messages and All Above (Default)  
    4 = Informational Messages and All Above  
    5 = Debug Messages and All Above  
/OB<Filename>: Specifies target Btrieve file (Default is read from file).  
/ON: Perform Output to No Btrieve file, reporting what would be done.  
/OW<owner> specifies the Btrieve file owner name.  
/QF<Filename>: Specifies input filename for the binary Queue file.
```

If you look carefully, much of this should be familiar, as GSSyncImport options are very similar (or identical) to those used by the GSSync utility.

Using the Command Line Options

Unlike GSSync, the GSSyncImport tool does not require nor use a configuration file. Instead, it leverages command-line switches for all operational control. Because of this, you will always need to specify the options on the command line. At a minimum, the /QF option **must** be provided, but all other switches are optional.

- **/QF:** This switch specifies the binary queue file to load. While the file name and extension can be anything you wish, we recommend that you use the base filename of the table being synchronized, with an extension of BQF. This will help make it clear which files in your environment are binary queue files.

If you only provide this switch, then the GSSyncImport tool will read the specified queue file and import the data back into the target file (that is stored inside the file) on the current server. You can use the optional switches to modify this behavior. Let's briefly review the use of each switch here:

- **/DN:** Use this switch if you want to specify a target Database Name. If this switch is NOT used, then the Database name stored in the queue file (if present) will be used to indicate the target file location.
- **/DT:** Use this switch if you want to specify a target Database Table. If this switch is NOT used, then the Table name stored in queue file (if present) will be used to indicate the target file location.
- **/LA:** Use this switch if you want to append the log from this run to an existing log file. You can provide an optional pathname to the log file, too.
- **/LF:** Use this switch to specify the log file name and to erase the log file before starting this run. (This is the default option.)
- **/LL:** This switch sets the log level for detailed logging, where 3 is the default.
- **/OB:** Specify this switch, along with the Btrieve file path, to indicate which file the data should be loaded into. This file MUST already exist, and it MUST have system data. For optimal flexibility, no validation is done on the file name, so you should ensure that the file is the proper one for your queue file before starting! If this option is specified, it will override the database name and table names from the file AND provided on the command line, so do not use this with the /DT and/or /DN options at the same time!
- **/ON:** This switch disables the writing to the Btrieve file. The net result is that the log file is written based on the source data only, which provides a simple way to export the source filename and the number of records from any binary queue file.
- **/OW:** If the target Btrieve file has an owner name set, use this switch to specify the owner name so that the target file can be properly updated.

For additional information, see the primary GSSync documentation.

Chapter 9: SyncAllTables

In many cases, GSSync will be used to replicate data from a large number of files. With a large number of files, however, comes a large number of unique configurations. It is rarely feasible to create a different Configuration file for each table, though, so using the command line switches to override the CFG file settings can be invaluable to reducing your setup efforts.

At the same time, replicating hundreds of files still means setting up batch files with hundreds of lines with the proper configuration switches inside. Luckily, there are ways to automate much of this, and SyncAllTables is one such example. This tool was specifically built to facilitate replication on a two-hour schedule, but you can always use it as a starting point and modify this to suit your own requirements.

Configuring SyncAllTables

Before you try to use this tool, you should first edit the source file (SyncAllTables.VBS) and edit the default settings and location indicated towards the top of the file (between the lines with the asterisks in them), and then save your changes.

For each setting, the default value gives you a clue as to what is needed. “OtherSwitches” can be used to provide additional switches to GSSync without having to modify the core code at the bottom of the script. “ExcludeFiles” is a comma-delimited list of files that should be skipped in the replication process, and it can be useful for skipping files that are in the data dictionary but which do not need to be replicated.

Running SyncAllTables

To run the SyncAllTables tool, open a command line and issue the following command:

```
CScript SyncAllTables.vbs
```

This command will read each of the database table names from your defined database and execute a GSSync command line to replicate it for the indicated time period.

Like the other tools in this package, the process can be altered through the use of command line options.

Using the Command Line Options

SyncAllTables supports the use of some command line options that can easily reconfigure it on the fly.

- /database:dbname: Use this switch to change the database name you want to read tables from.
- /PDC:PDCLocation: When using PDC files for metadata, you can override the PDC file location with this option.

- `/target:targetpath`: Sometimes you want to just redirect data from a normal replication cycle to a test server. Use the `/target` switch to specify the new target path to use.
- `/sync:<type>`: The `/sync` switch allows you to specify the type of the synchronization you'd like to do. If you do not specify anything, SyncAllTables will synchronize data changes from 4 hours old to 2 hours old. You can also specify one of these options:
 - Now: synchronizes changes from 2 hours ago until the current time.
 - This: synchronizes changes from 2 hours ago until the start of the current hour.
 - Previous: synchronizes the previous replication window – from 6 hours old to 4 hours old. This can be used to “make up” for a single missed replication window.
 - Today: synchronizes all changes made today.
 - Yesterday: synchronizes all changes made yesterday.
 - ThisMonth: synchronizes all changes made from the start of the month until the current time.
 - LastMonth: synchronizes all changes made during the previous month.

The last types are helpful if you ever have to restore the target environment from a backup and resynchronize everything since that backup. If you need additional types, please contact us and we can assist with the process of updating the script file.

Extending This Tool

SyncAllTables is written in VBScript and should be fairly easy to modify or extend as needed. If you have an extremely-useful idea, please consider submitting your script to Goldstar Software for inclusion in the product.

Chapter 10: The GSSync Scheduler

Included with the GSSync package is a special PowerShell utility script known as GSSyncScheduler.ps1. This script provides a way to schedule the execution of GSSync commands directly on your Windows server environment using a simple PSQL/Zen database to host the configuration data.

The current version of the GSSync Scheduler is single-threaded and can run only one instance of GSSync at a time, but it is possible to run multiple instances of the scheduler (each with its own database) to effect true multi-tasking. (A multi-threaded version of the scheduler is being considered for a future release.)

Preparing to Run GSSync Scheduler for the First Time

Before running the GSSync Scheduler tool, you must configure the environment properly. This requires two steps, namely ensuring that PowerShell will run unsigned scripts and creating the configuration database.

Configuring PowerShell to Run Unsigned Scripts

Launching the GSSync Scheduler script is done from simple command prompt using the PowerShell executable. At this time, the PowerShell script is unsigned, so you will need to configure your server to run unsigned PowerShell scripts. You can do this by starting a PowerShell window and executing the statement:

```
Set-ExecutionPolicy unrestricted
```

For additional information on security options and signing, please see:

<https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.security/set-executionpolicy?view=powershell-7.2>

Creating the Configuration Database

The scheduler configuration database must be created within your PSQL/Zen environment. Start by creating an empty working directory somewhere on the server. Then, using the Control Center, right-click the server name in the tree view and select New/Database. For the database name, enter "GSSyncDB" to use the default name. (You can also enter a different name here if you prefer, but you will need to specify the database name when launching the scheduler.) In the second box, browse to provide the path to the empty folder you just created. The rest of the database options are less critical, though we do recommend creating a v2 metadata database on PSQL v10 and above. Click OK to create the new database.

Running the GSSync Scheduler the First Time

Launching the GSSync Scheduler script the first time will create the database tables needed for it to run properly. This can be done from PowerShell directly, or you can run it from a command line with the following command:

```
PowerShell .\GSSyncScheduler.ps1
```

By default, GSSync Scheduler will look for a configuration database called “GSSyncDB” on the local machine. If your scheduler database is located on a remote server, then use the -DBServerName option to specify the new server name, like this:

```
PowerShell .\GSSyncScheduler.ps1 -DBServerName servername
```

If you need to specify a different database name, then use the -DBName option to specify the new database name, like this:

```
PowerShell .\GSSyncScheduler.ps1 -DBName databasename
```

If you get any errors back, then verify your server and database name values.

Assuming the named database exists on the server indicated, then the tables Jobs and Log will be created within that database. You can now jump to the section on Creating Jobs to start building your schedule.

Understanding the GSSync Scheduler Database

The GSSync Scheduler database consists of two tables, the Jobs table and the job Log table. Both tables are created automatically the first time the scheduler runs, but the commands used to create this are included here for completeness.

The Jobs Table

The Jobs table contains the information about each job in the database. This table is defined as follows:

```
CREATE TABLE Jobs (
    ID                                IDENTITY,
    Name                             VARCHAR(32) NOT NULL,
    Priority                          INTEGER DEFAULT 0,
    LastRunStatus                     INTEGER,
    LastRunStartTime                  TIMESTAMP,
    LastRunEndTime                    TIMESTAMP,
    NextStartTime                     TIMESTAMP,
    RunInterval                       INTEGER,
    ScheduleValidFrom                TIME,
    ScheduleValidUntil               TIME,
    LogLevel                         INTEGER,
    LogFilePath                       VARCHAR(128),
    ConfigFilePath                    VARCHAR(128),
    Override1                         VARCHAR(50),
    Override2                         VARCHAR(50),
    Override3                         VARCHAR(50),
    Override4                         VARCHAR(50),
    Override5                         VARCHAR(250),
    SyncStartTimeReference            TIMESTAMP,
    SyncEndTimeReference              TIMESTAMP
);
CREATE UNIQUE INDEX NameKey ON Jobs(Name);
CREATE INDEX PriorityName ON Jobs(Priority, Name);
CREATE INDEX PriorityTime ON Jobs(Priority, NextStartTime);
CREATE INDEX TimePriority ON Jobs(NextStartTime, Priority);
```

The fields include:

- **ID:** A unique identity field for the Job records. When creating a new job, specify 0 for this field to assign the next number automatically.
- **Name:** The name of this job can be up to 31 characters. This field is unique to eliminate confusion in the displayed data, but you can change it at any time without impacting the scheduler.
- **Priority:** This INTEGER field is used to order jobs in the scheduler. A Priority of 0 is the default so that new jobs are considered disabled until specifically enabled. Jobs with a Priority field greater than 0 are executed in numeric order of priority, so you can use a simple 0/1 for disabled/Enabled, or you can force some tables to be executed first with a simple 1,2,3 ordering, or perhaps 10,20,30, or even 1000,2000,3000 to leave more room for implementing changes later on.
- **LastRunStatus:** This INTEGER field indicates the status returned from GSSync the last time this job was started by the scheduler. This value is set to NULL when a job is launched and reset when the job finishes, so you can detect when a job has not completed normally.
- **LastRunStartTime:** This TIMESTAMP field indicates the last time this job was started by the scheduler. It is set immediately prior to starting the job.
- **LastRunEndTime:** This TIMESTAMP field indicates the last time this job completed. This column is also set to NULL when the job starts and is updated when the job finishes.
- **NextStartTime:** This TIMESTAMP field indicates the next time this job should be started. This column is updated when the job completes and is set to the sum of the LoopStartTime field plus the number of minutes indicated in the RunInterval column.
- **RunInterval:** This INTEGER field indicates the number of minutes between automated jobs in the scheduler. A value of 0 can be used here to run a job ONE time and then change its priority to 0 (Disabled).
- **ScheduleValidFrom:** This TIME field indicates the time of day AFTER which this job can be run by the scheduler. A NULL value disables the schedule entirely, and the job will run every time.
- **ScheduleValidUntil:** This TIME field indicates the time of day BEFORE which this job can be run by the scheduler. A NULL value disables the schedule entirely, and the job will run every time.

- **LogLevel:** This INTEGER field indicates the LogLevel to send to GSSync through the /LL configuration field on the command line. This allows you to enable a higher logging (such as debug) on a single job within a schedule. If this column is NULL, then no /LL option is sent on the command line, and GSSync will use the setting from the GSSync.CFG file instead.
- **LogFilePath:** This 127-character STRING field indicates the LogFilePath to send to GSSync through the /LF configuration field on the command line. This allows you to configure a special log file on a single job within a schedule. If this column is NULL, then no /LF option is sent on the command line, and GSSync will use the setting from the GSSync.CFG file instead (or GSSYNC.LOG if no setting is found there).
- **ConfigFilePath:** This 127-character STRING field indicates the ConfigFilePath to send to GSSync through the /CF configuration field on the command line. This allows you to configure a special configuration file on a single job within a schedule. If this column is NULL, then no /CF option is sent on the command line, allowing you to run GSSync with command-line only options.
- **Override1-4:** These 49-character STRING fields are simply appended to the command line and can be used to override configuration options. If you have specific overrides that you need to specify, you can place them all in one block, or spread them out over multiple blocks. Splitting the overrides can make it easier to modify your schedules *en masse* using SQL UPDATE statements.
- **Override5:** This 249-character STRING field is also appended to the command line and can be used to override configuration options that do not fit in the shorter fields above.
- **SyncStartTimeReference:** This special TIMESTAMP field is used for PDC replication and is passed to the /ST command line option. The scheduler may automatically update this field upon job completion, as discussed in the next section.
- **SyncEndTimeReference:** This special TIMESTAMP field is used for PDC replication and is passed to the /ET command line option. The scheduler may automatically update this field upon job completion, as discussed in the next section.

The Log Table

The Log table holds the results of each completed job within the scheduler database. The table definition is as follows:

```
CREATE TABLE Log (
  ID                      IDENTITY,
  JobID                   INTEGER NOT NULL,
  JobName                  VARCHAR(32) NOT NULL,
  Status                   INTEGER,
  StartTime                TIMESTAMP,
```



```

        EndTime                TIMESTAMP,
        CommandLine            LONGVARCHAR);
CREATE INDEX IDTime on Log(JobID, StartTime);
CREATE INDEX NameTime on Log(JobName, StartTime);
CREATE INDEX StatusName on Log(Status, JobName);

```

The fields include:

- **ID:** A unique identity field for the Log records.
- **JobID:** The unique ID of the job from the Jobs table.
- **JobName:** The name of the job at the time it was launched. (Because names can change over time, this provides a secondary way to identify each job.)
- **Status:** The Status Code returned from GSSync from this job.
- **StartTime:** The time that the job was started.
- **EndTime:** The time that the job was completed.
- **CommandLine:** The arguments sent to GSSync from the scheduler. This is useful to understand which options were used, or to troubleshoot or further understand the scheduler's handling of various options.

Internal Data Definitions

In the process of running, the GSSyncScheduler script also leverages internal data values that you will need to understand. One of these critical fields is:

- **LoopStartTime:** The scheduler will scan through all of the records in the Jobs table periodically to see if there is any work to be done. Each time this loop runs, it first initiates an internal value called *LoopStartTime*, which stays consistent for all jobs executed during that loop. This value is then used to ensure that synchronization times are consistent, even as jobs are run sequentially. (Without this, jobs later in the schedule may have additional changes that are missing from earlier jobs.)

Creating Jobs

Creating jobs for the scheduler is actually pretty easy, as they are defined solely by the columns within the Jobs table. Because of this, you can manually create jobs through the Control Center user interface, or you can create them via SQL statements.

When you insert a new row to the Jobs table through the Control Center, you will see a number of default values pre-populated for you:

Column Name	Value	Type
ID	0	IDENTITY
Name		VARCHAR
Priority	0	INTEGER
LastRunStatus		INTEGER
LastRunStartTime	SYSDATETIME()	TIMESTAMP
LastRunEndTime	SYSDATETIME()	TIMESTAMP
NextStartTime	SYSDATETIME()	TIMESTAMP
RunInterval		INTEGER
ScheduleValidFrom	CURTIME()	TIME
ScheduleValidUntil	CURTIME()	TIME
LogLevel		INTEGER
LogFilePath		VARCHAR
ConfigFilePath		VARCHAR
Override1		VARCHAR
Override2		VARCHAR
Override3		VARCHAR
Override4		VARCHAR
Override5		VARCHAR
SyncStartTimeReference	SYSDATETIME()	TIMESTAMP
SyncEndTimeReference	SYSDATETIME()	TIMESTAMP

The default value for the *ID* column (0) will assign the next IDENTITY value to this job, so keep 0 for this value. Be sure that the *Name* column remains unique, as you cannot have two jobs with the same name (to avoid confusion). You can keep the default value for *Priority* while you are configuring the job itself, and then set it to some other value when you are ready to enable the job. The Control Center fills in default values for the TIME and TIMESTAMP fields, but you don't want these, so be sure to blank these fields out as you go to avoid problems.

Once you have defined the entire job as needed, click the “Add” button to add it to the database. You can add multiple jobs from this screen, and then click Close when done.

When creating a new job through SQL, you need to ONLY indicate the fields that you need to initialize, and all others will start with a NULL value. At minimum, you only need to supply the Name column, though such a job likely won't do much (unless you have a configuration file called GSSYNC.CFG file already configured for it), so it is more common to include both a *Name* and a *ConfigFilePath* for any new job, like this:

```
INSERT INTO Jobs (Name, ConfigFilePath) VALUES ('Newname', 'GSSyncJob.CFG');
```

Again, the default value for the *Priority* field will be 0 for any new job, allowing you to go back and confirm the job configuration prior to actually running it. If you are creating a complete Job record via SQL and are ready to schedule it immediately, be sure to set the *Priority* field to a non-zero value as well.

Creating a Run-Once Job

A **Run-once Job** is one that you enter into the scheduler and set up to run one time and then automatically disable itself afterwards. This can be helpful when you need to re-generate seed data with a full export, but you want to retain the job configuration for historical purposes or to re-use in the future.

To create a run-once job, simply create the job normally as per the steps above, but then set the *RunInterval* field to NULL. When the scheduler fires this job and attempts to compute the next start time, it will see that it is not scheduled to run again and instead set the *Priority* field to 0 (Disabled). You can easily enable the job again (by setting *Priority* to 1) to run it again.

Creating a Repeating Job

A **Repeating Job** is one that the scheduler will repeat at specified time intervals. The interval is specified in Minutes and is indicated by the *RunInterval* field.

To create a repeating job, simply create the job normally as per the steps above, but then set the *RunInterval* field to the number of minutes between runs. When the scheduler fires this job and attempts to compute the next start time, it will add the specified number of minutes to the current *LoopStartTime* and set the *NextStartTime* to that computed value. Note that setting the *RunInterval* to 0 will set the *NextStartTime* to the *LoopStartTime*, so that it will run on EVERY loop through the scheduler.

Defining the Job Schedule

The job schedule is defined by two TIME columns in the Jobs table called *ScheduleValidFrom* and *ScheduleValidUntil*. These time fields indicate the time period during the day within which a job can be started.

To allow the job to run at all times during the day, leave either of these fields NULL. If both times are set, then the job will only be considered between these times. Note that the scheduler does NOT wrap around midnight. If you set the *ScheduleValidFrom* to be AFTER the *ScheduleValidUntil*, the job will be ignored and a warning message posted at runtime. If you need to wrap a schedule around midnight, then you should schedule TWO jobs, one from the start time through 23:59, and the other from 00:00 through the ending time. (This can be done easily with the *CloneJob()* procedure.)

Specifying Synchronization Jobs

When synchronizing data using either PDC or Archive Log metadata, GSSync supports the use of special starting and ending timestamps, so that you can replicate database changes only made within a specific time interval. This is critical for environments where having data out of sync can result in major issues.

One such example of this is a simple invoicing system where two tables are used, *InvoiceHeader* and *InvoiceLine*, and you need to replicate ALL changes made within each table. If you replicate the *InvoiceLine* table first and THEN replicate *InvoiceHeader* later, it is possible that the target database will have records in *InvoiceHeader* that were created after the *InvoiceLine* job already completed – and therefore you will see invoices with no line items on them in the target database (at least until the next sync interval). If you swap the order of the tables, then you may end up with *InvoiceLine* records with no *InvoiceHeader* table – perhaps just as bad (or even worse).

To avoid these scenarios, GSSync supports the use of a specific cutoff time for the start and end of the synchronization window, specified by the /ST and /ET options on the command line. The GSSyncScheduler script understands how these values should be maintained and automatically manages them for you in the Jobs database within the two fields *SyncStartTimeReference* and *SyncEndTimeReference*.

Ignoring Synchronization Start/End Times

If you are not utilizing PDC or Archive Log metadata for synchronization, then you do not need to manage these times. Leave both values set at NULL to ignore the starting and ending time references.

Defining Run-Once Synchronization Jobs with Start/End Times

It is possible to schedule a **Run-once Synchronization Job** with specific start and end times. This can be helpful if a specific job was somehow missed or if you just want to see a list of all records that last changed during a specific time period.

To set this up, create your job normally with a *RunInterval* of NULL to force it to run only one time. Then, specify the *SyncStartTimeReference* and *SyncEndTimeReference* fields as the times you wish to supply to the /ST and /ET options. The scheduler will then run the job the next time through the scanning loop and then disable it immediately afterwards, without making any changes to the time reference fields. This will allow you to verify your results and even run the process again – useful for tracking down replication issues.

Spawning a New Current Synchronization Job

A **New Current Synchronization Job** is one that attempts to replicate all data changes from the last synchronization point to the current time.

To set up such a job, you create the job normally with *RunInterval* set to a valid value (i.e. non-NULL). You then specify the *SyncStartTimeReference* value that you want to use, and leave the *SyncEndTimeReference* set to NULL. The scheduler will understand that you are starting a new sync job and will replicate all data from the indicated *SyncStartTimeReference* to the *LoopStartTime*. (In other words, the /ST option is set to the *SyncStartTimeReference*, and /ET is set to *LoopStartTime*.) This will replicate all data changes from the indicated start time through the current loop start time. You can even specify a very old time (like 1980-01-01) for the starting time to effect a replication of all records in the data set.

When the process completes, the scheduler will then set both *SyncStartTimeReference* and *SyncEndTimeReference* to the *LoopStartTime*. This will change the job to a Current Synchronization Job for the next run.

Continuing a Current Synchronization Job

A **Current Synchronization Job** is defined by having the *SyncStartTimeReference* value being the same as the *SyncEndTimeReference* value. The scheduler will use the start time as the /ST

command line option, and the LoopStartTime as the /ET command line option, effectively replicating all changes from the last replication time to the current time.

When the process completes, the scheduler again sets both times to the LoopStartTime, so that it can repeat, effectively replicating the most recent changes on each scheduled run.

Spawning a New Lag Synchronization Job

In some environments, replicating the most recent data is not always a good thing. In these cases, a **Lag Synchronization Job** may be preferred. In a lag synchronization job, the data is not replicated from the last sync time to the current time, but rather from the *previous* sync time to the *last* sync time. The net effect is that the data is always one cycle behind. This job type can be useful to help ensure referential integrity, for example having customer data replicated normally, but then having customer invoices lagging one cycle behind. This can ensure that the customer record is pushed to the target database before any invoices for that customer, regardless of the priority order of the two tables.

To create a New Lag Synchronization Job, create the job normally as usual. Then specify the *SyncStartTimeReference* value as NULL, and set the *SyncEndTimeReference* set to some time in the immediate past. (This could be the current time when the job is created.) When the job is first executed, all data from the beginning of the database through the provided end time will be replicated.

When the GSSync process completes, the scheduler will update the *SyncStartTimeReference* value to the current value for *SyncEndTimeReference*, and then set the *SyncEndTimeReference* value to the *LoopStartTime*. This job will then always be lagging one cycle behind the other jobs.

Continuing a Lag Synchronization Job

If you do not want to replicate ALL data from the beginning of time when starting a Lag job, it is ALSO possible to manually define a **Lag Synchronization Job** instead. If you read the above section carefully, you'll see that a lag job is maintained when there are two **different** values for the *SyncStartTimeReference* and *SyncEndTimeReference* fields. (This is contrasted with a Current Synchronization Job which uses the SAME time for these two values.)

When the scheduler sees the times set to two different values, it will run with the /ST and /ET command line options set accordingly, and then shift *SyncEndTimeReference* to the *SyncStartTimeReference* column and set the *SyncEndTimeReference* to the *LoopStartTime* so that the next time the scheduler fires the job, it is again lagging one cycle behind.

Managing Jobs

Managing the Job queue is fairly simple, as it can be done solely within the Control Center inside the GSSyncDB database.

There are two ways to put a job on hold: Set the *Priority* to 0 (Disabled), or set an invalid schedule (*ScheduleValidFrom* > *ScheduleValidUntil*). You can use either option, as you see fit. You can also do so programmatically with a simple SQL statement like this one:

```
UPDATE Jobs SET Priority = 0 WHERE Name = 'job name';
```

Be sure to remember the setting that you changed so that you can change it back to release the job from hold. (If you are not using all of the Override fields, you can consider moving the original value to one of them, but again, remember to move it back later.)

To delete a job permanently that you will no longer require, simply delete the entire record from the Control Center, or issue a SQL statement like this one:

```
DELETE FROM Jobs WHERE Name = 'job name';
```

The GSSyncDB database also comes with a stored procedure called *CloneJob()* that allows you to easily clone an existing job. This can be done with this SQL statement:

```
CALL CloneJob('oldjobname', 'newjobname');
```

This stored procedure creates a new job record that is identical to the old job record (but with the new name) and immediately sets the *Priority* field to 0 (Disabled). This is done to prevent the job from running before you have properly edited the job configuration. Once you have completely set up the new job, you can then set the *Priority* field appropriately to have it seen by the scheduler.

Note that *CloneJob()* simply ignores errors. If you attempt to clone a job that does not exist, or if you attempt to create a new job with the same name of an existing job, it will complete normally without telling you what went wrong. So, if your clone operations are not working, verify that you are supplying the correct names.

Running the GSSync Scheduler After the First Time

Subsequent runs of the GSSync Scheduler script won't need to do any of the extra setup work, but instead will run the configured schedule. However, there are a few operational parameters that control how the scheduler works. You may elect to use the default values that we chose, or you may wish to configure your own values as needed. Each of these values are named parameters to the PowerShell script and are invoked using the same syntax as the ones listed in the above section. Here is a complete list of the operational parameters:

- **-DBServerName:** Indicates the name of the server holding the scheduler configuration database. (Default: 'localhost')
- **-DBName:** Indicates the name of the scheduler configuration database. (Default: 'GSSyncDB')

- **-LogLevel:** Like GSSync, the schedule supports 5 logging levels to control what data is displayed in the scheduler window. (Default: 3)
- **-SleepSeconds:** Each time the GSSync Scheduler completes a full pass through the schedule, it will sleep for this period of time. This not only reduces workload on the server, but also allows you to reconfigure the scheduler between loops. (Default: 60 seconds)
- **-PathToGSSync:** If the GSSync.EXE file does not reside in the current directory, then you can specify a path to GSSync with this option. This also allows you to test a new version of GSSync without making too many changes. (Default: “.\GSSync.EXE”)

Using GSSync Scheduler Off-Label

While the GSSyncScheduler script has been designed specifically to run GSSync, it is *technically* possible to use it to schedule other types of jobs instead, similar to Task Scheduler. This may be quite useful if you need to schedule a number of SQL jobs within your database environment and then spawn those requests through either the Actian tools like PVDDL or DEU, or through our own SQLExec tool.

To set this up, you would first create an alternate scheduler configuration database for your SQL statements, such as “SQLExecDB”. Then, when you spawn the scheduler script, specify the **-DBName** option to indicate the alternative database, and use the **-PathToGSSync** value to provide your alternate command (i.e. “.\SQLExec.EXE”). Now, when the schedule runs, it will automatically call SQLExec along with any command line options provided in the *Override* fields. Be sure to keep all other fields set to NULL to avoid passing any invalid command line parameters to the external tool. You could even use this same process to call a batch file with specific parameters, which can then execute an entire range of functions based on those parameters, though perhaps Task Manager is a better solution.

Reconfiguring GSSync Scheduler While Sleeping

Once the scheduler starts up, it will start working through the job schedule as given. If there is no active schedule, it will go back to sleep and display “Sleeping” on the screen. During this period of time, several keystrokes are available to you that can use to reconfigure the scheduler’s operation. These keystrokes include:

- <Enter>: Stop waiting and immediate start another schedule scan.
- <Esc>: Stop waiting and exit the GSSync Scheduler script.
- Digits 1-5 (on main keyboard, not keypad): Change the GSSync Scheduler log level to that number. This allows you to quickly enable debug-level logging (5) for a single schedule run and then go back to normal logging afterwards.

Note: If you have configured the Scheduler to not sleep (i.e. “-SleepSeconds 0”), or if you are running GSSyncScheduler.ps1 from a PowerShell ISE window, then these options will not be available to you. Instead, use Ctrl-C instead of <Esc> to exit the script at any time.

Managing the Job Log

The Log table can be easily queried with the Control center or any SQL tool to extract information about scheduled jobs.

A stored procedure named *CleanLog()* has also been provided to offer cleanup options. This procedure accepts a single parameter -- the number of days back to clear. Here is an example cleaning out all records older than 30 days.

```
CALL CleanLog(30);
```

The default, if you provide no parameter value, is to delete all Log records older than 90 days.

Appendix A: Sample XML Configuration File

A sample XML configuration file is provided with GSSync. In case you modify or delete this file unexpectedly, here is a copy of the original file.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
    GSSync Configuration File
    Goldstar Software Inc.
    www.goldstarsoftware.com
-->
<GSSyncConfig version="2.08">

  <LogOptions>
    <!--
      This section contains the log file options.  First, we select the
      Log File Name. (The default is GSSYNC.LOG if not specified.)
      We then select whether to append to the existing file or create new.

      Second, we provide the log level, from the following list:
      Level 1 = Required Messages Only
      Level 2 = All Above + Errors
      Level 3 = All Above + Warnings (This is the default setting).
      Level 4 = All Above + Informational Messages
      Level 5 = All Above + Debug Messages
    -->
    <LogFileName>GSSYNC.LOG</LogFileName>
    <LogFileAppend>0</LogFileAppend>
    <LogLevel>3</LogLevel>
    <!--
      Five different message display languages are currently supported:
      0 = English
      1 = Spanish
      2 = Italian
      3 = French
      4 = German
    -->
    <DisplayLanguage>0</DisplayLanguage>
  </LogOptions>

  <SourceOptions>
    <!--
      This section contains the information about the source data file.
      Exports to BTRV and UNF do not require access to the data dictionary,
      but all other options need the DDF's or XML definition to proceed.
    -->

    <!--
      Use ONE of these options to locate the data dictionary.
      Named Databases are only supported on PSQlV9 and higher.
      The DatabaseDirectory option is valid for any version, and works
      with DDFs of type V1 (PSQlV9 and earlier) and V2 (PSQlV10+)
    -->
    <!-- Define the Named Database to Connect To (Valid on PSQlV9+) -->
    <DatabaseName></DatabaseName>
    <!-- Define the Database Directory (Use "." for current directory) -->
    <DatabaseDirectory>.</DatabaseDirectory>

    <!--
      Once we have the DDF's, we need a suitable user name and password.
    -->
```

```

<!-- Define the Database User Name -->
<DatabaseUser>Master</DatabaseUser>
<!-- Define the Database Password -->
<DatabasePassword></DatabasePassword>

<!--
    We now need to specify the Btrieve file and owner name (if needed).
    If the DDF's were specified above, we can leave the BtrieveFile entry
    blank, which will use the SQL Table definition to locate the Btrieve
    file. If the DatabaseName is used above, a database URI will be
    constructed to create a proper path to the file.

    However, if your DDF's and data files are not in the same folder, then
    you may find it necessary to indicate the exact Btrieve file name and
    path in the BtrieveFile field.
-->
<!-- Define the Table Name from the DDF's to Access -->
<DatabaseTable>Person</DatabaseTable>
<!-- Define the Btrieve File Name (Pulled from Table Name if empty) -->
<BtrieveFile></BtrieveFile>
<!-- Define the Btrieve Owner Name (If needed) -->
<BtrieveOwner></BtrieveOwner>

<!--
    We're going to run the Btrieve file by a key, or we can run it by
    physical record order. The physical record order (-1) is the fastest
    for operations that do not need access to the System Data value.
    However, Using the System Data Key (125) can enable additional optimizations
    that can greatly improve extraction performance.
-->
<!-- Define the Btrieve Key Number (Default is -1, Use 125 for SysData
Optimization) -->
<BtrieveKey>-1</BtrieveKey>

<!--
    If we don't have DDF's, we can get the definition from an XML file.
-->
<!-- Define the XML File Definition Name (Only if no DDF Available) -->
<XMLFileDefinition></XMLFileDefinition>

<!--
    If our data file includes a variant definition, then we need to specify the
    variant record type.
-->
<!-- Determine if we need to enable Variant Record Processing, 0=No, 1=Yes -->
<VariantRecordEnabled>0</VariantRecordEnabled>
<!-- Specify the Byte Offset (0-relative) of the Variant Field -->
<VariantRecordFieldOffset>0</VariantRecordFieldOffset>
<!-- Indicate the Field Type of the Variant Field (Limited support for options
may be available; see documentation) -->
<VariantRecordFieldType>0</VariantRecordFieldType>
<!-- Indicate the Field Length of the Variant Field (Limited support for options
may be available; see documentation) -->
<VariantRecordFieldLength>0</VariantRecordFieldLength>
<!-- Specify the Variant Record field value that must match for this record to
be processed -->
<VariantRecordData>0</VariantRecordData>

</SourceOptions>

<MetadataOptions>

```

```

<!--
    Metadata is data about your data that is used to control the replication
    process, and to replicate only those records that need to be replicated.

    Define the Metadata Type from the following list:
        NONE: No Metadata, Full Export Only
        PDC: Use PDC Metadata (From a valid DataExchange configuration)
        ARCHIVE: Use PSQL Archive Log Metadata (See documentation for caveats)
        GSCREATE: Create a new GSSync Metadata file and Perform a Full Export
        GSSYNC: Use the GSSync Metadata Format with Record Position as the
primary key
        GSSYNCSYS: Use the GSSync Metadata Format with System Data value as the
primary key

    Based on your selection, you may need to provide more data below.
-->
<MetadataType>GSCREATE</MetadataType>

<!-- Define the PDC Metadata File Name: Required for PDC Metadata -->
<PDCMetadataFile>PDCPERSON.MKD</PDCMetadataFile>

<!-- Define the Archive Log File Name: Required for ARCHIVE Metadata -->
<ArchiveLogFile></ArchiveLogFile>

<!-- Define the GSSync Format Metadata File Name: Required for GSSYNC Metadata -
->
<GSMetadataFile>PERSON.GS</GSMetadataFile>
</MetadataOptions>

<ExportOptions>
<!--
    Define the Export Type from the following list:
        BTRV: Export by Updating an identical Btrieve File
        CSV: Export to Comma-Delimited File
        FORK: Export to a SQL Statement Script AND to an ODBC Database
        JSON: Export to JSON Output Format
        NONE: Do not export (Useful for building metadata only)
        ODBC: Export via ODBC Linkage
        QUEUE: Export via binary Queue File for importing (with GSSyncImport)
        SQL: Export to a SQL Statement Script
        UNF: Export to Btrieve UNF Format
        VWL: Export to Vector VWLOAD File
        XML: Export to XML Output Format
-->
<ExportType>CSV</ExportType>

<!-- Do you want to prepend or append the date/time to the export file name?
0: No
1: Yes, Date Only in Format MMDD
2: Yes, Date Only in Format YYYYMMDD
3: Yes, Date/Time in Format YYYYMMDD_HHMM
4: Yes, Date/Time in Format YYYYMMDD_HHMMSS
-->
<PrependDatetimeToFileName>0</PrependDatetimeToFileName>
<AppendDatetimeToFileName>0</AppendDatetimeToFileName>

<!-- Define the CSV File Export File Name: Required for CSV Export Type -->
<ExportCSVFile>PERSON.CSV</ExportCSVFile>
<ExportCSVHeaderRow>1</CSVHeaderRow>
<!--
    Let's define the CSV Export Format Here

```

```

        Set to 0 to quote none of the fields (Raw Comma File)
        Set to 1 to quote all fields (Comma-Quote File))
        Set to 2 to quote only String fields
        Add 10 to eliminate quotes from the header row field names
-->
<ExportCSVFormat>1</ExportCSVFormat>

<!-- Define the UNF File Export File Name: Required for UNV Export Type -->
<ExportUNFFile>PERSON.UNF</ExportUNFFile>

<!-- Define the XML File Export File Name: Required for XML Export Type -->
<ExportXMLFile>PERSON.XML</ExportXMLFile>
<!-- Define the XML Export Format from the following list:
    0: RAW Format (TableName / Row / Field=value)
    1: Field List Format: (TableName / FieldName /Value)
-->
<ExportXMLFormat>1</ExportXMLFormat>

<!-- Define the JSON File Export File Name: Required for JSON Export Type -->
<ExportJSONFile>PERSON.JSON</ExportJSONFile>
<!-- Define the JSON Export Format from the following list:
    0: Rewadable Format (CR/LF After each field, plus indentation)
    1: Compact Format: (Minimal Whitespace)
-->
<ExportJSONFormat>0</ExportJSONFormat>

<!-- Define the Vector VWLOAD Export File Name: Required for VWL Export Type -->
<ExportVWLFile>PERSON.VWL</ExportVWLFile>
<!-- Leave 0 to write output to ONE data VWLOAD Export File, change to 1 for 10
files -->
<ExportVWLFileParallel>0</ExportVWLFileParallel>

<!-- Define the Btrieve File Export File Name: Required for BTRV Export Type -->
<ExportBTRVFile>PERSON.BTV</ExportBTRVFile>

<!-- Define the Queue File Export File Name: Required for QUEUE Export Type -->
<ExportQueueFile>PERSON.BTV</ExportQueueFile>

<!--
    For all of the above items, we need some additional data:
        Should we export to one or multiple files?
        Where should we write Deleted record information?
        Should we ignore all deleted records (for use in a data warehouse)?
        Should we output the Delta Flag?
        Should we export the System Data value?
        Should we export the Record Position value?
        Should we export the Last Changed timestamp value?
        Should we export the Export timestamp value?
-->
<!-- Leave 0 to write output to ONE Export File, or split output to up to 100
files -->
<ExportParallelOutput>0</ExportParallelOutput>

<!-- Define the Deleted Data File Export File Name -->
<ExportDeletedDataFile>del_person.csv</ExportDeletedDataFile>

<!-- Set this to 1 to export Delta Flag (I/U/D) -->
<ExportDeltaFlagFormat>0</ExportDeltaFlagFormat>
<ExportDeltaFlagFormatName>DeltaFlag</ExportDeltaFlagFormatName>

<!-- Set to 1 to Export Record Position Data Field -->

```

```

<ExportPositionDataField>1</ExportPositionDataField>
<ExportPositionDataFieldName>BtrievePosition</ExportPositionDataFieldName>

<!--
    Note: Use this chart for the next FOUR TimeStamp Fields:
    Set to 0 to NOT export the field.
    Set to 1 to export time in PSQL Septasecond Format.
    Set to 2 to export time in Unix Format (from epoch 01/01/1970).
    Set to 3 to export time in ODBC Timestamp format with UTC Time.
    Set to 4 to export time in ODBC Timestamp format with Local Time.
    Set to 5 to export time in US String format with UTC Time.
    Set to 6 to export time in US String format with Local Time.
    Set to 7 to export time in European String format with UTC Time.
    Set to 8 to export time in European String format with Local Time.
    Set to 9 to export time in Oracle TO_TIMESTAMP format with UTC Time.
    Set to 10 to export time in Oracle TO_TIMESTAMP format with Local Time.
    Set to 11 to export time in Vector TIMESTAMP format with UTC Time.
    Set to 12 to export time in Vector TIMESTAMP format with Local Time.
    Set to 13 to export time in Big Numeric format with UTC Time.
    Set to 14 to export time in Big Numeric format with Local Time.
    Set to 15 to export time in YearFirst String format with UTC Time.
    Set to 16 to export time in YearFirst String format with Local Time.
    Set to 17 to export time in Quoted YearFirst String format with UTC Time.
    Set to 18 to export time in Quoted YearFirst String format with Local Time.
-->
<!-- Controls Export of ther Last Change Timestamp Value -->
<ExportLastChangeTimeStamp>0</ExportLastChangeTimeStamp>

<ExportLastChangeTimeStampName>LastUpdatedTimestamp</ExportLastChangeTimeStampName>

<!-- Controls Export of the TimeStamp the Export Is Running -->
<ExportExportTimeStamp>0</ExportExportTimeStamp>
<ExportExportTimeStampName>ExportTimestamp</ExportExportTimeStampName>

<!-- Controls Export of the System Data Field NOTE: VALUES OVER 1 ARE *ONLY*
VALID ON V13 FILES WITH V2 System Data ENABLED -->
<ExportSystemDataField>1</ExportSystemDataField>
<ExportSystemDataFieldName>SystemData</ExportSystemDataFieldName>

<!-- Controls Export of the Update System Data Field NOTE: REQUIRES v13 FILES
WITH V2 System Data ENABLED -->
<ExportUpdateSystemDataField>0</ExportUpdateSystemDataField>

<ExportUpdateSystemDataFieldName>UpdateSystemData</ExportUpdateSystemDataFieldName>

<!-- A comma-delimited list of fields to NOT export, thus eliminating extraneous
fields. Leave empty string if none. -->
<IgnoreFieldList></IgnoreFieldList>

<!--
    If you are exporting to a SQL script file, set the target filename.
-->
<!-- Define the SQLFile Export File Name -->
<ExportSQLFile>PERSON.SQL</ExportSQLFile>

<!--
    If you are exporting to an ODBC DSN, then we have a few options.

    The first option is to include the ODBC Connection String directly
    inside this configuration file. To use this, set the
    ExportODBCConnectionString option.

```

The second option is to read the ODBC connection string from a text file. This can be best if you are replicating numerous files and would like the ability to modify the ODBC Connection string for all exports simultaneously. To use this, create a text file with the ODBC connect string as the only line, then provide the filename in the ExportODBCConnectionStringFile below.

```
-->
<!-- Define the ODBC Connection String -->
<ExportODBCConnectionString>DSN=DEMODATA</ExportODBCConnectionString>
<!-- Define the ODBC Connection String File-->
<ExportODBCConnectionStringFile></ExportODBCConnectionStringFile>

<!--
    The following settings are used to build the SQL statements for the
    SQL Script and ODBC Export options. Please see the docs for more
    information on setting these options.
-->
<!-- Define the ODBC Query to Use for Exporting INSERT statements -->
<ExportODBCInsertQuery>INSERT INTO {{TNAM}} VALUES ( {{SKEY}}, {{****}}
);</ExportODBCInsertQuery>
<!-- Define the ODBC Query to Use for Exporting UPDATE statements -->
<ExportODBCUpdateQuery>DELETE FROM {{TNAM}} WHERE SystemData = {{SKEY}}; INSERT
INTO {{TNAM}} VALUES ( {{SKEY}}, {{****}} );</ExportODBCUpdateQuery>
<!-- Define the ODBC Query to Use for Exporting DELETE statements -->
<ExportODBCDeleteQuery>DELETE FROM {{TNAM}} WHERE SystemData =
{{SKEY}};</ExportODBCDeleteQuery>
<!-- Strip out semicolons from the query before submitting to ODBC -->
<ExportODBCStripSemicolons>0</ExportODBCStripSemicolons>

<!--
    The formatting options help us to control the export of
    potentially troublesome elements, like bad control characters in
    data, date formats, etc.
-->
<!--
    Let's define the Boolean Export Format Here
    Set to 0 to use 1 or 0
    Set to 1 to use TRUE or FALSE
    Set to 2 to use YES or NO
-->
<BooleanExportFormat>1</BooleanExportFormat>
<!--
    Let's define the Date Export Format Here
    Set to 0 to use the format mm/dd/yyyy
    Set to 1 to use the format dd.mm.yyyy
    Set to 2 to use the format yyyy-mm-dd
    Set to 3 to use the format yyyymmdd
    Set to 10 to use the SQL format {d 'yyyy-mm-dd'}
    Set to 11 to use the SQL format TO_DATE('yyyy-mm-dd','YYYY-MM-DD') (for
Oracle)
    Set to 12 to use the SQL format DATE 'yyyy-mm-dd' (for Vector)
-->
<DateExportFormat>11</DateExportFormat>
<!--
    Let's define the Time Export Format Here
    Set to 0 to use the format hh:mm:ss.ff
    Set to 1 to use the format hhmmss
    Set to 2 to use the format 'hh:mm:ss.ff'
    Set to 10 to use the SQL format {t 'hh:mm:ss.ff'}
    Set to 11 to use the SQL format TO_DATE('hh:mm:ss','HH24:MI:SS') (for
Oracle -- Loss of Precision Expected)
```

```

        Set to 12 to use the SQL format TIME 'hh:mm:ss.ff' (for Vector)
-->
<TimeExportFormat>0</TimeExportFormat>
<!--
    Let's define the Timestamp Export Format Here
    Set to 0 to use the format yyyy-mm-dd hh:mm:ss.ff
    Set to 1 to use the format yyyymmddhhmmss
    Set to 2 to use the format 'yyyy-mm-dd hh:mm:ss.ff'
    Set to 10 to use the SQL format {ts 'yyyy-mm-dd hh:mm:ss.ff'}
    Set to 11 to use the SQL format TO_TIMESTAMP('yyyy-mm-dd hh:mm:ss','YYYY-
MM-DD HH24:MI:SS') (for Oracle -- Loss of Precision Expected)
    Set to 12 to use the SQL format TIMESTAMP 'yyyy-mm-dd hh:mm:ss.ff' (for
Vector)
-->
<TimestampExportFormat>10</TimestampExportFormat>

<!--
    Character Field Filtering Options:
        1: Change CR and LF to space characters.
        2: Change Null byte (0) to space character.
        4: Change control characters (1-31) to space character.
        8: Clear high bit on every character.
        16: Force all text characters to UPPERCASE.
        32: Eliminate Trailing Blanks from CHAR and VARCHAR Fields
        64: Change Vertical Bar to space character (for VWL Exports).
        128: Change Double Quote character to Space.
        256: Change Single Quote character to Space.
        512: Change Backslash character to Space.
    (Add multiple option values together, if needed.)
-->
<CharacterFilterOptions>2</CharacterFilterOptions>

<!--
    NUMERIC Field Filtering Options:
        0: No filtering, output bad bytes as bad data.
        1: Change bad digits to 0
        2: If bad digits, change value to 0
        3: If bad digits, change value to NULL
        4: If bad digits, change value to 9's
        5: If bad digits, change value to -9's

-->
<NumericFilterOptions>0</NumericFilterOptions>
<!-- Allows the retention of leading zeroes in NUMERIC/DECIMAL values -->
<RetainLeadingZeroes>0</RetainLeadingZeroes>

<!--
    Date Field Filtering Options:
        0: Output all dates as-is.
        1: Output any invalid dates as NULL.
        2: Output any invalid dates as 01/01/1901.
        3: Output any invalid dates as 01/01/1980.
        +10: 00/00/0000 dates are considered invalid dates.
    (Add multiple option values together, if needed.)
-->
<DateFieldFilterOptions>11</DateFieldFilterOptions>

<!--
    The following items are used to handle export record filtering.
-->
<!--
    This expression will be evaluated on the source data to

```

```
        determine if a record should be exported or not
    -->
    <RecordFilterExpression>ID &ge; 200000000</RecordFilterExpression>
    <!-- Allow the user to dynamically activate the record filter -->
    <RecordFilterActive>0</RecordFilterActive>

</ExportOptions>
</GSSyncConfig>
```

This is provided for reference purposes only and may change in subsequent releases.

Appendix B: Version History

This section described the most recent changes to GSSync so that you can easily see what changed between your current release and any new release. These changes are listed in reverse chronological order, with the latest update at the top of the list.

Version 2.10

- Fixed export of floating point values (and integer/scale) to avoid “.###” or “-.###” format, on which JSON chokes.
- Support the optional removal of leading spaces and zeroes from DECIMAL and NUMERIC fields with a new /LZ switch and a RetainLeadingZeroes option in the configuration file. For JSON output, this setting is ALWAYS enabled.
- If the user provides a backslash in the table name (i.e. using UNC paths), then double the backslashes to make them work with JSON.

Version 2.09

- Fixed issue where Deleted records were not recorded properly to the XML or JSON export formats.

Version 2.08

- Removed the /VP option and replaced it with the /PO option, and Parallel Output now works with most export formats.

Version 2.07

- Addressed memory allocation issues under certain circumstances.
- Enabled String Filtering to handle double quote, single quote, and backslash characters to help address flawed import routines in other environments.

Version 2.06

- Addressed issue with JSON data and the en dash and em dash characters.

Version 2.05

- Added ability to configure an Ignore Field List to provide a comma-delimited list of fields to skip on the export side, in order to reduce overhead and “junk” fields.
- Handled escaping *all* invalid characters in JSON-formatted output.

Version 2.04

- Changed default name for XML output fields to match other formats: “DeltaType” to “DeltaFlag”, “Position” to “BtrievePosition”.
- Added ability to configure the output field names via new options in the CFG file. Note: To use this, be sure to update your existing CFG files to the new 2.04 format.
- Updated DECIMAL output formatting to eliminate leading zeros from output data.
- Added support for a new output method – JSON formatted data.

Version 2.03

- Added header row to CSV file for Delete records. (Note that if you previously attempted to detect a 0-byte deleted record file, but have header rows enabled in your main CSV output file, this will now create a 1-row deleted file.)

Version 2.02

- Fixed issue with timestamp conversion function not handling 2022-01-01 correctly.

Version 2.01

- Fixed issue with BTRVEX function call in some circumstances.
- Fixed issue of not terminating the export loop at EOF.

Version 2.00

- Changed core to use BTRVEX function call where needed, now supporting v13 files over 4 billion records and greater than 256GB in size (requires v13.30+).
- **IMPORTANT: GSSync now requires W3BTRV7.DLL, which implies PSQLv7 and above only. (Users of Btrieve 6.x should stay on v1.78 or older.)**
- When used on v13 files with UpdateSysData enabled, both SystemData and UpdateSysData can be exported as true timestamp values.
- UpdateSysData (System Data v2) can be exported in all modes (when available) and imported properly from a Btrieve Queue File. Make sure that the target database has v2 system data enabled, if needed.
- New v2 GSMetadata and v2 Queue File formats are required to support larger record positions and the UpdateSysData. **IMPORTANT: Old metadata tables and queue files are not compatible and must be re-generated. Stay on v1.78 or older until you have an opportunity to rebuild all of these files.**
- Implemented multi-lingual message tables, supporting English, Spanish, Italian, French, and German. (Field names are NOT translated.) Contact Goldstar Software if you need a specific translation added or if you see inaccurate translations.
- XML definitions were truncating table/column names to 20 characters; now supporting 128 characters to be compatible with V2 metadata.

Version 1.78

- Added new data type option for Binary, VarBinary, and HexBytes.

Version 1.77

- Added ability to specify the ODBC connection string from the command line using the /OO option.

Version 1.76

- Fixed bug in extending negative decimal values.

Version 1.75

- If bad NUMERIC fields are too short to apply their decimal point, silently extend the field with leading 0's and apply the decimal in the correct spot anyway.
- Added line at end of log to report the filtered record count.

Version 1.74

- Fix up any output XML row or column tags to ensure they meet XML rules.
- Optimized XML Output code to improve performance by caching field names.
- Added support for additional XML DDL fields, such as Position (instead of Offset), Length, Decimal, and Type, as well as support for numerous XML DDF Type definitions used by the RecordEditor utility (by Bruce Martin).

- Echo Config File, Btrieve File, and/or TableName command line options to output screen so you can distinguish between multiple, concurrent processes.
- When writing NUMERIC fields, invalid bytes are now changed to 0 to avoid writing potential garbage to the output stream. (No warning is currently posted.)

Version 1.73

- Addressed a bug in handling records over 57000 bytes.
- Optimized several core functions to improve performance.

Version 1.72

- Added the ability to handle INTEGER and UNSIGNED fields with implied decimal points (i.e. Scale). This is non-standard and can ONLY be done within an XML data definition, as the PSQL/Zen database chokes on any non-zero value there.

Version 1.69

- Fixed issues with passing some error codes back to the main process.

Version 1.68

- Added support for 32-byte owner names, as well as the AutoTimestamp data type.

Version 1.67

- Added the /PD switch to add a timestamp before the export filename.

Version 1.66

- Added the ability to write data to a binary queue file.
- Changed the /AD function to add an underscore before the timestamp, and changed this to use local time, not UTC time.

Version 1.65

- Updated the logging capabilities with the following improvements:
 - Added a new option to append to an existing log file.
 - Now initializing log file only AFTER entire configuration has been read.

Version 1.64

- Added two new replacement strings to the SQL statement parser, {{BNAM}} for the Btrieve file name, and {{DNAM}} for the database name.

Version 1.63

- Specifying an invalid field number (such as 1099 in a table with only 50 fields) was crashing the application. GSSync now detects this situation and posts an Error to the log instead of crashing. The record will still be exported, but the bad field will be exported as a NULL.
- Invalid field names, specifically those starting with a number, were confusing the SQL statement parser. The code now can differentiate between fields like {{1099Name}} and the numeric field number {{1099}}.

Version 1.62

- A new Ending Timestamp (/ET) switch was added to allow synchronization of records only up to a specific time. This can be used to provide discrete replication windows (such as one hour at a time) which can help to keep data in sync across relationships.

Version 1.61

- Added a new CSV Export (/CE) Format option to allow for exported data to have all fields quoted, no fields quoted, or only strings quoted.

Version 1.60

- A new algorithm for GSSync metadata was included, after it was found to be about 20% faster than the older code and reduced CPU load. **Note that this version of the GSSync Metadata is NOT directly compatible with the previous format, so if you are using GSSync metadata and are updating from pre-1.6 to post-1.6 code, you must regenerate your metadata tables.**

Version 1.55

- A new Replacement String {{TNAM}} was added that refers to the source database table, making it possible to use generic SQL script strings for many tables.
- The Bad Date filter now considers dates with years over 9999 as invalid dates.
- Added more output options for the metadata timestamp fields LastChangeTimestamp and ExportTimestamp.
- Cleaned up Time and Timestamp displays for fractional seconds.
- Added new output format options for Date and Time fields to support Vector.
- Added output option switch /SE for TimestampExportFormat to indicate export format for DateTime and Timestamp fields, including support for Vector.
- Added output option switch /BE for BooleanExportFormat to indicate export format for Boolean fields, including 1/0, TRUE/FALSE, and YES/NO options.
- Added new Vector output format, useful for sending data directly to VWLOAD, with the ExportType=VWL or the /OV switch. Also added ability to export to a single VWL file or to split the output into 10 VWL files, so that VWLOAD can run parallel imports.
- Added string filtering capabilities to VARCHAR, LSTRING, and NOTE fields, as well as a new option to filter vertical bar characters, needed for VWL exports.

Appendix C: Known Limitations

GSSync, like all tools, has some limitations. When we identify the more critical ones, we will note them here for reference. These limitations may or may not be fixed in a future release, so be sure to let us know if you run into one of them.

- Limits on Exported Fields: CLOB Fields can be exported, but only up to 128KB in size, and BLOB fields can be exported through 64KB. This is due to an internal buffer size for each field of 128KB. If you exceed this limit, the field will be replaced with the text “ERROR: CLOB OF LENGTH 1816990 TOO LONG FOR GSSYNC TO PARSE” and an error will also be logged to the log file.
- There is limited support for WSTRING and WZSTRING fields at this time, due mainly to lack of sample data and preferred processing directives. Contact us if you need support for these field types expanded.