

Database Integration With DB2® Relational Connect Building Federated Systems with Relational Connect and Database Views

Stephen H. Rutledge
John Medicke
IBM Software Group Center for e-business Solution Integration
Raleigh, NC

Contents

1	Introduction.....	2
2	Background	2
3	Federated databases.....	5
4	A federated database scenario.....	7
4.1	DB2 Relational Connect	7
4.2	Bank database	9
4.3	Brokerage database	10
4.4	Bank and brokerage database design	11
4.5	Federated data model	12
4.6	Federated database design.....	13
5	Data access layers	15
5.1	Low-level database APIs (DB2-CLI) and embedded SQL precompilers.....	16
5.2	JDBC and SQLJ	16
5.3	Enterprise JavaBeans (EJBs)	17
5.4	Web services	17
5.5	Decoupling SQL complexity from the data access layer	19
6	Performance implications	19
7	Proof-of-concept implementation.....	21
8	Conclusion	22
9	Appendix: Excerpts from scenario installation scripts	23
9.1	Bank database DDL	23
9.2	Brokerage database DDL.....	25
9.3	Federated common database DDL.....	27
10	References	31
11	Notices and trademarks	32
12	About the authors.....	32

© Copyright International Business Machines Corporation, 2001. All rights reserved.

1 Introduction

The topics covered in this paper include:

- An architectural context for the use of IBM® DB2® Relational Connect in federated database systems.
- A design approach using database views for integrating multiple heterogeneous databases together into a single synchronous, consistent federated database system.
- A simple example showing implementation of a federated database system using DB2 Relational Connect and database views.

2 Background

Sharing data

The introduction of database management systems (DBMSs) in the early 1970s was intended to provide more central control over information and to reduce data redundancy. This was achieved by allowing the sharing of a central database between systems that could be developed in a distributed fashion. Systems could be distributed, and the database would become the central data authority. The key benefit provided by the DBMS approach was the ability to separate process (application software) from data. This eliminated the need for each application to have its own proprietary and redundant data [6].

However, a shared database requires more than one development team to work together. Previously, most data structures were designed specifically for the system that was processing it. Rather than data requirements being defined by a single system and its transactions, DBMSs required a more enterprise-wide view of the data because multiple systems would be utilizing it.

Because team efforts can sometimes be more difficult to implement than individual efforts, it was not uncommon for people to resist the concept of a common database between systems. By the time DBMSs made their way into most enterprises, decentralized computing had led to the loss of control, and the key benefit of DBMSs was lost in the confusion. Most implementations of DBMSs just became replacements of single systems' flat files or indexed files – the proverbial paving of cow paths. Very few systems were merged into a shared database. Whereas previously there were three applications, now there were three applications *and* three databases [2].

During the 1980s, while personal computers proliferated and data from mainframe systems remained difficult to access, a large number of stand-alone PC applications were created to replicate the functions of certain key central systems. These "shadow systems"

were developed by individual departments to support a variety of business functions, such as budget and account management. These stand-alone systems carried the same or similar data as was found in the central systems, periodically downloaded from the mainframe or sometimes even re-keyed into local systems [7].

Distributed computing

In the 1990s, distributed computing began to offer the potential to radically transform our businesses. This transformation has led to the expectation that distributed computing will support the interconnection of a rich mix of heterogeneous components into a single, integrated information system solution. However, this is still a work-in-progress. In today's typical computing environment, there are still multiple systems keeping copies of the same data, stored in incompatible formats, and accessed by heterogeneous database management systems. It is still difficult to aggregate this data for institution-wide reporting, and to reconcile the conflicting information produced by these myriad systems.

Heterogeneity is a fact of life in most businesses: a wide variety of desktop computers and desktop software; an equally wide variety of department branch and division systems; and a mixed-vendor environment of database systems. Heterogeneity and the "open standards" movement offer freedom from the constraints of a single-vendor or proprietary information system architecture by allowing organizations greater choice in the marketplace [5].

Distributed data processing

Distributed data processing offers businesses an opportunity to integrate fragmented data resources. One approach to centralizing data is to simply decommission existing database systems and to build a new integrated database. However, it is often prohibitively expensive and inconvenient to retool existing systems. An alternative approach is to build an integration layer on top of pre-existing systems. Building an integration layer on top of existing database systems is a challenge in complexity and performance, but this option sometimes makes the most business and engineering sense.

In a data-sharing environment, there is no single best architecture that will solve all problems. The trade-offs between remote data access and data redundancy that provides local access is a thoroughly studied issue. On the one hand, replicated data for local access creates the challenge of data inconsistency but provides better performance for frequently accessed data, diminishes reliance on the network (the least reliable of the common computing elements), and provides greater reliability by providing a local copy of data. On the other hand, directly accessing the remote database removes the danger of data inconsistency; however, it requires a fault-tolerant architecture because it has now become a single point of failure for many applications, and it may easily become a performance bottleneck due to irreducible network latency.

In short, there is no easy answer. Large installations of database systems may be accessed hundreds of thousands of times a minute. The irreducible latency present even in a fully optical network is not capable of supporting such a performance requirement. Indeed, local disks are also too slow, and most of this sort of information is cached off disk and into memory. In some organizations, if critical data is unavailable for even a matter of minutes, it could affect millions of dollars of revenue. This is why remote data access is not used in such large-scale situations where high availability is critical.

There are many small and medium-weight applications with modest performance requirements for data. Often, such applications are designed to work with a copy of data because getting a copy and loading it on a local database seems like the easiest solution. Such design does not factor in the cost of maintaining a separate copy of the data. When the applications are put into production and begin having problems keeping their data in sync, these costs become all too apparent. Such applications would probably do better to remotely reference their data.

In such cases, it is a good architecture to remotely reference application databases for shared data. Such “distributed” databases need to incorporate some high-availability design, depending on the weight of the applications served and their availability requirements. Each application should be analyzed to determine its performance and reliability requirements.

Distributed database management systems

A distributed database management system is a collection of centralized (or local) database management systems that are linked through a network and integrated in their operations. Ideally, this distributed functionality is achieved within the DBMS software itself and does not rely on custom-written application programs and inter-application "bridges" to maintain the integrity of the database or the appearance of logical integration.

Here are the commonly recognized features of a distributed database management system (paraphrased from Date’s 12 Rules of Distributed Databases) [5].

Table 1. Date's 12 rules of distributed databases

1	Local autonomy	Each site is independent of every other site.
2	No reliance on central site	No single site is more important to the distributed database than any other.
3	Continuous operation	Should not require shutdown for routine operations.
4	Data location independence	Entities accessing data will not know where it is stored.
5	Data fragmentation independence	A fragmented, distributed table must appear to be a single table to any entity accessing it.

6	Data replication independence	Copies of replicated data must be updateable.
7	Distributed query processing	Will provide distributed query optimization.
8	Distributed transaction management	Must provide transaction integrity.
9	Hardware independence	Does not require specific hardware platforms.
10	Operating system independence	Does not require specific operating systems.
11	Network independence	Does not require specific network implementations or protocols.
12	DBMS independence	Must support heterogeneous database management systems.

Distributed data processing is not yet a full and complete set of mature technologies. The ideal distributed database management system is still as much vision as it is reality. Within the more narrow constraints of single-vendor and proprietary solutions, the technical problems of location transparency, replication transparency, and full logical integration (including concurrency) across multiple computing platforms are beginning to be solved. In the heterogeneous, multi-vendor environment that is often typical of distributed data processing, distributed database management systems have not yet achieved this same level of integration.

While some software developers and vendors pursue the ideal of full logical integration within the DBMS itself, various intermediate strategies are being offered by other DBMS and software vendors to achieve some of the same objectives.

Standardization on DBMSs and on SQL for data access, manipulation, and management can provide the necessary shared infrastructure to permit some level of integration among heterogeneous database management systems. The basic strategy for integration in a heterogeneous RDBMS environment is the SQL gateway, a facility for translation of SQL dialects among unlike database management systems. This strategy is most effective at providing users with a unified view of distributed databases and offering a single, SQL-based interface to the data [1].

3 Federated databases

A federated database is a distributed database management system composed of different DBMSs, some of which may be from different database vendors. A federated database pulls together data from several different physical locations and abstracts the data to appear as a single, consistent view. Federated databases typically support both read and write operations by means of distributed transaction management. This is an appropriate architecture for small and medium-weight applications that have modest transaction and performance requirements.

The particular scenario examined here has several unique design points:

- This federated database need only provide read access.
- This federated database is composed of pre-existing production database systems that will remain essentially intact.
- Data stays partitioned into multiple disparate physical locations. There is no explicit replication of data into a central location.

So, the architectural challenge is to link the data in multiple heterogeneous databases together into a single synchronous, consistent view. This “integrated view” must be available for query access, which simplifies the task of querying for result sets that span the multiple heterogeneous database systems.

Architectural considerations

When integrating database systems, there is an assumption that these systems must contain information that can be assembled as parts of a whole. But, similar information in different systems will likely be represented in different formats and structures, and at different levels of detail. Integration of data from pre-existing production systems will require some transformation and aggregation to make heterogeneous partitioned data appear as a consistent whole.

One consideration is whether to have this data abstraction occur within the database layer or within the application layer. The preferred approach is to keep data transformation in the database, because this is what the DBMS does best. Ideally, this layer of abstraction will occur within database views. Database views can serve to bring together partitioned data and apply mappings and transformation where appropriate by the judicious use of SQL.

It is useful to examine a federated database architecture in the context of the software engineering principles of spatial locality, simplicity, and abstraction [5].

Partitioning shared databases and distributing the partitions can improve performance by eliminating inessential sharing of a key resource and by reducing the distance between the resource and its users. Leaving the pre-existing production database systems essentially intact will preserve existing spatial locality because data has already been partitioned to match existing usage patterns. The downside of leaving pre-existing production database systems intact is that spatial locality does not exist for the federated database.

Adding replication to this architecture provides spatial locality at the expense of complexity. There are performance, maintenance and scalability trade-offs between synchronization of copies of data images vs. sharing of a single data image. If sharing of a single data image meets requirements, this is probably the simpler solution.

However the federated database is assembled, whether replicated or not, abstraction is a key benefit. The federated database system can hide the “unnecessary details”

(heterogeneous database vendors, data structures, data representations, and physical data locations).

4 A federated database scenario

Below is a simple proof-of-concept design for a federated database scenario using IBM's DB2 Relational Connect as a SQL gateway to connect a legacy bank system in DB2 and a legacy brokerage system in Oracle. Although this is a somewhat simple scenario, it still provides useful insights into the design and performance of a real-world system.

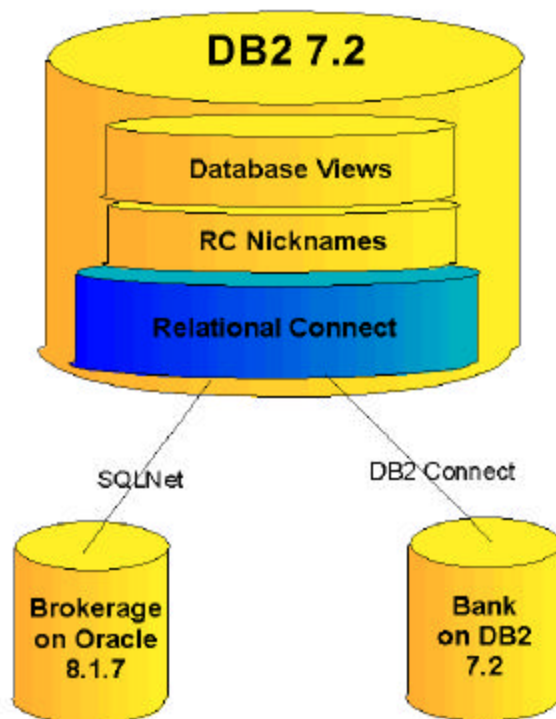


Figure 1. A federated database scenario

4.1 DB2 Relational Connect

IBM's DB2 Relational Connect is a "best in class" SQL gateway that provides much more than SQL dialect translation. Relational Connect provides native read access to Oracle, Sybase, Informix® and Microsoft SQL Server databases. DB2 Relational Connect makes data source access completely transparent to the calling application and supports the full SELECT API features of DB2. Without any loss of function, automatic compensation built into DB2 masks differences between the DB2 capabilities and the data source function.

IBM's DB2 Relational Connect, in conjunction with DB2 Universal Database® Enterprise Edition (EE) or Extended Enterprise Edition (EEE), allows you to:

- Build a highly integrated, consistent system that exploits the full power of business intelligence through the DB2 SQL interface using DB2 Relational Connect.
- Optimize business data capabilities by combining and managing data in other DBMSs with data managed by DB2.
- Adopt a data inclusion strategy, which means you can keep data where it makes sense, even if it is in different vendors' products.
- Query and retrieve information from Informix Dynamic Server™ (IDS) data sources.
- Query and retrieve information from Oracle Versions 7 and 8.
- Query and retrieve information from Sybase Versions 10, 11, and 12 (Windows NT and AIX®) data sources.
- Query and retrieve information from Microsoft SQL Server, Versions 6.5 and 7 (Windows NT) data sources.

A DB2 federated system with Relational Connect currently operates under some restrictions. Distributed requests are limited to read-only operations in DB2 Version 7, but write capability is being planned, most likely in the next version. In addition, utility operations (such as LOAD, REORG, REORGCHK, IMPORT and RUNSTATS) cannot be executed against remote tables, aliases and views. However, a pass-through facility is available to submit DDL and DML statements directly to DBMSs using the SQL dialect associated with that data source [4].

For more information on IBM's DB2 Relational Connect, see <http://www-3.ibm.com/software/data/db2/relconnect/>.

4.2 Bank database

The bank database represents a legacy database system in a bank branch office. The DBMS is DB2. Figure 2 shows the data model.

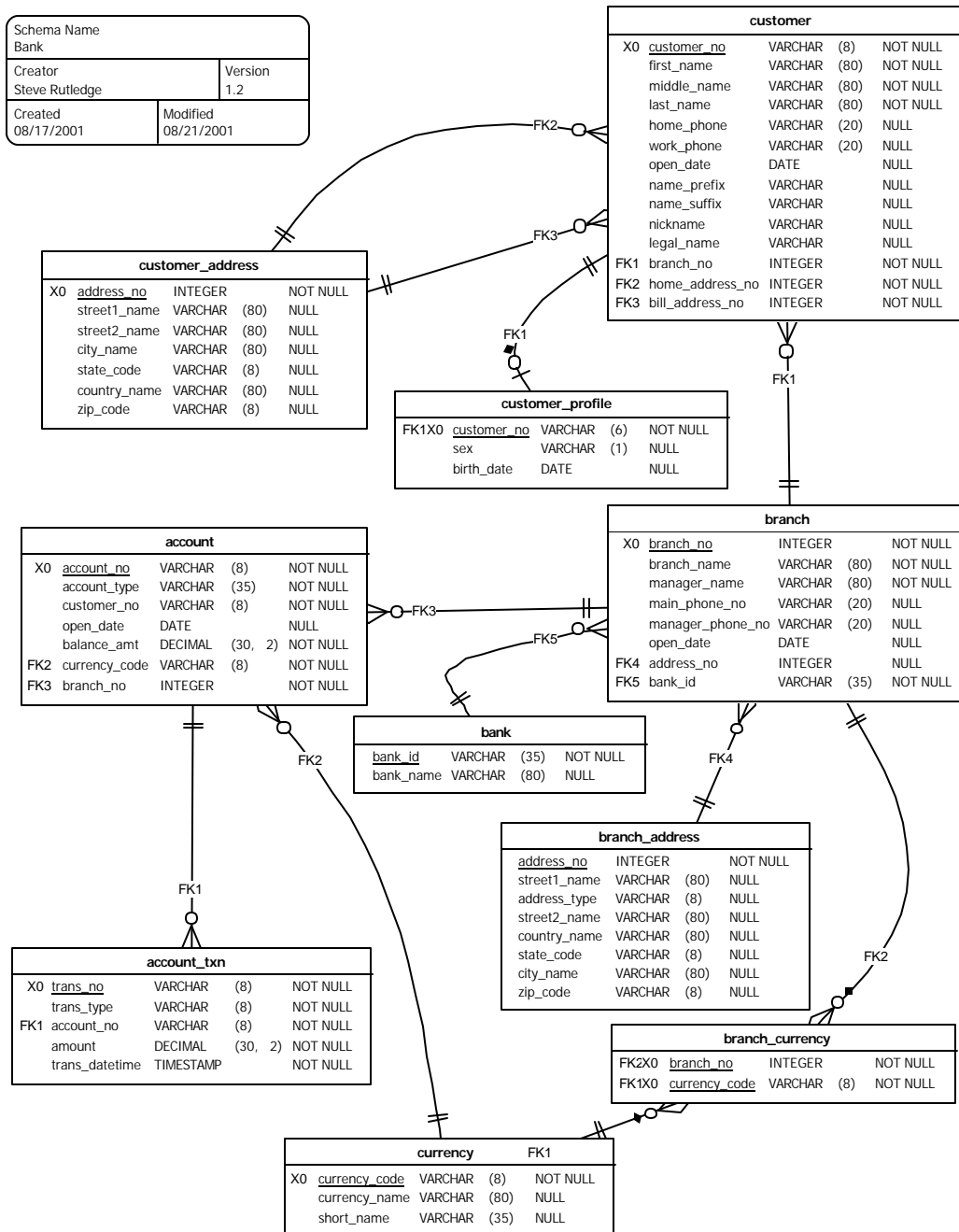


Figure 2. A data model for a bank branch office

4.3 Brokerage database

The brokerage database represents a legacy database system in a brokerage branch office. The DBMS is Oracle. Figure 3 shows the data model.

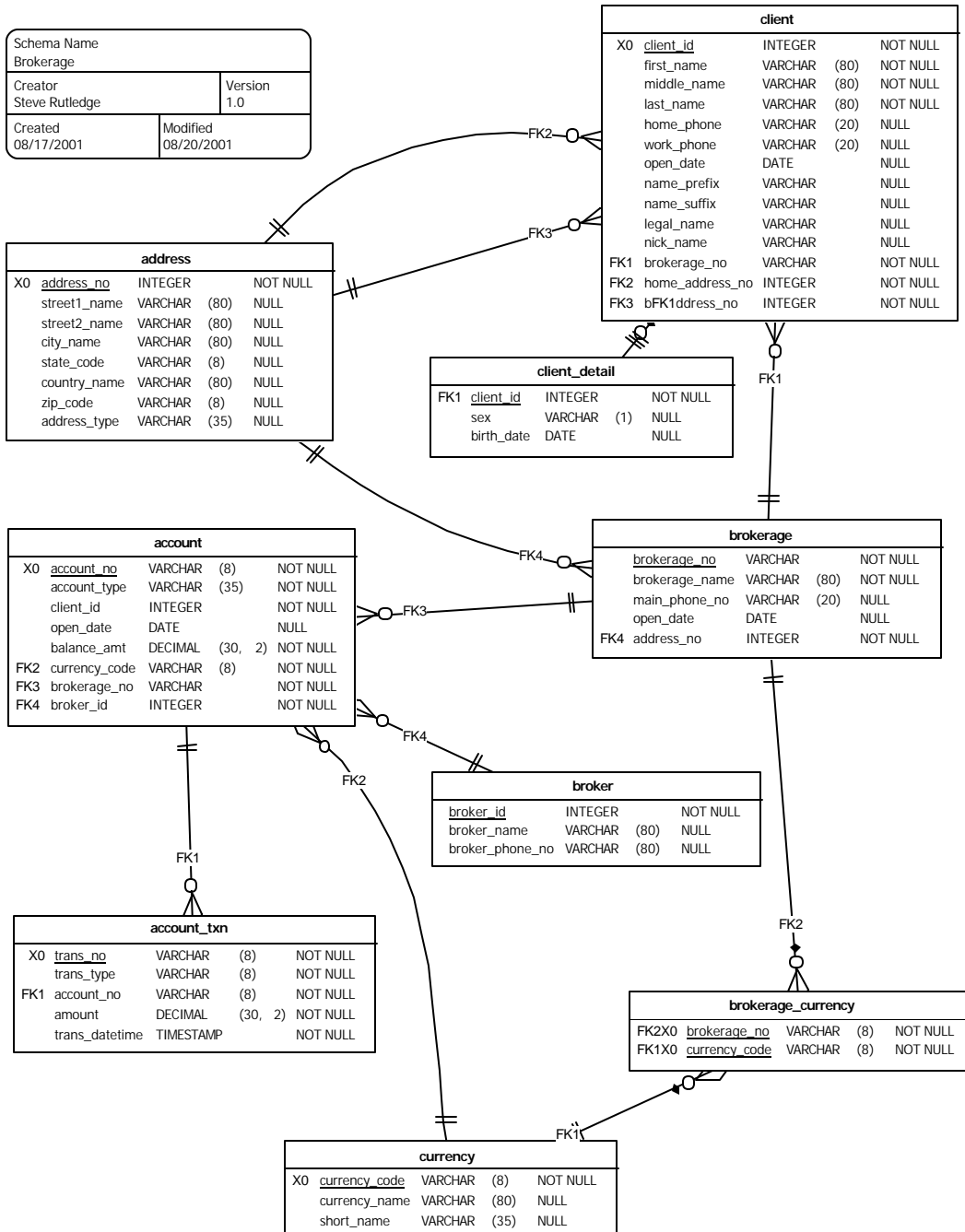


Figure 3. A data model for a brokerage branch office

4.4 Bank and brokerage database design

The data models in the bank and brokerage databases intentionally contain design differences. Table names for data with similar meanings are different. For example, bank.branch is similar to brokerage.brokerage.

Column names are different, too, such as branch_no vs. brokerage_no and branch_name vs. brokerage_name.

Many column data types are different (especially key columns). For example, brokerage_no (varchar) is the key for the brokerage.brokerage table, while bank_no (integer) is the key column for the bank.branch table.

Several entities are implemented at different levels of abstraction or in different implementations. Addresses appear in the bank data model in two places. Customer addresses are in the customer_address table, and branch addresses are in the branch_address.table. In the brokerage data model, the address entity is implemented as a single brokerage.address table.

In both the bank and brokerage models there is a concept of a customer contact, or representative for the business, but this is implemented very differently in each model. In the brokerage model, the broker entity is factored out into a separate table, and has a relation to the account table. A broker may be related to one or several accounts and an account must be related to one and only one broker. The account table then has a relation to the brokerage table. An account must be related to one and only one brokerage, and a brokerage may be related to several accounts.

In the bank model, the customer contact appears in the bank table as the manager name and phone number columns. An account must be related to one and only one branch. A branch may be related to several accounts. There is no equivalent table for the brokerage's broker table.

There are also differences in data representation. For example, bank account types are either "C" for Checking Account, or "S" for Savings Account. There are account types in the brokerage with the same meaning but different representation: "CASH" is the account type code for the brokerage equivalent to a checking account.

All these differences between the data models need to be resolved by the federated database.

4.5 Federated data model

The federated data model is represented by the common schema. The common schema is implemented in DB2. Here is the data model:

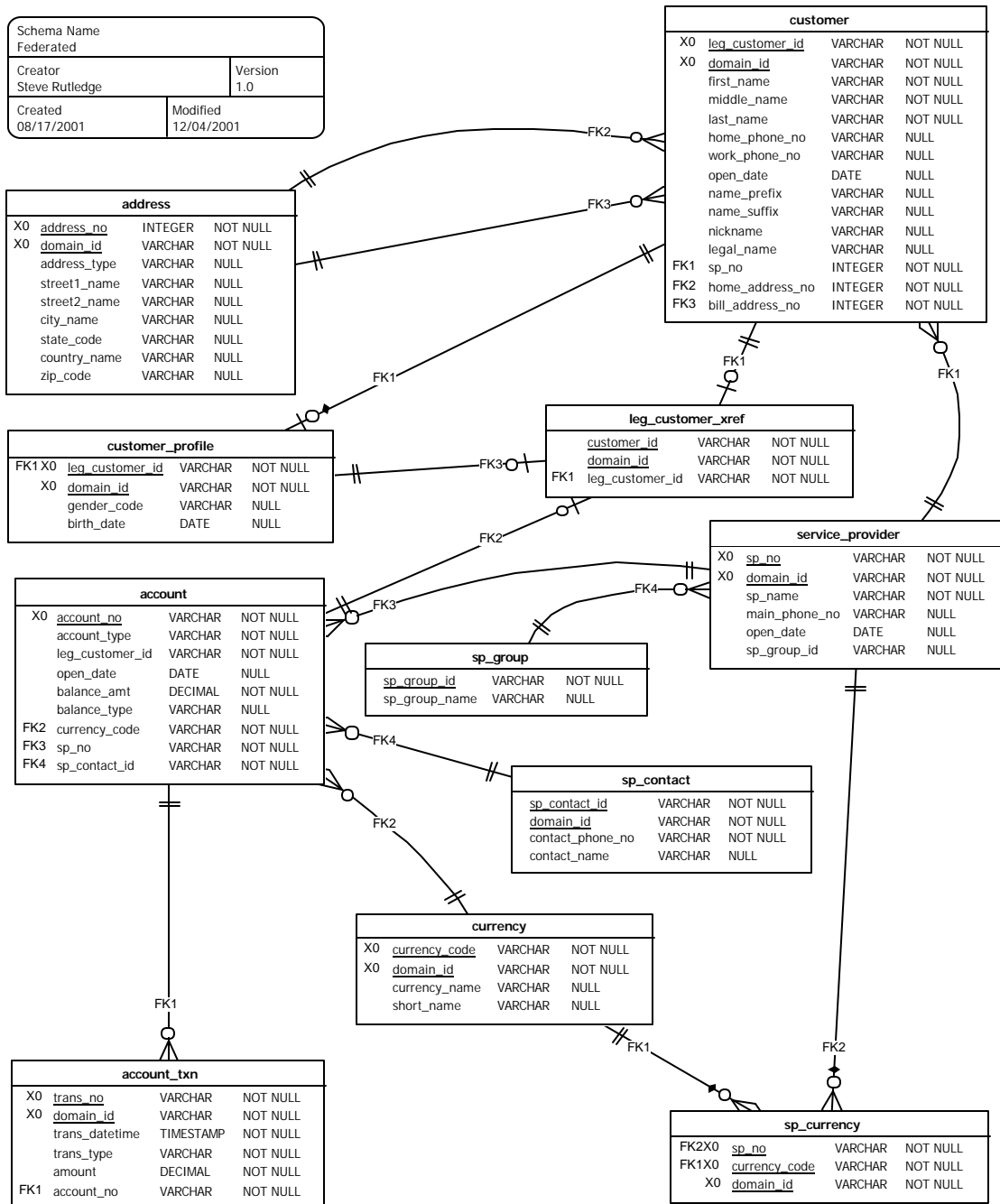


Figure 4. Common schema for federated data model

4.6 Federated database design

Abstraction

The general approach to resolving the differences between the bank and brokerage data models is by using SQL views to transform the data. Each entity in the federated database has a view that aggregates and transforms the legacy data.

The views reference the legacy data through Relational Connect *nicknames*. Nicknames are identifiers stored in the federated database that reference specific data source objects (tables, aliases, views). Applications reference nicknames in queries just like they reference tables and views.

Here is an example:

```
1  create federated view common.account as
2  select
3      account_no,
4      'BANK' domain_id,
5      case upper(account_type)
6          when 'C' then 'CMA'
7          when 'S' then 'SDA'
8      else upper(account_type)
9      end account_type,
10     customer_no leg_customer_id,
11     open_date,
12     balance_amt,
13     'Avail' balance_type,
14     currency_code,
15     varchar(char(branch_no)) sp_no,
16     varchar(char(branch_no)) sp_contact_id
17 from bank.account
18 union all
19 select
20     account_no,
21     'BROKERAGE' domain_id,
22     case upper(account_type)
23         when 'CASH' then 'CMA'
24         when 'MKT' then 'MMA'
25     else upper(account_type)
26     end account_type,
27     varchar(char(integer(client_id))) leg_customer_id,
28     date(open_date) open_date,
29     balance_amt,
30     'Avail' balance_type,
31     currency_code,
32     brokerage_no sp_no,
33     varchar(char(integer(broker_id))) sp_contact_id
34 from oaccount;
```

There are two select clauses in this view joined by a UNION ALL. The clause from lines 2-17 is against the account table in the bank database. The clause from lines 19-34 is

against the oaccount nickname. This Relational Connect nickname references the account table in the brokerage Oracle database.

The FEDERATED keyword indicates that the view being created references a nickname. If not specified, a SQLSTATE 01639 warning will be issued when creating the view.

Many columns have different names between brokerage and bank databases. This is hidden within the view by using column aliases. Column aliases are a DB2 SQL feature that allows renaming of columns within select clauses. There are column aliases on lines 4, 9, 10, 13, 15, 16, 21, 26, 27, 28, 30, 32 and 33.

There are data type transformations for several columns on lines 27, 28 and 33.

There is also an example of code mapping on lines 5-9 and 22-26. Account types are represented differently between the bank and brokerage models. This mapping is handled with the DB2 SQL CASE clause.

Aggregation

UNION ALL was chosen for aggregation over UNION for two reasons:

- UNION ALL preserves duplicate rows – this was our preferred behavior.
- UNION requires full retrieval of all rows, sorting, and duplicate elimination before the first row is returned. UNION ALL does not require that additional processing.

Aggregation of partitioned data with unions instead of joins has historically been a useful approach in partitioned tables in data warehousing environments and is similarly appropriate here. Staying with the “keep it simple” rule, UNION ALL is a fundamentally simpler operation than JOIN.

The leg_customer_xref table is the only local table in the federated database (the rest are views). This table maps inconsistent legacy customer identifiers to a new common identifier. The leg_customer_xref table was the only mapping table implemented in the proof-of-concept scenario, but the same approach could be extended to map service providers, accounts, or any other inconsistent legacy identifiers.

The concept of *domain* is introduced in the federated data model as a way of preserving information about data origin. Each view introduces a domain_id column to explicitly preserve the data origin. Without the domain_id column, it would not be possible to query just the bank or just the brokerage because the two underlying sources would be aggregated with no explicit way of de-aggregating them. In this simple scenario, there are only two domains: BANK and BROKERAGE. These domains are hardcoded into each view. In a more complex implementation requiring more flexibility, domains could be factored out into a separate table.

Design implications

This federated database design uses database views as the mechanism for abstraction, aggregation, and federated data access. This is both the strength and weakness of this design. Views are probably the simplest mechanism for encapsulating all of this data manipulation into one place, but they create performance, flexibility and maintenance limitations.

In appropriate design applications, database views will simplify maintenance tasks by providing a single logical location for all data manipulation. This is easier to maintain than the same logic decoupled and spread over several different architectural layers, languages, and contexts.

In federated database designs, performance is often a relevant concern. Consolidating performance-sensitive code into a central location will make performance analysis simpler. The database views in this design are very performance-sensitive.

Database views can provide flexibility by hiding data manipulation changes from application code. For example, SQL within database views can be changed for a performance optimization without affecting application code.

On the other hand, when database views reach a certain level of complexity, performance analysis, maintenance, and design changes become very difficult. Factoring a single view into several new views to manage complexity may appear to be the simple answer, but database optimizers do not typically rewrite multi-level views. Without query rewrite, views produce large intermediate result sets that can cause serious performance problems.

Flexibility for adding new legacy databases to a view-based federated system is limited. Remember, each view contains a UNION ALL for each legacy database table. Adding legacy databases will rapidly increase view complexity.

5 Data access layers

A federated database system is useless if it does not provide ready access to data. Ideally, access is via common mechanisms already used for database access. This federated system design does not “break” the integration points with familiar database access layers. It is still within the context of a relational database system. As stated earlier, it is ideal for the federated database functionality to be achieved within the DBMS software itself, and not rely on custom-written application programs and inter-application "bridges" to maintain the integrity of the database or the appearance of logical integration. DB2 and DB2 Relational Connect provide this functionality

The following sections are a brief survey of common data access approaches in the context of a federated database system.

5.1 Low-level database APIs (DB2-CLI) and embedded SQL precompilers

An application that uses an embedded SQL interface requires a precompiler to convert the SQL statements into code. The code is then compiled, bound to the database, and executed. In contrast, a DB2 Call Level Interface (CLI) application does not have to be precompiled or bound. Instead, it uses a standard set of functions to execute SQL statements and related services at runtime.

This difference is important because precompilers have traditionally been specific to each database product. This ties applications to a particular product. DB2 CLI enables creation of portable applications that are independent of any particular database product. This independence means that DB2 CLI applications do not have to be recompiled or rebound to access different DB2 databases, including federated database systems. They just connect to the appropriate database system at runtime.

These are probably the most labor-intensive and flexible approaches to data access. These APIs reference the database through the SQL interface. So the federated database system is transparent to the low-level interface.

5.2 JDBC and SQLJ

Java™ Database Connectivity (JDBC) is an API that provides access to tabular data from Java applications. It provides cross-DBMS connectivity to a wide range of SQL databases. The JDBC 3.0 API also enables access to non-relational data sources, like spreadsheets or flat files, through its support of the DATALINK data type. To use the JDBC API with a particular database management system, you need a JDBC technology-based driver to mediate between JDBC technology and the database. Depending on various factors, a driver might be written purely in the Java programming language or in a mixture of the Java programming language and Java Native Interface (JNI) native methods. There are also JDBC-ODBC bridge drivers that make most Open Database Connectivity (ODBC) drivers available to programmers using the JDBC API.

SQLJ is a set of programming extensions that allow a programmer using the Java programming language to embed SQL database statements. SQLJ is similar to existing extensions for SQL that are provided for C, FORTRAN, and other programming languages. IBM, Oracle, and several other companies have proposed SQLJ as a standard and as a simpler and easier-to-use alternative to JDBC.

SQLJ is more concise and thus easier to write than JDBC, and it provides compile-time schema validation and syntax checking for easier debugging. Input to SQLJ can be either a file of SQLJ clauses or a file of Java source code in with embedded SQLJ clauses. The SQLJ precompiler translates the SQLJ clauses into their equivalent JDBC calls. SQLJ supports static SQL.

JDBC and SQLJ are two common Java-specific data access approaches that are less labor-intensive than low-level database APIs. These approaches reference the database through the SQL interface, so the federated database system is transparent and can be accessed just the same as any other database.

For more information about JDBC, see <http://java.sun.com/products/jdbc>. For more information out SQLJ, refer to the following ISO specifications at www.iso.org: [ISO/IEC FCD 9075-13](#) Information technology -- Database languages -- SQL -- Part 13: Java Routines and Types (SQL/JRT), [ISO/IEC 9075-10:2000](#) Information technology -- Database languages -- SQL -- Part 10: Object Language Bindings (SQL/OLB).

5.3 Enterprise JavaBeans (EJBs)

The Enterprise JavaBeans (EJB) specification defines the server component model for javabeans. An EJB is a specialized, non-visual JavaBean that runs on a server. An EJB is primarily a contract between a server-side JavaBean and component coordinator, known as the server-side container, or Object Transaction Monitor (OTM).

The OTM provides the following services:

- Component packaging and deployment.
- Declarative transaction management.
- Factory support.
- Bean activation and passivation.
- Bean state management.
- Security [3].

For data access JavaBeans within this model, database access usually occurs with JDBC or SQLJ; there is no restriction.

5.4 Web services

Web services are a new standards-based architecture for providing dynamic, loosely coupled internet-enabled applications. Web services are based on WSDL, SOAP, UDDI and XML specifications.

Web Service Description Language (WSDL) is an XML format for describing network services as a set of services operating on messages containing either document-oriented or procedure-oriented information. The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define a service [8].

Simple Object Access Protocol (SOAP) is a lightweight protocol for exchange of information in a decentralized, distributed environment. It is an XML-based protocol that consists of three parts:

- An envelope that defines a framework for describing what is in a message and how to process it.
- A set of encoding rules for expressing instances of application-defined data types.
- A convention for representing remote procedure calls and responses [9].

Universal Description, Discovery and Integration (UDDI) is a platform-independent, open framework for describing services, discovering businesses, and integrating business services using the Internet.

The Extensible Markup Language (XML) is designed to improve the functionality of the Web by providing more flexible and adaptable information identification. It is called extensible because it is not a fixed format like HTML (a single, predefined markup language). Instead, XML is actually a metalanguage that lets you design your own customized markup languages for limitless different types of documents.

IBM provides several programs and tools for developing and deploying Web services. One way to access relational data through Web services is by using technology known as a document access definition extension (DADX).

A DADX allows transformation of an SQL statement into a Web service without writing any Java. A DADX document is an XML document that specifies operations, input and output parameters for the Web services supported by the DADX runtime. A document access definition (DAD) document (as implemented with DB2 XML Extender) describes the actual mapping between the SQL data and the desired XML output.

Web services reference the database via SQL, so the federated database system is transparent and can be accessed the same as any other low-level database.

For more information on Web services, see the following references:

- The white paper entitled “Dynamic e-business with DB2 and Web Services” at <http://www-4.ibm.com/software/data/pubs/papers/db2webservices/db2webservices.pdf>
- The DB2 Developer Domain article entitled “Running DB2 Web Services on WebSphere Application Server Advanced Edition 4.0” at <http://www7b.boulder.ibm.com/dmdd/library/techarticle/preisig/0108preisig.html>
- The DB2 Developer Domain article entitled “Sample Application using Web Services on WebSphere” at <http://www7b.boulder.ibm.com/dmdd/library/techarticle/preisig/0111preisig.html>
- The Web services Object Runtime Framework download and documentation at <http://www-4.ibm.com/software/data/db2/extenders/xmlxt/docs/v72wrk/WORF.html>

5.5 Decoupling SQL complexity from the data access layer

For all of the data access methods described above, the complexity of the “inner workings” of the federated database is hidden.

Queries against a federated database system are by nature performance sensitive. In the case of DB2, there is significant distributed optimization. DB2 with Relational Connect has the ability to provide cost-based global query optimization. The optimization information stored in the global catalog includes relative central processing unit (CPU) speeds, relative input/output (I/O) rates, communication bandwidth between the database and a remote data source, cardinality of the remote data table, and other information. But still, the network is a bottleneck.

Division of labor between a database developer and application developer leads to a natural decoupling between the SQL code for data access and the application code, usually in another language. The complexity of using SQL to abstract inconsistent heterogeneous databases is hidden in our proof-of-concept federated system design.

For the purposes of isolating performance-sensitive code and decoupling appropriate components for development and maintenance, database views are an advantageous design choice.

6 Performance implications

A small- to-medium weight federated database system with modest performance requirements can be categorized for purposes of estimating reasonable response time objectives.

A response time of a few seconds or more (5-15s) is a normal expectation. Active performance analysis and tuning is still required to meet this goal.

A response time of minutes or longer is easy to achieve with little or no tuning.

In the proof-of-concept scenario, there are two transactions, “GetAccounts” and “Get AccountTransactions”. Both of these queries are of medium complexity. Here are example queries:

GetAccounts:

```
SELECT DISTINCT
  lcx.customer_id,
  c.first_name,
  c.middle_name,
  c.last_name,
  c.name_prefix,
  c.name_suffix,
```

```

        c. nickname,
        c. legal_name,
        a. account_no,
        a. account_type,
        a. balance_amt,
        a. balance_type,
        sp. sp_name,
    sp. sp_no,
        sg. sp_group_name,
    sg. sp_group_id,
    add. street1_name,
    add. street2_name,
    add. city_name,
    add. state_code,
    add. country_name,
    add. zip_code,
    sc. currency_code
FROM
    COMMON. leg_customer_xref lcx,
    COMMON. customer c,
    COMMON. address add,
    COMMON. account a,
    COMMON. service_provider sp,
    COMMON. sp_group sg,
    COMMON. sp_currency sc
WHERE
    lcx.customer_id = '1001' and
    lcx.leg_customer_id = c.leg_customer_id and
    lcx.domain_id= sp.domain_id and
    c.leg_customer_id = a.leg_customer_id and
    a.domain_id = sp.domain_id and
    a.sp_no = sp.sp_no and
    sp.sp_name = 'Federal Credit' and
    sg.sp_group_id = sp.sp_group_id and
    sg.domain_id = sp.domain_id and
    add.domain_id=sp.domain_id and
    add.address_no=sp.address_no and
    add.address_type='bank' and
    sp.sp_no=sc.sp_no and
    sp.domain_id=sc.domain_id;

```

GetAccountTransactions:

```

SELECT DISTINCT
    a. account_no,
    a. account_type,
    a. balance_amt,
    a. balance_type,
    sp. sp_name,
    sp. sp_no,
    sg. sp_group_name,
    sg. sp_group_id,
    add. street1_name,
    add. street2_name,
    add. city_name,
    add. state_code,
    add. country_name,
    add. zip_code,
    sc. currency_code,
    at. amount,
    at. trans_type,
    at. check_No
FROM
    COMMON. address add,
    COMMON. account a,
    COMMON. service_provider sp,
    COMMON. sp_group sg,
    COMMON. sp_currency sc,
    COMMON. account_txn at
WHERE

```

```

a. account_no='90001' and
  a. account_type='CMA' and
  a. account_no=at. account_no and
  a. domain_id = sp. domain_id and
a. sp_no = sp. sp_no and
  at. trans_datetime>= '1999-08-19-00.00.00.000000' and
  at. trans_datetime<= '2007-01-19-00.00.00.000000' and
sg. sp_group_id = sp. sp_group_id and
sg. domain_id = sp. domain_id and
  add. domain_id=sp. domain_id and
  add. address_no=sp. address_no and
  add. address_type='bank' and
  sp. sp_no=sc. sp_no and
  sp. domain_id=sc. domain_id and
  at. domain_id=sp. domain_id;

```

We did no tuning of the transaction queries. We did light performance analysis of the GetAccount query. The local query performed significantly faster than the equivalent federated query on the first execution, but subsequent executions of the federated query were as fast as the local query. This slow first-time-only behavior was probably due to a combination of the network, database caching, and connection pool initialization. The largest one-time only factor was probably the connection pool initialization. In a production implementation, these factors have to be carefully evaluated as part of the performance analysis exercise.

7 Proof-of-concept implementation

Prerequisites

- Oracle 8.1.7 installation, legacy brokerage system created.
- DB2 7.2 installation, legacy bank system created.
- DB2 7.2 for federated system installation (in the proof-of-concept scenario, bank and federated system were on the same DB2 instance), in this order:
 - 1) DB2 7.1.
 - 2) DB2 Relational Connect 7.1.
 - 3) DB2 fixpack3 to upgrade to DB2 7.2.

DB2 Relational Connect setup

Here are the types of statements used to create the setup used in our proof of concept. For information on the SQL statements shown here, see the *SQL Reference*, which is available at the DB2 library page at: http://www-4.ibm.com/cgi-bin/db2www/data/db2/udb/win0s2unix/support/v7pubs.d2w/en_main.

- 1) Create an Oracle net8 wrapper.

```
create wrapper net8 library 'net8.dll';
```

2) Set up a relational connect mapping to the Oracle server. This creates a syscat.servers entry.

```
create server oraclouser
  type oracle
  version 8.1.7
  wrapper net8
  options (node 'oraclinstance');
```

3) Set up authorization. The DB2 user must be authorized to access Oracle tables. This creates a syscat.useroptions entry.

```
create user mapping for db2user
  server oraclouser
  options (remote_authid 'oraclouser',
          remote_password 'oraclpassword');
```

4) Set up access. Create DB2 nicknames for remote Oracle tables.

```
create nickname nickname for oraclouser.schema.table;
```

The actual DDL used for our proof of concept is available in the Appendix and as a downloadable zip file at

<http://www7b.software.ibm.com/dmdd/library/techarticle/rutledge/samplescripts.zip>.

These scripts are provided as-is with limited documentation, for the purposes of illustrating this scenario. Although these scripts work fine on our machines, there is no guarantee that it will work on yours.

8 Conclusion

A small-to-medium weight federated database system with modest performance requirements is architecturally appropriate for IBM's DB2 Relational Connect. Building an integration layer on top of existing database systems is a challenge in complexity and performance but sometimes makes the most business and engineering sense. The use of database views for integrating multiple heterogeneous databases together into a single synchronous, consistent federated database system can be a useful design approach in such scenarios.

The proof-of-concept design implemented in this document shows that the application of DB2 Relational Connect in production architectures is a pragmatic approach to data integration and sharing. Wholesale replacement of legacy systems is not always the best alternative. Sometimes building on what came before: evolution, not revolution, is the best solution.

9 Appendix: Excerpts from scenario installation scripts

A downloadable version of the scripts is available at

<http://www7b.software.ibm.com/dmdd/library/techarticle/rutledge/samplescripts.zip>.

9.1 Bank database DDL

```
create table account
(
    account_no      varchar(8) not null,
    account_type    varchar(35) not null,
    customer_no     varchar(8) not null,
    open_date       date,
    balance_amt     decimal(30, 2) not null,
    currency_code   varchar(8) not null,
    branch_no       integer not null,
    primary key (account_no)
);

create table bank
(
    bank_id         varchar(35) not null,
    bank_name       varchar(80),
    primary key (bank_id)
);

create table account_txn
(
    trans_no        varchar(8) not null,
    trans_type      varchar(8) not null,
    account_no      varchar(8) not null,
    check_no        varchar(8),
    amount          decimal(30, 2) not null,
    trans_datetime  timestamp not null,
    primary key (trans_no)
);

create table branch
(
    branch_no       integer not null,
    branch_name     varchar(80) not null,
    manager_name    varchar(80) not null,
    main_phone_no   varchar(20),
    manager_phone_no varchar(20),
    open_date       date,
    address_no      integer,
    bank_id         varchar(35) not null,
    primary key (branch_no)
);

create table branch_address
(
    address_no      integer not null,
    street1_name    varchar(80),
    address_type    varchar(8),
    street2_name    varchar(80),
    country_name    varchar(80),
    state_code      varchar(8),
    city_name       varchar(80),
    zip_code        varchar(8),
    primary key (address_no)
);

create table branch_currency
(
    branch_no       integer not null,
```

```

        currency_code varchar(8) not null,
        primary key (branch_no, currency_code)
    );

create table currency
(
    currency_code varchar(8) not null,
    currency_name varchar(80),
    short_name varchar(35),
    primary key (currency_code)
);

create table customer
(
    customer_no varchar(8) not null,
    first_name varchar(80) not null,
    middle_name varchar(80) not null,
    last_name varchar(80) not null,
    name_prefix varchar(80),
    name_suffix varchar(80),
    nickname varchar(80),
    legal_name varchar(80),
    home_phone varchar(20),
    work_phone varchar(20),
    open_date date,
    branch_no integer not null,
    home_address_no integer not null,
    bill_address_no integer not null,
    primary key (customer_no)
);

create table customer_address
(
    address_no integer not null,
    street1_name varchar(80),
    street2_name varchar(80),
    city_name varchar(80),
    state_code varchar(8),
    country_name varchar(80),
    zip_code varchar(8),
    primary key (address_no)
);

create table customer_profile
(
    customer_no varchar(6) not null,
    sex varchar(1),
    birth_date date,
    primary key (customer_no)
);

-- =====
-- add foreign keys
-- =====
alter table account add
    foreign key (currency_code)
        references currency
    foreign key (branch_no)
        references branch;

alter table account_txn add
    foreign key (account_no)
        references account;

alter table branch add
    foreign key (address_no)
        references branch_address;
alter table branch add
    foreign key (bank_id)
        references bank;

alter table branch_currency add

```

```

foreign key (currency_code)
  references currency
foreign key (branch_no)
  references branch;

alter table customer add
  foreign key (bill_address_no)
    references customer_address
  foreign key (home_address_no)
    references customer_address
  foreign key (branch_no)
    references branch;

alter table customer_profile add
  foreign key (customer_no)
    references customer;

```

9.2 Brokerage database DDL

```

-- =====
-- create tables
-- =====
create table account
(
  account_no      varchar(8) not null,
  account_type    varchar(35) not null,
  client_id       integer not null,
  open_date       date,
  balance_amt     decimal(30, 2) not null,
  currency_code   varchar(8) not null,
  brokerage_no    varchar(8) not null,
  broker_id       integer not null,
  primary key (account_no)
);

create table account_txn
(
  trans_no        varchar(8) not null,
  trans_type      varchar(8) not null,
  account_no      varchar(8) not null,
  check_no        varchar(8) null,
  amount          decimal(30, 2) not null,
  trans_datetime  date not null,
  primary key (trans_no)
);

create table address
(
  address_no      integer not null,
  street1_name    varchar(80),
  street2_name    varchar(80),
  city_name       varchar(80),
  state_code      varchar(8),
  country_name    varchar(80),
  zip_code        varchar(8),
  address_type    varchar(35),
  primary key (address_no)
);

create table brokerage_currency
(
  brokerage_no    varchar(8) not null,
  currency_code   varchar(8) not null,
  primary key (brokerage_no, currency_code)
);

create table broker
(

```

```

    broker_id      integer not null,
    broker_name    varchar(80),
    broker_phone_no varchar(80),
    primary key (broker_id)
);

create table brokerage
(
    brokerage_no    varchar(8) not null,
    brokerage_name  varchar(80) not null,
    main_phone_no   varchar(20),
    open_date       date,
    address_no      integer not null,
    primary key (brokerage_no)
);

create table client
(
    client_id       integer not null,
    first_name      varchar(80) not null,
    middle_name     varchar(80) not null,
    last_name       varchar(80) not null,
    name_prefix     varchar(80),
    name_suffix     varchar(80),
    nickname        varchar(80),
    legal_name      varchar(80),
    home_phone      varchar(20),
    work_phone      varchar(20),
    open_date       date,
    brokerage_no    varchar(8) not null,
    home_address_no integer not null,
    bill_address_no integer not null,
    primary key (client_id)
);

create table client_detail
(
    client_id       integer not null,
    sex             varchar(1),
    birth_date      date,
    primary key (client_id)
);

create table currency
(
    currency_code   varchar(8) not null,
    currency_name   varchar(80),
    short_name      varchar(35),
    primary key (currency_code)
);

-- =====
-- add foreign keys
-- =====
alter table account add
    foreign key (currency_code)
        references currency;
alter table account add
    foreign key (broker_id)
        references broker;
alter table account add
    foreign key (brokerage_no)
        references brokerage;

alter table account_txn add
    foreign key (account_no)
        references account;

alter table brokerage_currency add
    foreign key (currency_code)
        references currency;
alter table brokerage_currency add

```

```

        foreign key (brokerage_no)
            references brokerage;

alter table brokerage add
    foreign key (address_no)
        references address;

alter table client add
    foreign key (bill_address_no)
        references address;
alter table client add
    foreign key (home_address_no)
        references address;
alter table client add
    foreign key (brokerage_no)
        references brokerage;

alter table client_detail add
    foreign key (client_id)
        references client;

```

9.3 Federated common database DDL

```

-- =====
-- set up relational connect mapping to oracle server
-- =====
create server m%_oracleuser%
    type oracle
    version 8.1.7
    wrapper net8
    options (node '%_oracleinstance%');

create user mapping for %_db2user%
    server m%_oracleuser%
    options (remote_authid '%_oracleuser%',
            remote_password '%_oraclepasswd%');

create nickname oaccount for m%_oracleuser%. %_oracleuser%. account;
create nickname obrokerage for m%_oracleuser%. %_oracleuser%. brokerage;
create nickname ocurrency for m%_oracleuser%. %_oracleuser%. currency;
create nickname oclient for m%_oracleuser%. %_oracleuser%. client;
create nickname oaccount_txn for m%_oracleuser%. %_oracleuser%. account_txn;
create nickname oclient_detail for m%_oracleuser%. %_oracleuser%. client_detail;
create nickname oaddress for m%_oracleuser%. %_oracleuser%. address;
create nickname obrokerage_currency for
m%_oracleuser%. %_oracleuser%. brokerage_currency;
create nickname obroker for m%_oracleuser%. %_oracleuser%. broker;

-- =====
-- create leg_customer_xref (currently the only table)
-- =====
create table leg_customer_xref
(
    customer_id    varchar(8) not null,
    leg_customer_id varchar(8) not null,
    domain_id      varchar(35) not null,
    primary key (customer_id, domain_id)
);

-- =====
-- create account view
-- =====
create federated view account as
select
    account_no,
    '%_db2schema%' domain_id,

```

```

        case upper(account_type)
            when 'C' then 'CMA'
            when 'S' then 'SDA'
        else upper(account_type)
        end account_type,
        customer_no leg_customer_id,
        open_date,
        balance_amt,
        'Avail' balance_type,
        currency_code,
        varchar(char(branch_no)) sp_no,
        varchar(char(branch_no)) sp_contact_id
from %_db2schema%. account
union all
select
    account_no,
    '%_oracleuser%' domain_id,
    case upper(account_type)
        when 'CASH' then 'CMA'
        when 'MMKT' then 'MMA'
    else upper(account_type)
    end account_type,
    varchar(char(integer(client_id))) leg_customer_id,
    date(open_date) open_date,
    balance_amt,
    'Avail' balance_type,
    currency_code,
    brokerage_no sp_no,
    varchar(char(integer(broker_id))) sp_contact_id
from oaccount;

-- =====
-- create service_provider view
-- =====
create federated view service_provider as
select
    varchar(char(branch_no)) sp_no,
    '%_db2schema%' domain_id,
    branch_name sp_name,
    main_phone_no,
    open_date,
    bank_id sp_group_id,
    address_no
from %_db2schema%. branch
union all
select
    brokerage_no sp_no,
    '%_oracleuser%' domain_id,
    brokerage_name sp_name,
    main_phone_no,
    date(open_date) open_date,
    brokerage_no sp_group_id,
    address_no
from obrokerage;

-- =====
-- create sp_group view
-- =====
create federated view sp_group as
select
    bank_id sp_group_id,
    '%_db2schema%' domain_id,
    bank_name sp_group_name
from %_db2schema%. bank
union all
select
    brokerage_no sp_group_id,
    '%_oracleuser%' domain_id,
    brokerage_name sp_group_name
from obrokerage;

-- =====

```

```

-- create currency view
-- =====
create federated view currency as
select
    currency_code,
    '%_db2schema%' domain_id,
    currency_name,
    short_name
from %_db2schema%. currency
union all
select
    currency_code,
    '%_oracleuser%' domain_id,
    currency_name,
    short_name
from ocurrency;

-- =====
-- create sp_currency view
-- =====
create federated view sp_currency as
select
    varchar(char(integer(branch_no))) sp_no,
    '%_db2schema%' domain_id,
    currency_code
from %_db2schema%. branch_currency
union all
select
    brokerage_no sp_no,
    '%_oracleuser%' domain_id,
    currency_code
from obrokerage_currency;

-- =====
-- create customer view
-- =====
create federated view customer as
select
    customer_no leg_customer_id,
    '%_db2schema%' domain_id,
    first_name,
    middle_name,
    last_name,
    name_prefix,
    name_suffix,
    nickname,
    legal_name,
    home_phone home_phone_no,
    work_phone work_phone_no,
    open_date,
    varchar(char(branch_no)) sp_no,
    home_address_no,
    bill_address_no
from %_db2schema%. customer
union all
select
    varchar(char(integer(client_id))) leg_customer_id,
    '%_oracleuser%' domain_id,
    first_name,
    middle_name,
    last_name,
    name_prefix,
    name_suffix,
    nickname,
    legal_name,
    home_phone home_phone_no,
    work_phone work_phone_no,
    date(open_date) open_date,
    brokerage_no sp_no,
    home_address_no,
    bill_address_no
from oclient;

```

```

-- =====
-- create account_txn view
-- =====
create federated view account_txn as
select
    trans_no,
    '%_db2schema%' domain_id,
    case upper(rtrim(trans_type))
        when 'C' then 'CREDIT'
        when 'D' then 'DEBIT'
    else upper(trans_type)
    end trans_type,
    account_no,
    check_no,
    amount,
    trans_datetime
from %_db2schema%. account_txn
union all
select
    trans_no,
    '%_db2schema%' domain_id,
    case upper(trans_type)
        when 'DEP' then 'CREDIT'
        when 'WITH' then 'DEBIT'
    else upper(trans_type)
    end trans_type,
    account_no,
    check_no,
    amount,
    timestamp(trans_datetime) trans_datetime
from oaccount_txn;

-- =====
-- create customer_profile view
-- =====
create federated view customer_profile as
select
    customer_no leg_customer_id,
    '%_db2schema%' domain_id,
    sex gender_code,
    birth_date birth_date
from %_db2schema%. customer_profile
union all
select
    varchar(char(integer(client_id))) leg_customer_id,
    '%_oracleuser%' domain_id,
    sex gender_code,
    date(birth_date) birth_date
from oclient_detail;

-- =====
-- create address view
-- =====
create federated view address as
select
    address_no,
    '%_db2schema%' domain_id,
    'customer' address_type,
    street1_name,
    street2_name,
    city_name,
    state_code,
    country_name,
    zip_code
from %_db2schema%. customer_address
union all
select
    address_no,
    '%_db2schema%' domain_id,
    'bank' address_type,
    street1_name,

```

```

        street2_name,
        city_name,
        state_code,
        country_name,
        zip_code
from %_db2schema%. branch_address
union all
select
        address_no,
        '%_oracleuser%' domain_id,
        address_type,
        street1_name,
        street2_name,
        city_name,
        state_code,
        country_name,
        zip_code
from oaddress;

-- =====
-- create sp_contact view
-- =====
create federated view sp_contact as
select
        branch_no sp_contact_id,
        '%_db2schema%' domain_id,
        manager_name contact_name,
        main_phone_no contact_phone_no
from %_db2schema%. branch
union all
select
        broker_id sp_contact_id,
        '%_oracleuser%' domain_id,
        broker_name contact_name,
        broker_phone_no contact_phone_no
from obroker;

```

10 References

- [1] Bonnet, Olivier et al., “My Mother Thinks I’m a DBA! Cross-Platform, Multi-vendor, Distributed Relational Data Replication With IBM DB2 Data Propagator and IBM DataJoiner Made Easy!” IBM Redbooks, 1999.
- [2] Cook, Melissa A. “Building Enterprise Information Architectures.” New Jersey: Hewlett Packard Company, 1996. 10-19.
- [3] Harkey, Dan, Robert Orfali. “Client/Server Programming with Java and CORBA.” 2nd ed. New York: John Wiley & Sons, inc., 1998. ch 34.
- [4] IBM International Technical Support Organization (ITSO). “Datajoiner Implementation and Usage Guide.” IBM Redbooks, 1995.
- [5] Loosley, Chris, Frank Douglas. “High-Performance Client/Server: A Guide to Building and Managing Robust Distributed Systems.” New York: John Wiley & Sons, Inc., 1998. ch 10, 11, 20.
- [6] Martin, James. “Principles of Database Management.” New Jersey: Prentice-Hall, Inc., 1976. ch 12.

[7] Von Halle, Barbara, David Kudd, eds. "Handbook of Data Management." Boston: Warren Gorham Lamont, 1993. 59-64.

[8] Cristensen, Erik et al. "Web Services Description Language (WSDL) 1.1." W3C Note 15 March 2001. <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.

[9] Box, Don et al. "Simple Object Access Protocol (SOAP) 1.1." W3C Note 08 May 2000. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>.

11 Notices and trademarks

This document may refer to products that are announced but currently unavailable in your country. This document may also refer to products that have not yet been announced in your country. IBM does not make a commitment to make available any unannounced products referred to herein. The final decision to announce a product is based on IBM's business and technical judgment.

Every effort has been made to present a fair assessment of the product families discussed in this paper. The opinions and recommendations expressed in this paper are those of the authors, not necessarily those of IBM.

AIX, DB2, DB2 Universal Database, IBM, and WebSphere are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

IBM Informix and Informix Dynamic Server are trademarks of Informix Corporation or its affiliates, one or more of which may be registered in the U.S. or other jurisdictions.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft and Windows are trademarks of Microsoft Corporation in the United States, other countries, or both.

Other product names are trademarks or registered trademarks of their respective owners.

12 About the authors

Steve Rutledge is currently a software engineer with the IBM Software Group Center for e-business Solution Integration. Steve has been involved in the architecture and design of numerous IT database systems over the last 12 years. You can reach him at sterutle@us.ibm.com.

John Medicke is a Senior Consulting IT Architect working in Raleigh, North Carolina. As the lead architect for the Software Group Center for e-business Solution Integration, John has been involved in design of broad range of e-business solutions including e-commerce, supply chain, and CRM solutions in multiple industries including industrial, distribution, and healthcare. You can reach him at medicke@us.ibm.com.