University of Rostock

Department of Computer Science

Heterogeneous Data Replication

Peter Haase
born 31.05.1976 in Neubrandenburg
Tutors: Andreas Heuer, Holger Meyer

Preface

Today, more and more business applications are being designed to run in distributed computing environments. In these environments, data replication is a key to effectively distributing and sharing information. It is a fast and reliable way to disseminate and consolidate information to and from multiple locations. Furthermore, enterprise information systems become increasingly heterogeneous, with a growing number of diverse platforms and database systems. Data replication solutions need to address these requirements for heterogeneous support. This paper examines the major commercial replication designs. Based on IBM's DataPropagator solution, a new true heterogeneous architecture will be proposed, including a prototypical implementation.

How to read this paper

- **Section 1** gives an introduction to the main concepts of heterogeneous replication.
- **Section 2** examines a selection of four commercial replication products.
- **Section 3** explains the architecture of one replication product, IBM's DataJoiner / DataPropagator, in detail.
- **Section 4** analyses the inherent problems of the DataJoiner / DataPropagator architecture.
- **Section 5** establishes a list of requirements for a new architecture.
- **Section 6** lists considerations for the implementation of the new architecture.
- **Section 7** presents the prototype of the new architecture.
- **Section 8** compares the current with the new architecture and prototype.
- **Section 9** closes with a summary and an outlook to future work.

Contents

1 Introduction to Heterogeneous Data Replication

Replication is the process of maintaining a defined set of data in more than one location. It involves copying designated changes from one location (source) to another (target), and synchronizing the data in both locations.

Heterogeneity means the ability to replicate between unlike platforms. The differences might be in the hardware, operating systems, or DBMS. [1]
Heterogeneous support enables to consolidate data from different sources, to share data among disparate systems, and to assist in the evolutionary migration from old to new technologies.

The following section gives an overview of the major commercial replication designs and concepts.

1.1 Fragmentation and Partitioning

Fragments are logical portions of a table which are physically stored in one or several databases in the network. The mapping between tables and fragments logically defines a fragmentation schema. The allocation schema defines at which site(s) a fragment is located. A fragment that is redundantly stored at multiple sites is called replica. There are two classes of fragmentation, or partitioning:

- **Table Partitioning:** Subsets of a given table are defined using SQL predicates that refer to the columns in the table. This can be done by projection (vertically), selection (horizontally), or both (mixed).
- **Database Partitioning:** The tables in the database are partitioned on a common column using foreign key relationships.

1.2 Synchronicity

Data replicators can be classified into two broad classes: Synchronous and Asynchronous.

1.2.1 Synchronous Replication

With the synchronous approach, replication is performed within the source update transaction, usually using a two-phase-commit. The source transaction is not considered complete until the update has been successfully replicated to all targets.

[1] Differences in schemas (schema transformations) are not subject of this paper.

The main advantage of synchronous replication is that full synchronization at all times is guaranteed. Furthermore, the use of the two-phase-commit protocol eliminates the possibility of update collisions. If the performance impact on the source transaction is not acceptable, synchronous replication should be avoided. If, on the other hand, data consistency across all sites is critical, synchronous replication should be considered.

1.2.2 Asynchronous Replication

With an asynchronous replicator, the source update is independent of the replication process. The user's transaction is complete when the local update is complete. Usually, updates will be replicated only after the user's transaction commits the changes to the source database. [2] Replication may occur moments after the source transaction completes, or it may be scheduled for later execution.
The benefit of asynchronous replication is that it minimizes the impact on the user's transaction and increases the robustness of the replication process. In an event of a network failure, the replicator holds all updates in a persistent queue.
Asynchronous replication should be used when absolute data consistency across all sites is not critical.

1.3 Multi-Directional Replication

Multi-directional replication allows updates to databases at multiple sites. This implies the need for update conflict detection und resolution. Update conflicts occur when applications commit competing, potentially incompatible updates to two or more replica, and the existence of these competing updates cannot be detected until propagation of these updates to other replicas occurs. There are two general classes of update conflicts:

- **Intra-table** update conflicts that are detectable within the scope of a single table.
- **Inter-table** update conflicts that are not detectable within the scope of a single table, e.g. because foreign key relationships are violated.

Strategies for conflict resolution are for example:

- **Earliest/latest timestamp:** the update with the earliest/latest timestamp wins the conflict.
- **Designated reference table:** updates on the designated reference table always win the conflict.

[2] Some architectures allow to replicate uncommitted data, e.g. DB2 DataPropagator when bound UR (Uncommitted Read).

1.4 Serializablity and Transactional Integrity

The execution of transactions on replicas is called correct, if the effect of the transactions is the same as on a non-replicated database. This important property is called 1-copy-serializability [8].
A replicator that maintains the order of updates within a transaction, and transmits all source transaction updates as a single unit of work is said to provide "transactional integrity" or "transactional consistency". Transactional consistency is essential to assure referential integrity.
Remark: Transactional integrity only guarantees the consistency and atomicity properties of an ACID transaction. Isolation and durability can not be guaranteed in asynchronous update-anywhere replication environments.

1.5 Changed Data Capture Strategies

Any replication architecture will need to be able to capture changed data instead of replicating the entire set of data. This enables the replicator to perform differential refresh, opposed to full refresh. There are several approaches to capture changed data, but all major commercial replication products use either log based or trigger based changed data capture.

1.5.1 Log Based Capture

A log based replicator scans the logs maintained by the DBMS. It finds changes to data that is registered for replication. These changes are captured in some form and sent to the targets. Log based capture operates asynchronously, which implies a latency until the replicator becomes aware of the update. Although all "industrial strength" DBMSs maintain logs, log based capture is not always feasible. Some vendors do not publish their log structure. In a heterogeneous environment, the replicator would need a log reader for each source DBMS.

1.5.2 Trigger Based Capture

A trigger based replicator executes code invoked by database triggers. These triggers fire in the event of changes to data that is registered for replication. In theory, a trigger-based replicator could be synchronous, with the trigger updating the remote target databases. This approach, however, eliminates the benefits of asynchronous replication. Therefore replication triggers typically store changed data into some form of queue. Most DBMSs provide triggers, and although the trigger functionalities may vary, generally the same trigger design can be implemented for most DBMSs.

2 Existing Products for Heterogeneous Replication

A number of different vendors offer solutions for heterogeneous replication. Among them are almost all vendors of the well established DBMS. Besides those, there are other third-party providers of replication solutions. The following section investigates a selection of commercial replication solutions, with focus on their architecture and heterogeneous support.

2.1 IBM DB2 DataPropagator / DataJoiner

In conjunction with DataJoiner, DataPropagator forms a flexible solution for heterogeneous replication across platforms and DBMSs.

2.1.1 DataPropagator

DataPropagator is the central product of IBM's replication solution. The DataPropagator architecture is based on three main components: an administration interface, change-capture mechanisms, and the Apply program. These three components are independent of each other, they communicate through the use of replication control tables.

- **Administration Interface:** The DataJoiner Replication Administration tool (DJRA) is used to define the replication setup: Enable a database for replication, register tables as sources, define subscriptions.
- **Change Capture Mechanisms:** In the case of DB2 as a replication source, the Capture program captures changed data by reading the DB2 transaction log. In the case of a non-IBM replication source, capture triggers emulate the Capture functions. The captured changed data is stored in dedicated changed data tables.
- **Apply:** The Apply program reads the changed data from the changed data tables and applies them to the replication targets.

DataPropagator allows bi-directional replication ("Update Anywhere"), but only for DB2 to DB2 replication.

2.1.2 DataJoiner

DataJoiner is IBM's strategic gateway to enable transparent access to relational and non-relational, IBM and non-IBM databases. Heterogeneous data sources can be accessed as if they were DB2 databases.
DataJoiner is not only used in heterogeneous replication systems. It can also be

used for heterogeneous distributed data access. DataJoiner enables users and applications to perform distributed SQL queries over tables that are located in separate, multi-vendor databases.

The DataJoiner and DataPropagator architecture will be explained in detail in chapter 3.
For further reading on DataPropagator, [11] and [10] be recommended.

2.2 Sybase DataMovement

Sybase DataMovement is a complete suite of products that provides bi-directional, heterogeneous replication. The two central components are the Replication Server and Replication Agents.
The Replication Agents are used to capture changes at heterogeneous data sources. Currently, there are a log based Replication agent for DB2 and a trigger based Replication Agent for Informix and Oracle available. The Replication Agents passes the captured changes on to Replication Server which then distributes those changes. The changed data is extracted and distributed as entire transactions to maintain the transactional integrity of all replicated data. The replication process with Replication Server is asynchronous and transparent to applications.
As a replication target Sybase additionally supports any ODBC and DRDA targets.

An example replication scenario might look like this:

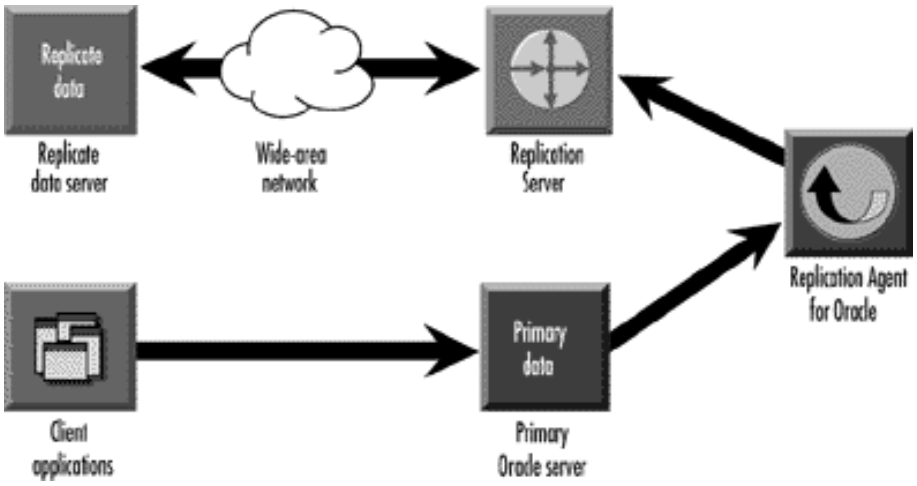

Figure 1: Replication scenario with Sybase DataMovement

Further information on Sybase Replication Server can be found in [5]

2.3 MS-SQL Server 7 Replication

The MS-SQL Server 7 replication model consists of Publishers (replication source servers), Subscribers (replication target servers) and Distributors, articles (an entire table or part of a table registered for replication) and publications (grouping of articles). SQL server replication allows three types of replication:

- **Snapshot replication:** takes a snapshot of the published data in the database at one point in time. Instead of monitoring and copying data changes, Subscribers are updated by a total refresh of the data set.
- **Transactional replication**: monitors changes to the publishing server at the transaction level. Only committed transactions are sent to subscribing servers, retaining their order and thus guaranteeing transactional consistency. With transactional replication, changes are only allowed at the publishing site (no bi-directional replication).
- **Merge replication**: allows multi-directional replication. Publishers and Subscribers can connect periodically to merge their results. Conflicts are resolved automatically by pre-assigned priorities. Merge replication uses triggers on the published tables (articles) to capture changed data.

The replication process is managed by four different replication agents, which all run under the control of the MS-SQL Server agent and can be administered from using SQL Server Enterprise Manager, as shown in figure 2. The replication agents are:

- **Snapshot Agent**: prepares the schema and initial data files of published tables, stores the snapshot on the Distributor, and records information about the synchronization status in the distribution database. Each publication has its own Snapshot Agent that runs on the Distributor and connects to the Publisher.
- **Log Reader Agent**: moves transactions marked for replication from the transaction log of the Publisher to the distribution database. Each database published using transactional replication has its own Log Reader Agent that runs on the Distributor and connects to the Publisher.
- **Distribution Agent**: moves the transactions and snapshots held in distribution database tables to Subscribers. Depending on the type of subscription, the Distribution Agent runs either on the Distributor and connects to the Subscriber, or it runs on the Subscriber (Pull Subscription).
- **Merge Agent**: moves and reconciles incremental data changes that occurred after the initial snapshot (for Merge Replication only).

Heterogeneous interoperability is one of the weaknesses of MS-SQL Server. Out of the box, it only supports replication to heterogeneous DBMSs through either the ODBC interface or OLE DB. Replication from heterogeneous DBMSs is addressed in the way that the replication programming interfaces are exposed and documented, allowing software developers to create their own replication agents and integrate them with SQL Server.

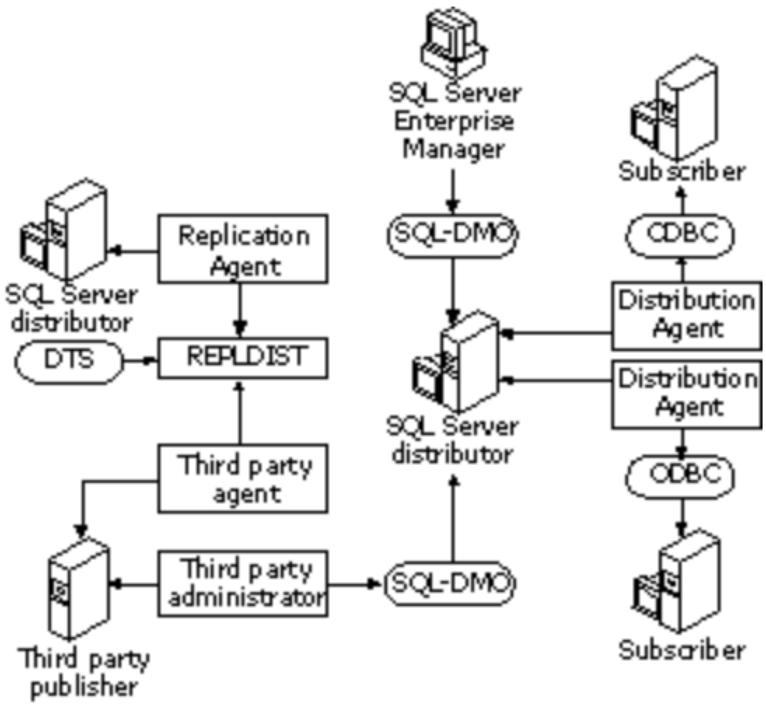

Figure 2: Heterogeneous Replication with MS SQL Server 7

SQL Server Replication was designed to be easy to setup and manage. It provides a Replication Wizard and a Replication Monitor.

2.4 Amdahl InfoReplicator

Amdahl InfoReplicator is a replication solution especially for heterogeneous environments. It provides asynchronous change data replication with full data integrity and transaction consistency, as well as bi-directional capabilities.
InfoReplicator consists of the following components:

- Change Capture component: specific to each source DBMS, place data updates into persistent queues

- Dispatcher component: readies the changes by transaction and target
- Distributor component: operates across the network via client/server middleware, performs data type conversions, applies data to targets

Supported DBMSs are: Oracle, Sybase, Informix, DB2, MS-SQL Server
Supported OS are: Windows NT, UNIX, MVS

2.5 Comparison

All the examined replication products have a similar asynchronous architecture: There is a component that captures changed data at the source, and a component that applies the changed data to the target. All of them claim to retain transactional consistency.
All architectures require a specific capture component for each supported source DBMSs, which usually has to be purchased separately. Only IBM's DataPropagator supports all its data sources "out of the box" (i.e., the support is included with the product). All solutions are limited to a fixed number of supported platforms and DBMSs. None of them is easily extensible.
IBMs combination of DataJoiner and DataPropagator is by far the most complex solution, it supports a great variety of DBMSs on almost all major platforms and offers many different options in the replication setup.

3 The Current DB2 DataPropagator Architecture in Detail

3.1 DB2 Data Replication Concepts

3.1.1 Replication Sources

A replication source is a user table or a view from which data will be copied. Before data can be replicated, a replication source must be defined. It describes the information that the change-capture mechanism will use. When registering a replication source, the following parameters need to be specified: subset of columns to replicate, before/after or before image, differential or full refresh, level of conflict detection for update anywhere.

3.1.2 Subscriptions

Before data can be replicated from the replication source, it must be associated with the target to which the changes are to be replicated. This information is defined using subscription sets and subscription members, and stored in various replication control tables.

3.1.3 Logical Servers

All the replication components reside on a logical server. In this sense, the term server actually refers to a database. There are three types of logical servers:

- **Source server:** contains the change-capture mechanism, the replication source tables and control tables used by the Change-capture component and the Apply program.
- **Target server:** contains the target tables.
- **Control server:** contains control tables used by the Apply program.

3.2 DB2 Data Replication Components

3.2.1 Control Tables

The replication components use control tables to communicate with each other and to manage replication tasks. The most important ones are

at the source server:

- *Register Table:* contains information about replication sources, such as the names of the replication source tables, their attributes, their staging table names, and synchronization information.
- *Register Synch Table:* a trigger on this table updates the register table, indicating how much changed data has been captured.
- *Prune Control Table:* coordinates the pruning of the changed data tables. For each subscription member there is an entry in this table indicating its synchronization status. Data that has been propagated to all subscription members, can be pruned from the changed data table.
- *Changed Data Tables:* records all changes made to a replication source.

at the control server:

- *Subscription Set Table:* lists all of the subscription sets with their replication status defined at the control server and identifies the source and target server pairs that are processed as a group.
- *Subscription Member Table:* contains information about the individual source and target table pairs defined for a subscription set.
- *Subscription Columns Table:* contains information about the columns of the subscription set members.
- *Subscription Statements Table:* contains user-defined SQL statements or stored procedure calls that will be executed before or after a subscription set processing cycle.
- *Subscription Events Table:* contains information about the events that trigger a subscription set processing cycle.
- *Apply Trail Table:* contains audit trail information for the Apply program.

3.2.2 Change Capture Mechanisms

The DB2 data replication solution offers these mechanisms for capturing changed data:

- **Capture Program** for DB2 source tables: reads the DB2 transaction log and records the changed data in CD (changed data) and the UOW (Unit of Work table).
- **Capture Triggers** for non-IBM data sources: capture committed changes to CCD (consistent change data) tables. They are fired when a particular database event (INSERT, UPDATE, DELETE) occurs. The Capture triggers are generated by the DataJoiner Replication Administration tool (DJRA).

3.2.3 Apply

The Apply program reads data from the source and changed data tables and propagates it to the replication targets. Each Apply program is associated with one control server, which contains the control tables that contain the definitions for the subscription sets.

3.2.4 DJRA

DJRA - the DataJoiner Replication Administration Tool is the administrative interface to define the replication setup. It can be used to create the replication control tables, register replication sources, define subscriptions, and for several further adinistration tasks. DJRA produces SQL-DDL statements that can be manually edited and executed. These statements also include the definition of the Capture triggers for non-IBM data sources.

3.3 The Replication Process

The replication process is a parallel process of capturing changed data and applying it to the target. The changed data capture in the heterogeneous case with capture triggers is fairly simple, whereas most of the work is done by the Apply program. Its work consists of five distributed steps at the control, source and target server:

3.3.1 Apply Cycle Overview

1. *Control Server:* Look for work - check the subscription control tables
2. *Source Server:* Pick up recent changed data to be applied
3. *Target Server:* Apply the foreign changed data to the target table
4. *Control Server:* Update subscription status
5. *Source Server:* Report subscription status in the pruning control table

The following figure gives an overview of the entire replication process with its involved components as well as the data and control flow.
The numeration in the figure corresponds to the servers as follows: source server: 1-9, control server: 10-15, target server: 16.

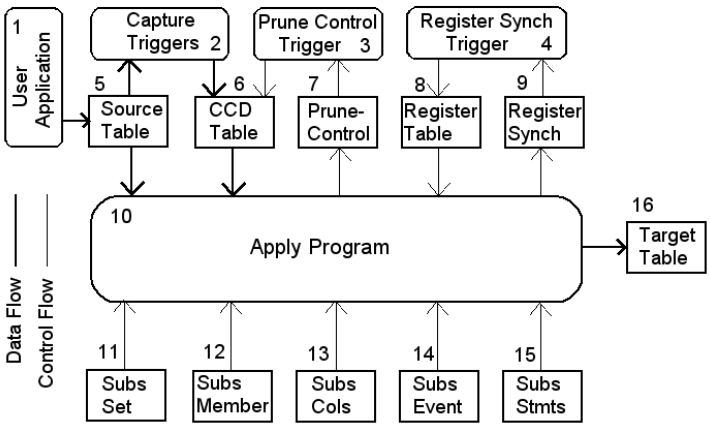

Figure 3: Data flow in a heterogeneous replication scenario with DataPropagator

3.3.2 Trigger-based Communication

DJRA, working through DataJoiner, creates Capture Triggers (2) on non-IBM source tables when they are defined as replication sources. Three types of triggers are created on the source tables: DELETE, UPDATE and INSERT. Also, UPDATE triggers are created on the pruning control table (7) and the register synchronization table (9). The Apply program uses these control tables to detect what needs to be copied to the target database. The following process describes how the Capture triggers and Apply communicate in a typical replication scenario to ensure data integrity:

1. Whenever a DELETE, UPDATE, or INSERT operation (1) occurs at the source table (5) that is defined as a replication source, a Capture trigger (2) records the change in the consistent-change-data (CCD) table (6).

2. When the Apply program (10) is started, it updates the register synchronization table (9). The Update trigger on this table (4) updates the register table (8) to record how much committed changed data has been captured.

3. The Apply program gets the source table information from the register table (8).

4. Before the Apply program can copy differential changes to the target, it synchronizes the target with the replication source by copying all the data from the replication source (5) to the target (16). This action is called a full refresh.

5. The Apply program updates the pruning control table (7) to synchronize the capture of the related changes in the CCD table.

6. The Apply program reads the CCD table (3) using DataJoiner nicknames, and applies the changes to the target table (16).

7. The Apply program updates the pruning control (7) table with a value that indicates the point to which it copied changes to the target database.

8. The UPDATE trigger on the pruning control table (3) checks all of the CCD tables (6) that are at the source server and deletes those entries that were replicated.

4 Problems with the Current DataJoiner/ DataPropagator Architecture

4.1 Limited Heterogeneity

The current architecture makes use of DataJoiner as an interface to access non-IBM replication sources and targets. DataJoiner uses a wrapper architecture that allows to access those non-IBM databases as if they were DB2 (Version 2) databases. DataJoiner has rich capabilities for distributed data access, including optimization for queries across heterogeneous databases. On the other hand, the Apply program uses none of those capabilities. In fact, the current architecture requires a separate database for each heterogeneous target database. Depending on DataJoiner implies limited heterogeneity in terms of both supported platforms and DBMSs: Although DataPropagator is available for a number of different platforms and DBMSs, there are some that are not addressed. The amount of resources necessary to support a new platform or DBMS is tremendous.
In today's world of heterogeneous systems it is desirable to develop platform and operating system independent solutions.

4.2 Data Type Mappings

With the current DataJoiner/DataPropagator architecture, all replication source and target databases are treated as if they were DB2 Version 2 databases.
Even though DB2 pretty much complies with the SQL standards, most other DBMSs do not. They may only support a subset of the SQL data types, or they may have different semantics. Moreover, all data types introduced after DB2 V2 are not supported by DataJoiner, e.g. most of the new SQL99 data types like LOBs, Bigint, or distinct types.
The datatype mappings for all supported DBMSs are hardcoded into DJRA. That implies considerable code changes for the support of a new DBMS. Even if a new version of a DBMS supports new data types, the DJRA code needs to be changed. To assure backwards compatibility the DJRA code needs to be version dependent.

4.3 Changed Data Capture, Trigger Architectures

The current architecture relies on triggers as a changed data capture mechanism. Although almost all major DBMSs support triggers, their functionality differs significantly. Some important differences are:

- Trigger Granularity: Some DBMS support row level, some statement level triggers, others support both. The triggers have to be designed for each support seperately. Statement level triggers are a problem in the way that changed data has to be captured row by row.
- Support of variables: Some DBMS do not support the use of variables in a trigger body (e.g. Teradata) or have other restrictions of equal consequences. This makes the trigger design challenging or potentially impossible.
- Trigger Definition Syntax: There is no common syntax for trigger definition among the DBMSs.

As a result of the differences, each supported DBMS has its own specific trigger design, some of them even make use of proprietary DBMS features like procedures or sequence generators (Oracle). Again, the trigger definitions are hardcoded in the DJRA backend. The support of an additional replication source requires significant code changes.

5 Requirements for an Improved Heterogeneous Replication Architecture

Based on the problems inherent in the current heterogeneous replication solution that were characterized in the previous section, this section will determine the requirements for a new heterogeneous replication architecture and implementation.

5.1 True Heterogeneity

The new replication architecture should provide maximum heterogeneous support, with respect to platforms as well as DBMSs. Ideally, the program should not even be aware of the platform it runs on or which DBMS it talks to.

5.2 Dynamic Data Type Mappings

When replicating data across different DBMSs, the data type mappings from the source table to the target table should be done by using metadata information of the target database about the supported data types. If a specific data type is not supported by the target DBMSs, the data type should be mapped to the closest data type that is supported, if possible without restricting precision.

5.3 Extensibility

The replication solution should be easily extensible in terms of supported DBMSs. None of the components that need to be DBMS dependent (e.g. changed data capture mechanisms) should be hardcoded. If support for an additional DBMS is to be added, this should be feasible by simply registering it in some way without changing any code.

In addition to these requirements that directly result from the problems listed in the previous section, two additional requirements are essential for the acceptance of a new architecture:

5.4 Performance

The new architecture should reach a similar or better performance as the current one using DataJoiner. Performance in this sense mainly refers to the replication latency and data throughput of the Apply program. Especially the performance for pull configurations (Apply program is running on the target server) will be of importance: True pull configurations are not possible with a DataJoiner architecture, because the Apply program always connects to the DataJoiner database. This limitation has been criticized by some customers; it would be a great advantage if the new architecture would perform better in this specific scenario.

5.5 Interoperability and Compatibility

Every part of the new replication architecture should work interchangeably with existing parts of the current product. For example, a replication scenario that was setup with the old DJRA, running with DPropR Capture should work with the new Apply component of the new architecture.
Especially the format of the replication control tables and changed data tables should be untouched. If the product should ever be used commercially, it should support establishing the IBM replication control tables as a standard for data replication.

6 Implementation Considerations for a New Architecture

6.1 Heterogeneneity with Java and JDBC

Java is the first choice for developing platform independent software. Java programs, written once, can run everywhere on virtually any platform on top of a Java

Virtual Machine. In combination with JDBC as the interface to access heterogeneous databases, a Java based implementation is the perfect solution for heterogeneous replication in terms of platforms and supported DBMSs. Also, Java provides several built in functionalities that could be useful for data replication, for example its Thread capability, which would allow quasi-parallel replication to multiple targets, or processing multiple subscription sets at the same time.

6.1.1 The JDBC Interface

The JDBC API is a set of abstract Java interfaces that allow an application programmer to open connections to particular databases, execute SQL statements, and process the results.
The most important interfaces are:

- java.sql.DriverManager: for loading drivers, creation of database connections
- java.sql.Connection: representation of a database connection
- java.sql.Statement: container for execution of SQL statements
- java.sql.ResultSet: allows access to the result of a query

6.1.2 javax.sql

javax.sql is an extension to the standard JDBC API that addresses two aspects that are important for the replication architecture: Heterogeneity and performance.

- **JNDI for naming databases:** The Java Naming and Directory Interface (JNDI) can be used in addition to the JDBC driver manager to manage data sources and connections. When an application uses JNDI, it specifies a logical name that identifies a particular database instance and JDBC driver for accessing that database. This has the advantage of making the application code independent of a particular JDBC driver and JDBC URL.
- **Connection Pooling:** The JDBC 2.0 Standard Extension API specifies hooks that allow connection pooling to be implemented on top of the JDBC driver layer. This allows for a single connection cache that spans the different JDBC drivers that may be in use. Since creating and destroying database connections is expensive, connection pooling is important for achieving good performance, especially for server applications.
- **Distributed transaction support:** Support for distributed transactions allows a JDBC driver to support the standard 2-phase commit protocol used by

the Java Transaction API (JTA). JDBC driver support for distributed transactions allows developers to write programs that are transactional across multiple DBMS servers.
Theoretically this could be a basis for a synchronous replication architecture, but not in the scope of this prototype.

- **Rowsets:** As its name implies, a rowset encapsulates a set of rows. A rowset may or may not maintain an open database connection. When a rowset is disconnected from its data source, updates performed on the rowset are propagated to the underlying database using an optimistic concurrency control algorithm.
  For the replication architecture Rowsets could be used to send entire sets of changed data rows across the network, instead of retrieving data row by row.

6.2 JDBC Data Type Mappings

JDBC defines a set of generic SQL type identifiers in java.sql.Types, which correspond to SQL standard data types. Programming in JDBC, there is generally no need to concern about the exact SQL type name used by the target database. The one major place where one needs to use SQL type names is in the CREATE TABLE statement. For this case, there are two choices: One choice is the restriction to the use of widely accepted SQL typenames (like Integer, Numeric,...), but this contradicts the idea of a generic replication architecture. The other choice is making use of the DBMS's Metadata to specify the correct data type name. The following figure demonstrates how this can be done for a mapping from an arbitrary source table to an arbitrary target table.

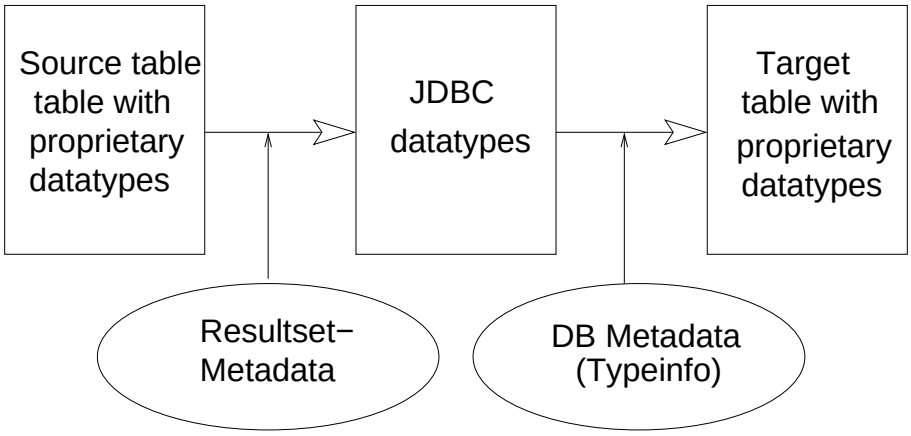

Figure 4: Java data type mappings using Metadata information

The ResultSet Metadata of a query against the source table (e.g. SELECT * FROM

source) can be used to map the proprietary data types of the source database to the generic JDBC data types. The Metadata of a Resultset can be obtained using the method ResultSet.getMetaData().
Using the TypeInfo information of Database Metadata of the target database, it is possible to map the JDBC types to actually supported data types of the target database. These typenames can be used in the CREATE TABLE statement for the definition of the target table.
Unfortunately, the information provided by the database metadata is not always correct or sufficient. In some cases, the metadata information says, the database does not support a specific data type even if it does. In other cases it is desirable to map to a similar data type if a specific data type is not supported. The following figure shows examples for reasonable alternative mappings.
Remark: If these alternatve mappings get more complicated one has to ensure to avoid cycles.

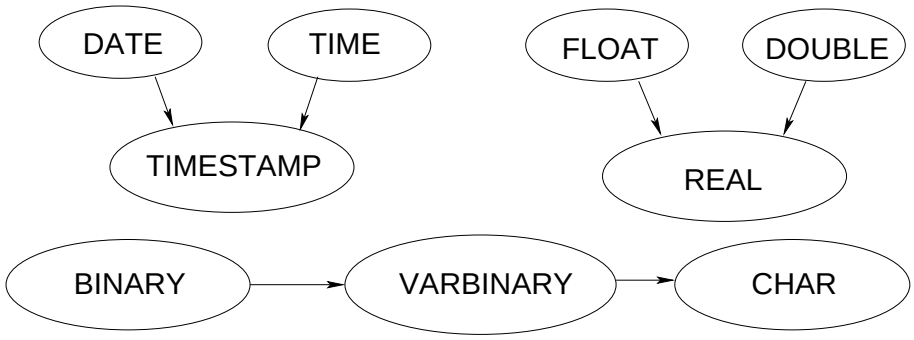

Figure 5: Alternative Type Mappings for JDBC

JDBC 2.0 introduces support for SQL3 data types, such as User Defined Types (Distinct Types, Structured Types, Reference Types) and LOBs (CLOBs and BLOBs). There are also new interfaces that allow a custumized mapping of UDTs. The DatabaseMetaData and ResultSetMetaData has been extended for getting Meta-Data about the new types.

6.3 An Extensible Approach for Changed Data Capture

There are several feasible approaches to capture changed data for heterogeneous replication. The focus will be on an extensible approach for trigger definitions using XSL-T technology. Alternative ways for changed data capture will also be explained.

6.3.1 Trigger Based

The trigger definitions need to be specific for each DBMS for the reasons explained in 4.3. To assure maximum heterogeneity and extensibilty, it is useful to separate the specific trigger definition component from the actual replication tools. This can be done using XSL technology as shown in Figure 6. The administration tool gen-

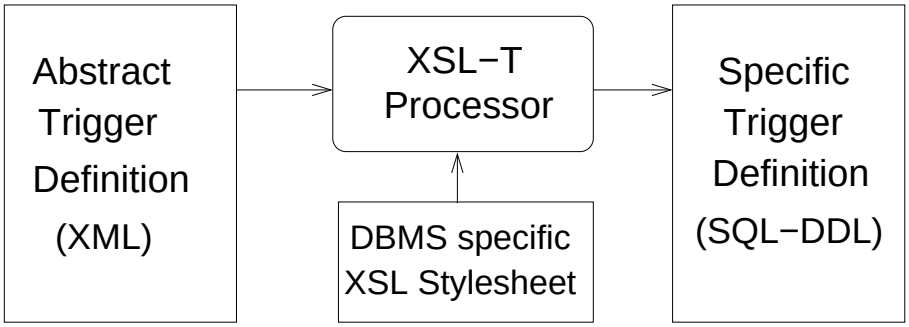

Figure 6: Trigger definitions using XSL technology

erates an abstract definition of what the change capture triggers should look like and stores it into an XML file.
An XSL-T processor transforms this abstract definition into specific SQL-DDL statements using an XSL stylesheet. The SQL-DDL output statements will be parsed again and executed against the source database. Which XSL stylesheet should be used can be determined using the Connection Metadata. Support for a new replication source DBMS is as simple as providing a new stylesheet.
The same technique can be used in all other places where DBMS dependent statements have to be executed, e.g. in the definition of the replication control tables.
An example of an abstract trigger definition, an XSL-T stylesheet and the transformed trigger definition in SQL can be found in Appendix A.

Alternative ways to perform changed data capture, e.g. in the case that the trigger based approach is not applicable, are:

6.3.2 Log Read

For the DBMSs that have published their log format, it is possible to write Capture components that read the log maintained by the DBMSs. One such program, DPropR's Capture is already available. For Sybase and MS-SQL Server there exists a program that reads the logs of these DBMSs.

6.3.3 Updating Application Captures Changed Data

It is also possible to let the application, which updates registered source tables, record the changed data directly into the changed data tables. Of course this contricts the idea of application independence, or transparence. It should therefore only be used if there is no alternative (e.g., no trigger support).

6.3.4 No Capture of Changed Data

Sometimes it may not even be necessary to capture changed data, for example if the data is highly fluctuating, or the amout of replicated data is very small. In these cases replication could be done by a regular full refesh.

6.4 Performance Aspects

An implication of the current changed data replication approach is that data is propagated row by row. This basically means: Read a row from the source (more precisely: fetch from a ResultSet), perform an update at the target, read a row, perform an update, etc. Especially with the high communication overhead of the JDBC interface this can be very costly.
An obvious improvement would be to fetch multiple rows of changed data at a time (Block Fetch) and also perform multiple updates at a time (Batch Updates). Batch Updates are a standard feature of the JDBC 2.0 interface. All major JDBC drivers have implemented Batch Update support. Unfortunately, block fetch is not a standard feature of the JDBC interface. [3] This limits the performance improvements of the Batch Updates.
Performance can also be improved by choosing the right JDBC driver. There are four categories of drivers:

1. JDBC-ODBC bridge plus ODBC driver: The Java Software bridge product provides JDBC access via ODBC drivers.

[3] Some JDBC drivers do support block fetches as an additional feature.

2. Native-API partly-Java driver: This kind of driver converts JDBC calls into calls on the client API for Oracle, Sybase, Informix, IBM DB2, or other DBMSs. Like the bridge driver, this style of driver requires that some operating system-specific binary code be loaded on each client machine.

3. JDBC-Net pure Java driver: This driver translates JDBC calls into a DBMS-independent net protocol, which is then translated to a DBMS protocol by a server. This net server middleware is able to connect its pure Java clients to many different databases. The specific protocol used depends on the vendor.

4. Native-protocol pure Java driver: This kind of driver converts JDBC calls directly into the network protocol used by DBMSs. This allows a direct call from the client machine to the DBMS server.

Category 1 and 2 drivers require a local database client on the client machine. Categories 3 and 4 offer all the advantages of Java technology, including automatic installation, and should therefore be used for a true heterogeneous solution. On the other hand, Category 2 drivers are typically much more performant. For example, when the changed data is fetched, the ResultSet is held within the local client, which significantly reduces the communication overhead.

7 Overview of the Prototype

The proposed architecture has been prototypically implemented. Similar to the current DataJoiner/DataPropagator architecture, it consists of a graphical administration interface and an Apply program. Both programs use the JDBC interface to access the heterogeneous databases. For ease of use, the connection configurations are stored in an XML file servers.xml, which, again, is used by the administration tool and the Apply program. The format of the changed data tables and control tables is the same as in the DataPropagator solution, to guarantee compatibility and interoperability (Section 5.5).

7.1 Replication Administration Tool

The Replication Administration Tool (RAT) acts as graphical interface to both setup as well as represent the entire replication scenario in a hierarchy of Servers, Replication Sources, Subscription Sets and Subscription Members (Figure 8 shows a screenshot). All DBMSs specific components (like the Capture triggers, specific control tables) are stored in XML stylesheets.
Moreover, the Apply process can be started from the administration tool.
Figure 7 shows an overview of the implemented RAT architecture.

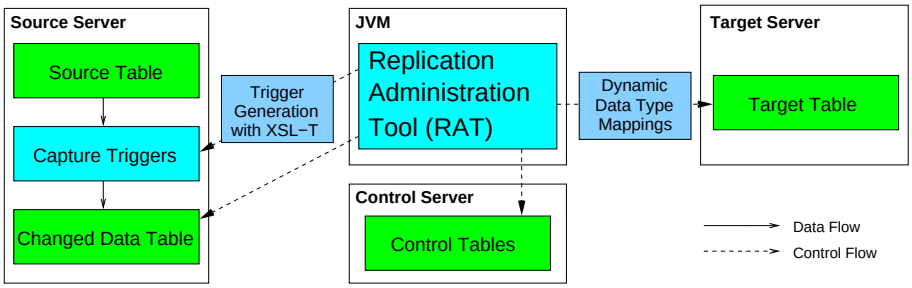

Figure 7: Replication Administration Tool

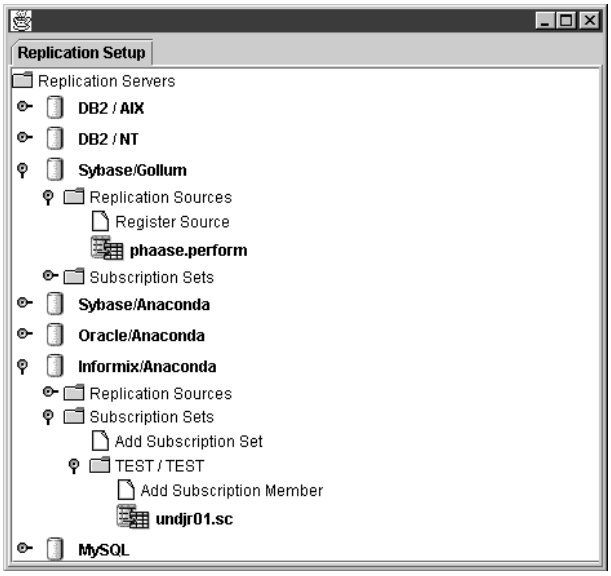

Figure 8: Screenshot Replication Administration Tool

7.2 Java Apply

The Java Apply program is very similar to the DataPropagator Apply. It is entirely platform independent and can be run on any Java Virtual Machine. The program can be started either from the command line or from the administration tool, with the Apply Qualifier and the Control Server as a parameter. The Apply makes use of Java's thread functionality: Multiple subscription sets can be run in parallel as threads.
The current prototype only supports differential refresh from CCD (consistent changed data) tables and supports only user copies as a target (see future work).

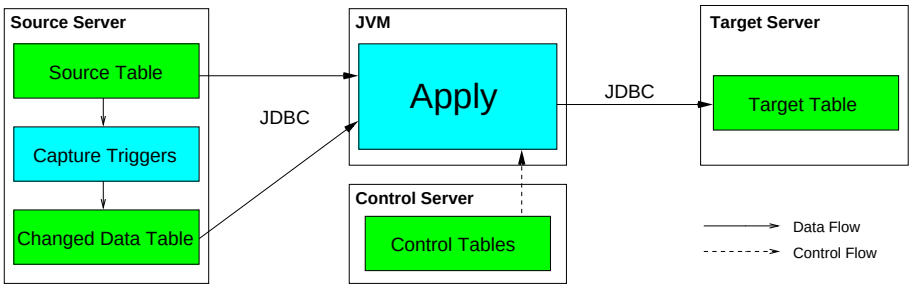

Figure 9: The Java Apply Program

8 Comparison of the Existing DataJoiner/ DataPropagator and the Proposed Architecture

8.1 Heterogeneity

The DataJoiner/DataPropagator solution supports a fixed set of DBMSs, which includes all major DBMSs like Sybase, Oracle, Informix. This set is not extensible for the user, whereas the proposed architecture uses the JDBC interface, which is a common, widely used and accepted interface, with drivers available for all relational DBMSs.
DataJoiner and DataPropagator are available on a great variety of operating systems. Theoretically, the Java solution is entirely platform independent, but this independence is limited by the availibilty of Java Virtual Machines (JVM) on specific platforms. The prototype has been tested on AIX, Windows NT/98 with the following DBMSs: DB2, Oracle, Sybase, Informix, MySQL in various combinations.

8.2 Functionality

The prototype provides the same functionalities as the current DataPropagator solution with some restrictions (e.g. range of target table types). These restrictions are not limitations set by the architecture, but simply not implemented in the scope of the prototype. Additional functionalities include the support of SQL3 data types (not fully implemented and tested up to this point).

8.3 Usability

Whereas DataJoiner is very difficult to install and configure, the JDBC interface only requires the presence of a JDBC driver (bundled with the replication package). No further local client installations are necessary. This is a real improvement in terms of usability.
Also, the new Replication Administration Tool, RAT, is very easy to use, since its functionality is limited to what actually is supported, whereas the DJRA tool strikes the user with a lot of functionality that is not even supported for hetereogeneous replication. Moreover, DJRA's lack of a graphical representation of the replication setup makes it hard to comprehend a replication scenario easily.

8.4 Performance

As mentioned in 6.5, propagating the replicated data row by row implies a lot of communication overhead. This is true for both using DataJoiner as well as the JDBC interface. Especially, when the Apply program is run on a different machine than the database server, the overhead may become very costly. This is usually the case when DataJoiner is used, since Apply has to run in the DataJoiner instance. A true Pull configuration is not possible with non-DB2 targets.

The following tables show results of a performance comparison of the DataPropagator Apply with DataJoiner and the Java Apply with JDBC. Shown are the times for a full refresh of a simple table with 10000 rows (with Integer, Real and Varchar columns). For the Java Apply, a JDBC class 2 driver was used for DB2, a class 4 driver for Oracle.

Table 1: Replication in a Pull Configuration, Oracle to DB2

Source	Target	Apply Type	Time	Batch Updates
Oracle/NT	DB2/NT	DPropR Apply	9s	no
Oracle/NT	DB2/NT	Java Apply	30s	no
Oracle/NT	DB2/NT	Java Apply	9s	yes (100 Statements)

Using no batch updates, the DPropR Apply needs only a third of the time to perform a full refresh as the Java Apply does, because it does not have the communication overhead of the JDBC interface when updating the target row by row. Using Batch Updates, the Java Apply can reduce the communication overhead considerably, achieving the same performance as the DPropR Apply.

Table 2: Replication in a Push Configuration, DB2 to Oracle

Source	Target	Apply Type	Time	Batch Updates
DB2/NT	Oracle/NT	DPropR Apply	70s	no
DB2/NT	Oracle/NT	Java Apply	75s	no
DB2/NT	Oracle/NT	Java Apply	11s	yes (100 Statements)

In the Push configuration, the DPropR Apply performs a lot worse, since the communication path through DataJoiner to the Oracle target is a lot longer. The Java Apply shows about the same performance using row by row updates. But the performance gain by the batch updates is even greater, outperforming the DPropR Apply by almost an order of magnitude.

In summary, using an appropiate JDBC driver, batch updates, and minimizing communication paths, the Java based solution can achieve the same or in certain scenarios much better performance than the DataJoiner/DataPropagator solution.

9 Summary and Future Work

With the fully functioning prototype as a proof of concept, it has been shown that it is feasible to do cross-platform, cross-DBMS replication by using Java and JDBC as an interface to access heterogeneous databases. In terms of performance, the Java/JDBC solution may be at least as performant as the DataJoiner/DataPropagator solution. Using new technologies like XML and XSL-T to encapsulate DBMS dependent code, extensibility can be improved, development work and code maintenance can be reduced.

9.1 Not Addressed Problems

Some problems were not addressed in the scope of this paper and the prototype. These include:

- Multi-directional replication and a conflict resolution algorithm have not been subject of the prototype. Generally, the DPropR methods for multi-directional DB2-DB2 replication could also be applied in the proposed heterogeneous solution without major changes.
- Support for target table types like CCDs (consistent changed data), Replica etc. has not been implemented.

9.2 Future Work

Extensive research in the field of accessing heterogeneous, including non- relational, data sources has been performed in the scope of the Garlic project at the IBM Almaden Research Center (See [13]). Garlic not only addresses data transformation, but also differences in the structure of the data (schema transformation). The result of this paper and the Garlic project could form a basis for future work in the field of heterogeneous data replication.

A Example XSL-T transformation

The example shows the transformation of an abstract trigger definition to DBMS specific SQL-DDL statements (in this case Sybase) using an XSLT stylesheet.

This could be the abstract definition of the Capture Triggers for a simple table with the columns:

```
<?xml version="1.0" encoding="UTF-8"?>
<trigger>
<capture_trigger before_image="" ccd_name="CCD124028" ccd_owner="phaase"
 table_name="perform" table_owner="phaase">
<column name="ID"><integer/></column>
<column name="RL"><real/></column>
<column name="CH"><char size="10"/></column>
</capture_trigger>
</trigger>
```

The Sybase stylesheet would look like this (only the part for the Insert trigger is shown):

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
<xsl:template match="/trigger">
<xsl:apply-templates/>
</xsl:template>
<xsl:template match="capture_trigger">
<statements>
<statement>
CREATE TRIGGER <xsl:value-of select="@ccd_owner"/>.I<xsl:value-of
 select="@ccd_name"/>
 ON <xsl:value-of select="@table_owner"/>.<xsl:value-of select="@table_name"/>
FOR INSERT AS
DECLARE @NEWSYNCH BINARY(10)
BEGIN
SELECT @NEWSYNCH=(CONVERT (BINARY(10), (CONVERT (NUMERIC(20),
(SUBSTRING(CONVERT(CHAR(25), GETDATE(), 9), 8 , 4) +SUBSTRING(CONVERT(CHAR(8),
 GETDATE(), 2), 4 , 2) +SUBSTRING(CONVERT(CHAR(8), GETDATE(), 2), 7 , 2) +
 SUBSTRING(CONVERT(CHAR(8), GETDATE(), 8), 1 , 2) +SUBSTRING(CONVERT(CHAR(8),
 GETDATE(), 8), 4 , 2) +SUBSTRING(CONVERT(CHAR(8), GETDATE(), 8), 7 , 2)
 +SUBSTRING(CONVERT(CHAR(25), GETDATE(), 9), 22 , 3))))))
INSERT INTO <xsl:value-of select="@ccd_owner"/>.<xsl:value-of
 select="@ccd_name"/>
   (<xsl:for-each select="column">
      <xsl:if test="position()>1">,</xsl:if><xsl:value-of select="@name"/>
    </xsl:for-each>
    <xsl:if test="not(@before_image='')">
      <xsl:for-each select="column">,<xsl:value-of select="../@before_image"/>
       <xsl:value-of select="@name"/>
      </xsl:for-each></xsl:if>,IBMSNAP_COMMITSEQ,IBMSNAP_INTENTSEQ,
    IBMSNAP_OPERATION,IBMSNAP_LOGMARKER) SELECT
   <xsl:for-each select="column">
      <xsl:if test="position()>1">,</xsl:if><xsl:value-of select="@name"/>
```

```
    </xsl:for-each>
    <xsl:if test="not(@before_image='')">
      <xsl:for-each select="column">,null</xsl:for-each>
    </xsl:if>, @NEWSYNCH, @NEWSYNCH,'I', GETDATE() FROM inserted
END
</statement>

...

</statements>
</xsl:template>
</xsl:stylesheet>
```

The result after the transformation are Sybase specific SQL-DDL statements (again, only the Insert trigger is shown):

```
<?xml version="1.0" encoding="UTF-8"?>
<statements><statement>
CREATE TRIGGER phaase.ICCD124028
 ON phaase.perform
FOR INSERT AS
DECLARE @NEWSYNCH BINARY(10)
BEGIN
SELECT @NEWSYNCH=(CONVERT (BINARY(10), (CONVERT (NUMERIC(20),
(SUBSTRING(CONVERT(CHAR(25), GETDATE(), 9), 8 , 4) +SUBSTRING(CONVERT(CHAR(8),
 GETDATE(), 2), 4 , 2) +SUBSTRING(CONVERT(CHAR(8), GETDATE(), 2), 7 , 2) +
 SUBSTRING(CONVERT(CHAR(8), GETDATE(), 8), 1 , 2) +SUBSTRING(CONVERT(CHAR(8),
 GETDATE(), 8), 4 , 2) +SUBSTRING(CONVERT(CHAR(8), GETDATE(), 8), 7 , 2)
 +SUBSTRING(CONVERT(CHAR(25), GETDATE(), 9), 22 , 3))))))
INSERT INTO phaase.CCD124028
   (ID,RL,CH
      ,IBMSNAP_COMMITSEQ,IBMSNAP_INTENTSEQ,
    IBMSNAP_OPERATION,IBMSNAP_LOGMARKER) SELECT
   ID,RL,CH
    , @NEWSYNCH, @NEWSYNCH,'I', GETDATE() FROM inserted
END
</statement>

...

</statements>
```

References

[1] Amdahl Corp. Global information sharing software: Selecting a data replicator. http://amdahl.com/doc/products/storage/gis/mm002623/data.html, 1998.

[2] Informix Corp. Enterprise replication: A high-performance solution for distributing and sharing information. http://www.informix.com/informix/whitepapers/entrep.pdf, 1998.

[3] Microsoft Corp. Accessing heterogeneous data with microsoft sql server 7.0. http://www.microsoft.com/sql/interopmigrate/heterodata.htm, 1998.

[4] Microsoft Corp. Replication for microsoft sql server 7.0. http://www.microsoft.com/sql/DeployAdmin/replication.htm, 1998.

[5] Sybase Corp. Data movement. http://www.powersoft.com /products/middleware/dmove.html, 2000.

[6] Rick Cattell Graham Hamilton. Jdbc: A java sql api, 1997.

[7] Eliotte Rusty Harold. *XML Bible*. IDG Books, 1999.

[8] Andreas Heuer. *Datenbanken: Implementierungstechniken*. MITP, 1st edition, 1999.

[9] IBM. *DB2 DataJoiner for Windows NT Systems: Planning Installation and Configuration Guide*. IBM Press, 1st edition, 1997.

[10] IBM. *My Mother Thinks I'm a DBA!* IBM Press, 1st edition, 1999.

[11] IBM. *Replication Guide and Reference, Version 7*. IBM Press, 1st edition, 1999.

[12] Jamie Jaworski. *Java 2 Platform Unleashed.* SAMS, 1st edition, 1999.

[13] B. Niswonger Laura M. Haas, R.J. Miller. Transforming heterogeneous data with database middleware: Beyond integration. *Data Engineering Bulletin*, 1999.

[14] Mark Hapner Seth White. Jdbc 2.0 standard extension api, 1998.