VLDB Journal, 4, 287-317 (1995), Nathan Goodman, Editor

InterViso: Dealing With the Complexity of Federated Database Access

Marjorie Templeton, Herbert Henley, Edward Maros, and Darrel J. Van Buer

Received April, 1993; revised version received, April, 1994; June, 1994.

Abstract. Connectivity products are finally available to provide the "highways" between computers containing data. IBM has provided strong validation of the concept with their "Information Warehouse." DBMS vendors are providing gateways into their products, and SQL is being retrofitted on many older DBMSs to make it easier to access data from standard 4GL products and application development systems. The next step needed for data integration is to provide (1) a common data dictionary with a conceptual schema across the data to mask the many differences that occur when databases are developed independently and (2) a server that can access and integrate the databases using information from the data dictionary. In this article, we discuss InterViso, one of the first commercial federated database products. InterViso is based on Mermaid, which was developed at SDC and Unisys (Templeton et al., 1987*b*). It provides a value added layer above connectivity products to handle views across databases, schema translation, and transaction management.

Key Words. Federated database, database integration, data warehouse.

1. Introduction

InterViso is a DBMS front end that allows a user to access data that are managed by existing DBMSs. The individual databases may have been independently designed and may be managed by different DBMSs. InterViso provides a standardized view across all of the databases, which masks the structural differences and provides location transparency. With InterViso, existing applications can continue to access data as they currently exist, and the new applications can gather and integrate data

Marjorie Templeton, B.A., is Vice President of Technology, *margie@dii.com;* Herbert Henley, B.S., is Senior Technical Staffmember, *herb@dii.com;* Edward Maros, B.S, is Technical Staffmember *edward@dii.com;* and Darrel J. Van Buer, Ph.D., is Senior Technologist, *darrel@dii.com;* Data Integration, Inc., 11965 Venice Blvd., Suite 305, Los Angeles, CA 90066.

from the existing databases, as required, to meet new and changing goals of an organization.

The user or application programmer sees the enterprise data as one single database and initiates a single query or update. InterViso determines how to answer that query or perform the update by breaking it down into parallel components. InterViso automatically locates the data, opens connections to the existing computers and databases (using whatever connectivity products are available), issues queries to the DBMSs in their query language (SQL or other), and integrates the returned data from the multiple sources. The integration may require translation and resolution of data types, fields and values, all of which is done automatically by InterViso. The user receives the answer to the single query in an integrated form.

InterViso is the culmination of many years of research at System Development Corporation (SDC) which was merged into Unisys. The prototype at SDC was called Mermaid (Templeton et al., 1987*b*; Thomas et al., 1990). The design of Mermaid was the result of an analysis, done in 1982, of Department of Defense (DoD) requirements for the next generation of data management systems. The requirements have been refined through the years, although the basic needs still exist. InterViso has been applied commercially, which has validated the basic requirements and demonstrated that commercial requirements are similar to DoD requirements, although commercial need may lag by a few years.

Each research prototype in the general area of federated databases has tended to emphasize different aspects of the general federated database problem (Elmagarmid and Du, 1990; Litwin et al., 1990; Ahmed et al., 1991; Perrizo et al., 1991; Breitbart et al., 1992*a*; Veijalainen and Wolski, 1992). InterViso has bridged the gap between theory and practice. Many pragmatic decisions have been made to satisfy conflicting requirements.

The basic design criteria for Mermaid and InterViso are:

1. Conform to industry standards: Mermaid and InterViso have tended to be implemented ahead of open systems standards, which means frequent modification. InterViso has a modular design to support replacement of parts as the standards change.
2. Local autonomy of existing databases: The types of autonomy include design autonomy, execution autonomy, and communication autonomy (Breitbart et al., 1992*b*).
3. Requirement for high availability: InterViso will continue to run in spite of node failures.
4. Minimal impact on the local system by external users: InterViso users have no priority and cannot hold locks that will lock out local users.
5. Tight access control: Databases are locally owned and permission is granted to individual users of InterViso.
6. Need for data and schema translation: Independent designs of the databases mean that names and data types for the same data elements may not be the same in different databases.

7. Replicated and fragmented relations: Similar data and similar schema may exist for different data in independent databases.
8. Primary use for decision support: It is expected that most users will be submitting ad hoc queries. Some organizations do not want users to submit any updates, but other users require update capability.
9. Distributed control: Users may run on many different servers across a large network.

The query portion of InterViso, called "IVQuery," is a direct descendent of Mermaid but with added capability. The differences between Mermaid and InterViso are in the supporting utility programs, the level of documentation, and the level of testing. Mermaid was a research prototype and lacked the utilities and documentation necessary for a commercial product. The data dictionary (DD/D) that contains the federated view across the existing databases had to be built by the Mermaid staff rather than by the end user organization. InterViso includes an interactive DD/D development program that comes with extensive documentation to guide a developer through the process. This program ensures internal consistency of the dictionary and includes tools to simplify repetitive tasks. It also includes an interactive program to maintain access lists and passwords, a program to copy data dictionaries between hosts and DBMSs, and a program to recover from update failures.

Some of the capabilities that were added to InterViso are: updates to underlying databases, more types of schema translation, improved error handling, extensive logging of activity and errors, and an application program interface that conforms to the Microsoft ODBC standard for call level interfaces.

Mermaid was an early prototype of a "federated database" system. Now, 12 years later, the commercial market is beginning to recognize the need for federated databases. However, there is still confusion between connectivity products, data warehouse products, and federated database products.

Many vendors have developed gateways from PCs and Unix servers to legacy DBMSs on mainframes. The leading gateway vendor is Information Builders with the EDA/SQL product. It offers access to 55 DBMSs on many types of platforms from many types of platforms. All RDBMS vendors and most application development tool vendors also offer gateways.

Gateway products connect the client side, where the application program is running, to the server side, where the data resides. The gateway offers language translation, data model translation (from the relational user view to the way that the DBMS stores the data), transparent log-in to the server host and DBMS, and some name mapping to rename data elements.

Gateway products do not offer a view across databases. They do not handle the transformation from a standardized, federated view to the way that data are stored. They do not manage the transformations necessary to combine data from different databases, and they do not manage replicated or fragmented table updates. Using an application development tool directly with the gateway product requires

that these capabilities be coded into each program. InterViso therefore offers an important component between the gateway and the application program.

InterViso is an integration product rather than a connectivity product. It includes gateways to relational DBMS products on Unix computers such as Oracle, Sybase, and Ingres, and language translators to other DBMSs using SQL such as DB2 and Informix. InterViso also works with gateway products from many vendors to provide the connection to non-Unix computers and non-relational databases.

Data warehouse products download and transform data from legacy databases to a relational DBMS. Transformed snapshots of the legacy databases are taken periodically. Retrieval for decision support is done against the snapshots in the relational DBMS. Updates continue to be made to the legacy databases through the existing application programs. The data warehouse solution requires that all data needed by the decision support application be stored in the warehouse. There is no way to reach through the warehouse to the underlying databases. This means that large amounts of data may have to be stored in the warehouse or that data must be summarized, which means losing the detail.

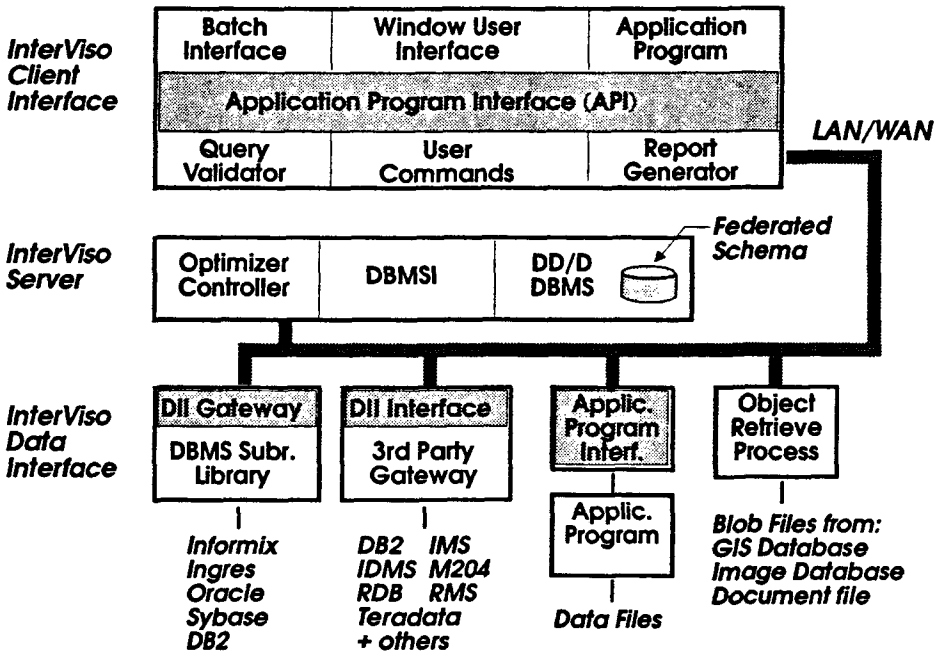
InterViso goes beyond the capabilities of a data warehouse, because it allows dynamic access to the changing database in conjunction with access to snapshots. Only the frequently accessed and seldom updated data need to be moved to the warehouse. Data may be collected for the warehouse in a batch job, as warehouse products do, or with a dynamic interactive query. Regularly used warehouse snapshots can be taken periodically, as with warehouse products. In addition, interactive users may dynamically develop snapshots that cover data of immediate interest. InterViso then supports joins between the warehoused data and the underlying data sources.

In the following sections, we discuss the implementation of InterViso. In Section 2, we present the system architecture. Section 3 discusses the development of the DD/D. Successful operation of InterViso depends upon complete descriptions of the existing databases and the environment. Section 4 describes the InterViso server, which does query optimization and transaction management across databases. Section 5 reviews the issues in connecting to existing DBMS products and existing databases. Section 6 discusses access control. Section 7 discusses the interaction with standards, and Section 8 discusses lessons learned in turning a prototype into a product. Conclusions are made in Section 9.

2. InterViso Architecture

InterViso runs most processes and all database access, using the identity of the user so that all system access controls will continue to operate. Each user has a user interface process that controls either an interactive interface or application program; one "controller" that performs query optimization and transaction control; one process that reads the DD/D; and one Database Management System Interface (DBMSI)

Figure 1. InterViso Architecture



Each box with a double line may reside on a different computer.

for each active database. The only shared InterViso process is a communications daemon on each host. Figure 1 shows the InterViso system architecture.

InterViso may have several types of user interfaces. Two Application Program Interfaces (API) are supplied. The Mermaid system had no API that was available to programmers. InterViso originally incorporated an API based on the Sybase API. In 1994, a second API was developed that conforms to the Microsoft ODBC standard, which is based on a draft of the SQL Access Group draft CLI (Call Level Interface; SQL Access Group, 1992; Microsoft, 1993).

Data Integration, Inc. (DII) supplies three interfaces that are "applications" built on the API: an X windows interactive interface, an ASCII terminal interactive interface, and a batch job stream interface. Programmers can develop their own interactive or batch interfaces for specific applications using the C language or any application development toolkit that can call ODBC compliant data sources.

The controller provides query optimization, access control, process control, and update replication control. Details will be supplied in several sections below.

One or more copies of the DD/D exist on the network at a node which runs Ingres, Oracle, or Sybase. The DD/D is a large database that needs to be managed reliably and with high performance, so InterViso uses a commercial relational DBMS

to manage the DD/D. Development of the DD/D is key to the successful operation of InterViso (Section 3).

Each DBMSI runs on the computer with the database or on the computer hosting the database gateway to the data to be accessed. The DBMSI performs DBMS-specific translation of queries, makes DBMS specific calls, and does local transaction management (Section 5). It also handles database specific schema translation. The relational DBMSs on Unix systems are accessed through DII-developed gateways. Other DBMSs are accessed through third party gateways.

Each set of InterViso processes for one-user runs with no knowledge of other InterViso users that may be running. There could be many different copies of the DD/D and many different computers where user interface and controller programs are running. Each controller process handles transaction management for one user only. The processes for different users could be on different nodes of a wide area network. Transaction management between users is left to the individual local DBMS, however InterViso does use timeouts to resolve potential deadlocks.

3. Data Dictionary Development

Before InterViso can run, it needs a definition of the "local schemas" for the existing databases, and the "federated schema," defining a view across the databases. These definitions and the mappings between them are stored in the DD/D database. Many of the types of problems that must be addressed when developing a federated schema have been discussed (Templeton, 1989; Collet et al., 1991; Kim and Seo, 1991; Sheth and Larson, 1991). These problems include naming differences, both different names for like data and unrelated data under the same name, missing data, extra data, different precision, different representation, and different grouping of similar data, both at the table and field level. In addition, the data for a single federated table may be spread across several sites, possibly with overlapping copies or mixed in with unrelated data.

Some of the development process can be automated, but many decisions must be made by a human designer who understands the content of the existing databases, the standards that a specific organization has for databases, the requirements for access control, and the application views of the data. This may be a major task, but it is necessary for control over existing databases. Once the existing database elements are cataloged in InterViso's DD/D, all programmers and users can refer to the DD/D. Without a central catalog, every application or query developer needs to find out what data are available and write the transformations that are needed to view the data in a common report.

The DD/D Builder program (called "IVBuild"), which is used to develop and maintain the DD/D, is a significant difference between Mermaid and InterViso. Before DII developed this program, building the DD/D required knowledge of the DD/D structure. IVBuild manages internal details and provides a number of tools for aiding the development process.

IVBuild differs from other data modeling tools in being primarily bottom up; thus the description of the individual databases is *input* rather than *output*. The DataBase Administrator starts by describing the schemas of the existing databases, often by reading in existing data definition language that describes them and then develops a federated schema or schemas that encompass all or parts of the local schemas. One federated schema is developed for each application or group of users who share a common view of the databases and who have similar access permission to view the local databases.

IVBuild is an editor/browser for DD/Ds. Much of its user interface resembles an X-windows 4GL providing reports and edit windows for many of the basic components of a DD/D. It also incorporates a number of features designed to speed up many of the steps in building an application. To capture existing data definition language, IVBuild can read a number of different formats defining existing databases. After loading the data definition language, the DataBase Administrator uses a series of data entry windows to add information about the contents of each local database. This step requires the DataBase Administrator to understand the contents of the current databases and to formulate the appropriate semantic labeling of the contents. Building a federated definition identical to one local site can be done in a single operation, though some follow-up editing is usually required to make it follow design rules and reach a form compatible with the other local databases. Other tools help speed the connection to the remaining sites by identifying possible matches between federated and local entities, and by storing idioms which are likely to recur later in the editing process. IVBuild can also produce a number of reports on the status of the process showing what has been done or remains to be done. Finally, IVBuild contains considerable checking and repair of the internal consistency of a DD/D, was well as logging and recovery of DD/D updates to minimize the problems caused by system failure during the development of an application.

Each DD/D consists of the directory and one database per federated schema. The "directory" contains global and administrative information about the users, the hosts, the network, the capabilities of each DBMS where the databases are stored, and a catalog of federated schemas. The directory information is independent of a specific federated schema, and is used to connect to the hosts and DBMSs.

Rather than using one dictionary to store *all* federated schemas and mappings, Mermaid and InterViso use a separate data dictionary for each independent federated schema. Multiple federated schemas may exist across the same underlying "local" databases to provide different views. A "local" database is an existing database where some or all of the data will be shared with users of InterViso. To a user of InterViso, the federated schema will appear to be a database with a schema, but all of the data are actually managed by the several local DBMSs. Many different federated databases can be built on different subsets of the same local databases. The different federated databases provide a way to meet the needs of different applications and user communities and control or limit access to data as needed.

Figure 2. Two local schemas

```
Local relation in database "SATVIEW"
   TABLE satrpt(
      shiptype CHAR(15), /* code assigned by photo interpreters*/
      lat      CHAR(7),  /* latitude DDDMMM +direction */
      lng      CHAR(7),  /* longitude DDDMMM +direction */
      time     INT)      /* month/day/hour/minute */

Local relation in database "SHIPRPT"
   TABLE shiprpt(
      shiptype  CHAR(15), /* code of type seen */
      shipname  CHAR(20), /* name of ship */
      country   INT,      /* country code */
      lat       CHAR(7),  /* latitude DDDMMM +direction */
      lng       CHAR(7),  /* longitude DDDMMM +direction */
      month     CHAR(4),  /* month name abbreviation */
      day       INT,      /* day */
      year      INT)      /* year */
```

For example, some data may be seen by intelligence analysts using names and types that are familiar to analysts. Another federated schema could be defined for the Navy headquarters staff with a different view of the data.

The next section shows an example of two local database relations and a possible federated view of the relations. This is followed by a more complete discussion of the possible field and relation mappings.

3.1 Example

To illustrate some of the decisions that must be made when defining a federated schema, assume that there are two relations in two different databases containing the sightings of ships. One database contains satellite reports and the other contains visual reports. The same ships are being sighted, but the content of the reports is different. The satellite reports have more precision in the report time, have more sightings, and cover more ships, but the visual sighting reports collect more information about the individual ships. The two local relations are shown in Figure 2.

There is no single "correct" federated schema. Several different federated

Figure 3. Federated database SHIPS

```
/* fragmented, map to satrpt and shiprpt */
      TABLE sighting(
      shiptype CHAR(15), /* connect to satrpt.shiptype
                            or shiprpt.shiptype */
      lat      CHAR(7),  /* latitude DDDMMM +direction */
      lng      CHAR(7),  /* longitude DDDMMM +direction */
      date     CHAR(5))  /* MM-DD */

/* single site map to satrpt */
      TABLE sat_rpt(
      shiptype CHAR(15), /* code assigned by photo interpreter*/
      lat      CHAR(7),  /* latitude DDDMMM +direction */
      lng      CHAR(7),  /* longitude DDDMMM +direction */
      date     CHAR(11)  /* MM-DD-HH-MM */

/* single site map to shiprpt */
      TABLE ship_rpt(
      shiptype   CHAR(15), /* code of type seen */
      shipname   CHAR(20), /* name of ship */
      country    CHAR(30), /* country name */
      lat        CHAR(7),  /* latitude DDDMMM +direction */
      lng        CHAR(7),  /* longitude DDDMMM +direction */
      date       CHAR(5),  /* MM-DD */
      year       CHAR(4))  /* year */
```

schemas could be developed for different sets of users and/or applications. An example of one possible federated schema is shown in Figure 3.

Figure 3 shows one fragmented relation, "sighting," to view all of the sightings as a single relation. The relation would contain the fields in common in a standard form and would use the precision in days, the lowest precision database. The relation is fragmented because each database contains different information. Some ships may be sighted about the same time by both another ship and a satellite, but the user must get data from both databases to see all of the sighting data.

Two other federated relations could be defined to view the individual types of sightings. Some conversion is done on the output to put it into a more standard form. The country code is converted to a name. The dates are converted into a character form with dashes between the components, and months expressed as a number rather than a name.

The sat_rpt relation maps directly to the satrpt relation in the SATVIEW database. The ship_rpt relation maps to the shiprpt relation in the SHIPRPT database. The sighting relation maps to the union of the satrpt and shiprpt relations, although not all fields are included in the federated view.

3.2 Field Definition

3.2.1 Domain and Units. The first step in developing a DD/D is to understand the meaning of every data element. Figure 2 shows comment meaning, which is turned into a more formal labeling with the domain and the unit. If an organization has defined standards for data element representation, data element standards are also collected and attached to the data elements. The data element standard influences the definition of the federated view of a field.

The field name, data type, and length describe the data element storage form but do little to describe the contents or its semantics. The *domain* is the semantic meaning of the data element (e.g., time). One local field containing time data may be called "time" and another may be called "HourMin," but both contain the same type of data, so they both belong to the same or related domains. One database may have the time stored in a single field, while another stores the date, hour, and minutes in different fields. One database may store time in minutes, while another has it in days (Figure 2).

The data element standard might specify that all fields with date and time data be called "date" and have the data represented as "MM-DD-HH-MM." This would be the preferred representation in the federated schema.

The unit of a field describes the way that quantities are represented, such as inches or miles for distance, and pounds or kilograms for weight. If a field in one local database contains the distance between ports in miles, and the corresponding field in another local database contains the distance between ports in kilometers, these fields would share the common domain "distance," but use different units. At the federated level, fields must use the same domain and unit if they will be joined or compared. When data element standards are used, the standard for the distance domain specifies which unit should be seen in the federated schema, and both local fields are converted to the unit of the federated schema as needed.

Units often apply to numeric quantities, but InterViso extends the unit concept to any data with multiple representations for the same value. For example, the unit label may be used to describe alphabetic formats such as two versions of a date in strings in the forms "YYMMDD," or a Julian date "YY-DDD." Fields of these types could store data from the sighting-date domain using the same precision. In general, when two data fields have the same domain but use different units, it means that some data transformation is needed.

3.2.2 Field Mappings. The federated relation (the federated view of the local relations) is composed of federated fields which draw data values from one or more

fields in one or more corresponding local relations. These field mappings may consist of combinations of the following:

- rename the field;
- change the length of the field;
- convert the data type;
- convert the case of a character field from lower to upper, upper to lower, or mixed to upper or lower;
- specify field composition of a federated field by concatenating and/or subsetting one or more local fields;
- perform arithmetic operations on one or more local fields;
- perform character pattern operations on one or more local fields;
- translate the values of a character or integer field using a table lookup to get the new value;
- return a constant which may have a value dependent upon which local database and relation is accessed;
- translate the values of a field using a function written in C programming language. This case is seldom needed, but provides for almost any computable transformation.

All of the field translations must be defined for both directions, federated to local and local to federated, because values from the database are translated for reporting and values in the query are translated for qualification or update. The transformations are stored in the DD/D.

3.3 Relation Definition

3.3.1 Relation Mapping. Each federated relation is a view of one or more local relations. One federated relation may map

- to one local relation with the same or a modified schema;
- to a selection of records in a local relation to eliminate some records from the federated schema (a "horizontal partition");
- to a projection of fields of one local relation to eliminate some fields from the federated relation;
- to two or more local relations that must be joined to create the federated relation (a "vertical partition");
- to two or more local relations in different databases that must be unioned to present all of the records ("fragmentation");
- to two or more local relations in different databases that contain the same records ("replications") but which may have schema differences;
- to relations combining the above cases.

3.3.2 Replicated and Fragmented Relations. Management of replicated and fragmented relations causes most of the complexity in the development of the DD/D and the query optimizer and the transaction manager.

Replicated relations have fields of the same domain, and records that are the same in all copies. They exist when there are redundant data in the local databases. *Fragmented relations* have fields of the same domain, but their records are disjoint subsets. They exist when different organizations collect the same information about different entities such as ships in different fleets. It is not possible to determine whether relations are fragmented or replicated by looking at the field names, data types, or domains; the determination depends solely upon the data.

Data are not always cleanly replicated or fragmented. Data may be collected in one location and then copied to another periodically so that the original site is more current. Data may be collected in several locations and then be processed before being combined centrally. Since replication and fragmentation are logical concepts in the federated schema, it is possible to view the data differently for different users or applications. If the application that will use the federated schema needs the most recent data at all times and updates do not occur through InterViso, then perhaps only the most recent copy should be mapped to the federated relation, and other copies should be excluded. If the application needs the most recent at some times but should update both copies if an update is made, then the relations should be declared "replicated" with the most recent one being declared the primary copy. The user of InterViso can specify in a query that the primary copy is to be retrieved if that is important.

There may be a case where two relations in different databases share most of the same fields and do have the same information in the common fields, but have other fields peculiar to a particular database. This may require the DataBase Administrator to define more than one federated table view of the data—one that incorporates only the lowest common denominator of the data elements, which is usable everywhere, and others which give access to data with limited availability. For example, assume that there are two databases where database 1 contains R1 with fields {A,B,C,E} and database 2 contains R2 with fields {A,C,D,E}. The DataBase Administrator can declare that the common fields belong to a replicated relation and that the other fields belong to single-location federated relations, or that the other fields are not part of the federated schema. Assuming that A is the key, three federated relations might be defined: Rrepl {A,C,E}, R1single {A,B}, and R2single {A,D}. The DataBase Adminstrator might also decide that field D does not need to be seen in the federated view, and omit R2single.

There also may be cases where some of the records are replicated and others are not. For example, one database may have military and commercial ships in the same relation, while another database has only commercial ships. Two federated relations might be defined as "MilShip" for a single site, and "ComShip" for a replicated relation. The mapping for MilShip and ComShip at the site where information on both types of ship is stored in the same relation would map to a horizontal partition

of the relation by defining a predicate to select only one type of ship at a time for the federated view.

Fragmented relations are similar to replicated relations except that each record is stored in only one copy. If some of the fields are not included in all copies of the fragments, the remaining fields can be seen through a different federated relation or be eliminated from the federated schema. Some records may also be eliminated as some sites to make the data consistent. For example, if the SATVIEW database in Figure 2 contains reports for 3 months and the SHIPRPT contains a 12-month history, then the DataBase Administrator may define a predicate to eliminate the older reports from the second database. The federated relation "sighting" would then retrieve a three-month history from both databases. The single site federated relation `ship_rpt` could be used to retrieve the entire history. From the logical, federated schema view, these are two different relations, although both retrieve data from the same local database relation.

Fragmented relations may have an explicit or implicit predicate for location. This predicate is an equality or range check on the value of some field that will determine the location of a record. For example, if each fleet maintains its own ship location database, the fleet number would determine the local database. The example in Figure 2 does not have an explicit predicate, but the presence of a record in a database carries the information that the report was collected by satellite or by visual sighting. A virtual field `Stype` could be added to the federated relation with a constant value `SAT` or `VISUAL`. If the user enters:

```
INSERT INTO sighting (shiptype, lat, lng, date, Stype)
    VALUES ('CG17', '045010N', '050000E', '0407', 'VISUAL')
```

then the value `VISUAL` will specify into which database the record will be inserted.

Several fragmented relations may be located in a specific database as a *fragment group.* For example, each database may have several relations containing information about ships. All information on any given ship is in the same database. Fragment groups have a master relation that locates the group. The other relations in the group, called *dependent fragments,* are located with the master relation. In the ship example, the fleet field in the master record determines which database contains the records for the ship. The master record might contain the characteristics of the ship such as the name, data commissioned, flag, etc. Other ship records such as the sighting history may not have a fleet field so their location is dependent upon the ship master. Again, this is a logical concept when developing the federated schema.

There may be cases where it is not clear whether relations are fragmented or replicated. For example, some records may be in multiple databases but not in all databases. Updating then becomes extremely difficult. The DataBase Administrator must define predicates that can generate a cleanly replicated or fragmented relation. If this is not possible, it may be necessary to treat each local relation as an independent federated relation, or else to select some subset of the local relations to include in

the federated schema. The DD/D can also specify that updates are not allowed on the federated relation.

3.3.3 Vertical Partition Joins. In some cases, two or more relations in one local database contain the same information that another local database stores in a single relation. For example, one database may have two ship records per ship, one containing the ship characteristics information and the other administrative information while a second database stores all ship information in one record. There are two alternatives for defining federated relations. One federated relation could map to the join of the two relations in one database and directly to the local relation in the other database. Alternately, two federated relations could map to the two relations in one database and map to two projections of the single local relation in the other database. At a minimum, the key field must be mapped to both federated relations.

3.4 Integrity

InterViso is designed primarily for retrieval rather than for database maintenance. Therefore, little has been done with integrity constraints. Each federated field is labeled with rules:

- *Insert rule:* field is required, optional (may be NULL), or excluded (used when there are multiple views of the same field).
- *Update rule:* update allowed or forbidden. It must be forbidden if the underlying DBMS forbids update. This flag is also set on by IVBuild for fields that cannot be updated because (1) the field locates the record and a change would involve moving the record to a different database, or (2) the field maps to only part of an underlying field and InterViso cannot update part of a field.
- *Qualification rule:* the field cannot be used in qualification due to limitations in the data source. This is used for non-relational data sources.

Values may be checked on update by adding a function or translate table to the field. InterViso does not keep a list of valid values or ranges for a domain.

3.5 Model Limitations

The InterViso DD/D format provides modeling to handle all of the above problems, but there are limits in both the model itself and the tools in the schema translator component of the DBMSI that implements the model-directed transformations.

When data are missing or of a lower precision, little restoration can be done except when the lost values can be inferred from their location. For example, if each data site stores data for a distinct subset of data, but only some sites explicitly record the subset identity, InterViso can supply the value for the other sites. If data in different databases have different precision, views across databases may have

to have the lowest precision, but other tables may be defined to show the single database view with the full precision as was done in the example above.

There are also limitations when several local fields must be combined to realize a federated value. Translations are needed in both directions, federated to local and local to federated, if the field is to be updated or used in qualification. Some combinations (e.g., addition) lose information so that mapping from federated to local values is impossible. This is, of course, a common problem when providing updates through views.

The model also restricts the definition of a federated field to the contents of a single record at each site. It is not possible to base transformation of records in one relation on the contents of another relation unless the federated relation is defined as a join between those local relations at the same site. Query processing would be very slow if joins of relations had to be done before any data could be qualified, and some joins would not be possible until schema translation occurred. So, for example, it is not possible to define a department budget as the sum of the costs from all of the project records within a department, or as a function of values in two tables which are joined.

When data are fragmented, the model requires each site to contain identifiable and disjoint subsets. Depending on the way data are distributed or overlap between sites, the model's subsetting mechanism may be unable to express the rules defining where the data should be found. The test is restricted to a single comparison or range of values for a single federated field at each site. In many cases, the local values can be mapped to federated values which meet these requirements, but not always. In the example above, the location of the same ship might seen in both databases. If the logical view retrieves all sightings, then a fragmented view makes sense. If the logical view is used to change characteristics of the ship, such as changing the name of the captain, then the view must declare the tables as replicated even though all of the data are not replicated. This causes numerous errors on update. The only solution is to have some indicator in the data that identifies them as redundant so that the non-replicated records can be removed with a horizontal partition, but this is usually not available.

4. Controller

The controller process provides query optimization, process initialization, process restart when a process fails, message retry, and transaction management. Details of the optimizer and transaction manager are discussed in this section.

4.1 Query Optimization

Access to data in multiple databases requires careful planning to minimize the time required to process the query. This is important in local DBMSs, especially when data must be moved across a network. Query optimization algorithms have

been developed for DDBMS such as SDD-1 (Bernstein et al., 1981), distributed INGRES (Epstein et al., 1978), Encompass, and R* (Williams et al., 1981; Lohman et al., 1984). The Mermaid system started with ideas from SDD-1 and distributed INGRES when developing the optimization algorithms that are currently used in InterViso. Mermaid made extensions to the existing distributed DBMS algorithms to support various process and network costs in a heterogeneous environment, and to deal with fragmented relations and potential savings from exploiting replicated relations.

The assumptions behind the optimizer design are discussed here. InterViso must provide a query optimization algorithm that can adjust to a wide variety of data distributions. In some environments, there may be closely coupled databases in which a relation from one database may be replicated in another database, or in which some relations may be fragmented with fragments stored in different databases. In other environments, the databases may be basically disjoint, although there must be some attributes that can be used to join relations across the databases or there would be little reason to treat them as a federation.

It is assumed that processing costs and communication costs will not be uniform. There will be different data management systems which will have different operational characteristics even when running on the same hardware, and there will be different types of computers. The network will also be nonuniform. There may be multiple local networks, possibly different types of networks, connected by gateways. Estimates of the cost factors for the processors and network connections are stored in the DD/D.

The basic processing steps are:

- Flatten the query to remove nested queries and disjunction using the Kim algorithm (Kim, 1982).
- If the query can be processed at one site, send the query to the single site.
- Identify data fragment groups and replicated copies of relations. Fragment groups are fragmented relations that join only to other fragmented relations at the same site. Replicated relations will be used at multiple sites if that will result in reduced data movement during processing, but the query can be processed even if some site is not available, as long as all relations can be accessed at some site.
- Perform local reduction on individual tables or groups of tables at each site in parallel. This may include select, project, and join. The result is stored in federated form and the record count is returned to the optimizer.
- Estimate the cost of moving data together to various sites for a semijoin or union fragment operation, choosing the least costly. Only one fragment group may stay fragmented.
- Estimate the cost of various sites as the destination for bringing all needed data together.
- Union some fragments (some many be unioned before processing and others may be left completely or partially fragmented).

- Select the most beneficial joins or semijoins to perform. Perform joins within and across sites. Perform semijoins across sites. In practice, few queries benefit from semijoins or inter-site joins done before moving data to the assembly site unless the data reduction saves more than one buffer of data transfer.
- After all beneficial joins have been performed, move data to the selected assembly site. Perform final joins and unions.
- Process a final query to combine and select results from the assembled data, returning any results to the user or API.
- Clean up temporary tables.

During these steps, the optimizer partitions and reorganizes the original query into a series of SQL statements which direct the operations at each data site. The optimizer deals with all data in federated format. All data operations except inter-site data movement are performed entirely by the local database engines (see Chen et al., 1986; Templeton et al., 1987*b* for more complete details).

Since it is assumed that users will be using InterViso interactively, minimization of response time is important. The response time includes the time to develop the plan for answering the query as well as the time to execute it. Plans are not compiled and saved because minor changes in the qualification of a query could change the sites that need to be accessed and the distribution of data at the sites, substantially altering the optimum strategy. Also, current plan heuristics are evaluated quickly relative to the costs of most local data operations and network delays, so compiled plans would offer limited benefits at best.

Since relational databases are becoming prevalent and relational interfaces are being developed for many nonrelational DBMSs, InterViso has emphasized access to relational DBMSs. InterViso assumes the existence of a local optimizer that determines which indices to use, how to perform a join, and the order of joins if multiple joins are required for local operations. If the access is through a gateway product, the gateway is expected to contain some optimization to convert from the SQL query to the local query.

A *capabilities table* in the DD/D specifies what operations a DBMS can do at a site. The capabilities include the number of joins that can be done in one query and the availability of `GROUP BY`, aggregates, `LIKE` pattern matches, string operations, `INDEX`, and `DISTINCT`. In some cases, the optimizer will avoid a site that cannot perform an operation. For example, if a join is to be done between two tables at sites that cannot perform a join, the tables must be moved to a third site to perform the join. In other cases, the schema translator will supply the missing functionality. For example, if a site cannot do `LIKE pattern` and the query contains a pattern match, the records will be read into the schema translator, which will determine which records qualify rather than using the DBMS to qualify the records.

One `SELECT` statement at the federated level may involve a series of operations in multiple databases. InterViso attempts to present a consistent read view to the

user even when data are retrieved from multiple databases. We avoid read locks for performance reasons and would not use them even if a lock command were always available. The window of inconsistency is kept small by performing parallel queries in all databases to gather data into temporary relations before doing the joins and unions across databases. However, local transactions or other InterViso transactions could still make changes within this window.

We have found that this read policy is adequate for most applications. In many applications, update schedules of the independent databases vary widely, often hours or days. Users without federated databases must log into each database independently, generally giving a much larger window of inconsistency. InterViso therefore offers a solution that is significantly better than what currently exists. When this is not adequate, other approaches, perhaps global locking, are required. The design of InterViso assumes that few applications can afford the costs and problems of full distributed locking in a networked environment.

4.2 Transaction Processing

The transaction processing strategies designed for centralized DBMS operation cannot be used in a heterogeneous federated database system such as InterViso for several reasons. The key reasons are the different capabilities of the available local DBMSs (i.e., no consistent and accessible test and control of locks); the potential that a site is off-line; the existence of replicated and fragmented relations; and the inability of the federated DBMS to see the local locks held and the other transactions that might be running at the same time (Levy et al., 1991). InterViso has therefore developed a pragmatic approach that gives up serializability for capability.

InterViso's criteria for updates are:

- *Reliability.* No update shall be lost, but updates are not necessarily made concurrently.
- *Performance.* Updates shall have a minimal impact on the performance of the local DBMSs.
- *Availability.* Updates shall be made even if all sites are not available.
- *Consistency.* Replicated and fragmented tables shall be kept consistent across databases as long as the updates come through InterViso.

The InterViso update design assumes that there is no external lock or prepare to commit command available, unlike some algorithms (Pu et al., 1991). Even if there were a lock command, it would often have an unacceptable impact on the performance of local systems, as it would subject them to delays caused by the network and remote systems. When operating in a federated environment with access to existing databases, it is very common to find fragmented and replicated relations which add complexity to updates. Most articles on updates assume that each relation is a single site relation so that the problem is an extension of the distributed DBMS update problem with the added complexity of autonomous DBMSs (Breitbart

et al., 1992*a*, 1992*b*; Mehrotra et al., 1992). However, new approaches are needed to deal with fragmented and replicated relations.

The basic design criteria (Section 1) require high availability and reliability, but not necessarily serializable updates. Retrieval availability and performance can be improved with replicated copies, and redundant data may exist for historical reasons rather than by design. Relations may be fragmented when many organizations or divisions collect information about the same entities, but the individual organizations do not see their data as a fragment of a larger system. There could be a large number of databases in the network, and some may be off-line at any given time. In InterViso, if one site with a part of a fragmented relation or a secondary replicated copy of a relation is not available, the update will be made anyway, and the update is logged for application when access is restored. A policy of strict serializability requires an update failure if one or more sites are not available. However, one site could be off-line indefinitely, which could prevent any updates to a relation. This is generally not acceptable.

InterViso places restrictions on the types of transactions that can be submitted. In particular, only the following classes are supported:

1. One relation, no `BEGIN/END`, single site, replicated, or fragmented relation, implicitly treated as a transaction.
2. Multiple single site relations, enclosed in `BEGIN ... END`, all relations at the same site.
3. Multiple relations enclosed in `BEGIN ... END`, all relations single site or replicated with all primary copies and single site copies at the same site, and the same set of sites for replication.
4. Multiple relations, enclosed in `BEGIN ... END`, all one fragment group with one fragmented relation and associated dependent fragmented relations. The transaction may *not* include a single site or replicated relation.

The reason that a transaction may not mix replicated and fragmented updates is that the rules for commit are not the same.

Replicated relation `UPDATE`, `INSERT`, or `DELETE` transactions are run first at the site of the primary copy. When the primary copy is in a prepare-to-commit state (all parts have run successfully but no `END` has been sent), the transaction will commit and will be logged in the InterViso log file. The transaction, minus single site relations, is then sent to each secondary site and the updates are made as soon as possible. If a site cannot be started or fails during update, updates will continue at the other sites. The user will receive a message that the "iv recover" program must be run when the site becomes available. The recovery program reads the logs for all users for all federated databases and applies updates to a failed site in timestamp order. This is shown in pseudo code in Figure 4.

A fragmented relation `UPDATE` or `DELETE` may have to be sent to every site if there is no fragmentation predicate on the relation, or the qualification does not include the field that determines the site. The `UPDATE` or `DELETE` will be committed

Figure 4. Replicated relation update

```
Send transaction to primary site
    IF any step FAIL
     ABORT
    IF SUCCEED without END sent  (prepared state)
     Log secondary sites
     Send END to primary site
     IF FAIL on END
      ABORT
      Remove logs
     IF SUCCEED (committed state)
      COMMIT
      Send transaction to all sites in parallel
      WHILE work pending
       {
       Read status as each site completes
       IF SUCCEED, not end
        Submit next statement in transaction
       IF SUCCEED on end
        Remove site from pending list and log
       IF FAIL
        Must run recovery
       IF timeout while waiting
        Treat as FAIL at site that timed out
       }
```

as soon as one site updates at least one record. If a site does not return any status, it is presumed to have failed. If a site returns `0 tuples affected`, the site will be removed from the list of pending updates, but the update will not be committed until some nonzero tuple count is received. Before the commit, the transaction will `ABORT` if a site fails. As soon as one site commits, the `UPDATE` or `DELETE` will be logged. If one or more sites fail after the commit, the iv recover program must be run. This is shown in pseudo code in Figure 5.

No `SELECT` is allowed inside a transaction, because it requires the return of data. If the transaction is running from an application program, the return of data might cause a wait for some user action before the rest of the transaction is submitted. The InterViso user interface submits the entire transaction from `BEGIN` to `END` to the controller as a unit. During the operation of the transaction, the controller has no interaction with the user, so there is no possibility of waiting for user response.

Failures may be difficult to detect. An update sent to a site may fail for several reasons:

- the update is rejected by the DBMS with an error message,
- the update is held or rolled back by the DBMS due to a local deadlock,

- the update message is lost due to a network or processor failure,
- the DBMS site is not available due to DBMS server shutdown or overload,
- the computer with the database is not accessible.

Figure 5. Fragmented relation update

```
Send transaction to all sites in parallel
WHILE work pending
    {
    Read status as each site completes
    IF any step SUCCEED, >0 tuples
     IF not yet committed
      COMMIT
      Log update at all sites still on list
    IF SUCCEED, not end
     Submit next statement in transaction
    IF SUCCEED on END
     Remove from pending list
     IF committed
      remove log for this site
    IF any step FAIL
     IF COMMIT state
      Must run recover
     IF not yet committed
      ABORT
    IF timeout while waiting
     Treat as FAIL at site that timed out
    }
```

InterViso sends a message to a site and then waits for a specified amount of time for a status message. If the message comes back with a failure message, the failure is obvious. When no status message arrives within the time period, it is assumed that the update failed. However, it may actually have succeeded and sometimes the status message arrives after the FAIL state is entered.

5. DBMS Interfaces

Each database is accessed through an InterViso DBMSI. A DBMSI has several parts that are assembled into a module that is specific to a DBMS and operating system. The functions are:

- *Communication.* Handles messages between InterViso processes. Currently the UDP or TCP protocol is used to transport InterViso messages.
- *Parser.* Reads text SQL commands and generates a parse tree. This is DBMS and OS independent.

- *Schema translator.* Transforms the parsed SQL query tree from federated to local form. This may mean renaming relations and fields, joining or splitting relations or fields, transforming fields, or transforming the data in the qualification of the query. The data that are returned are translated into federated form. The DBMS capabilities are described in the DD/D so that the schema translator can determine what functions can be performed in the DBMS and what functions must be done within the translator. This is DBMS and OS independent (see Section 5.1).
- *Language translation.* Generates a query in the local DBMS language from the query tree. Even DBMSs that support SQL require some translation, because each one has some differences in syntax or functionality. This is DBMS language specific.
- *Runtime interface.* Interfaces with the dynamic call level interface to the DBMS. May interact with the DBMS API library on the local host or with the client side of a gateway product to access a remote database. This is DBMS specific. It is built on a toolkit framework with enough flexibility to accommodate almost any database API.
- *Bulk load.* Load data that are being moved from one database to another for joins and unions across databases. Ideally it will call a DBMS bulk load facility if one exists through the API or as a separate process. Otherwise, it will generate an INSERT statement for each record. This is DBMS specific.

DII currently supplies DBMSI routines for relational DBMSs on Unix. Gateways from Oracle, Ingres, Sybase, or Information Builders (EDA/SQL) are used for access to other types of DBMS. In the following section, details of schema translation and the DBMS interface are discussed.

5.1 Schema Translation

Schema translation is the process of converting a query or other statement from the federated schema to the local schema and converting the results back to the federated schema. The DBMSI process performs schema translation on each subquery that is sent to the database it controls. The information needed to do the schema translation is stored in the DD/D by the DD/D Builder (Section 3).

InterViso users express all queries in SQL against the federated schema. The query is sent to the optimizer which, depending on the complexity of the query and data distribution, decomposes the query into one or more statements to the DBMSI processes. Each data element is converted to the federated schema during the first operation on it. Any intermediate results bypass schema translation as no longer needed.

There are two parts to the translation: structural translation and data translation. The structural changes (renaming relations and fields, joining or splitting relations, some changing of field length, and, in some DBMSs, splitting or combining fields) are done by modifying the query. Data translation must be done both on the data

in the qualification part of the query and on the data retrieved. When possible, data translation of results also is done by pushing arithmetic, string and other operations into the query to the extent supported by the local DBMS. The query translation is guided by consulting the DD/D and then calling functions or translate relations as necessary.

Unit, case, and data type changes and other data transformations beyond the local DBMS capabilities require that the data be read into the schema translator and converted into the federated form by calling functions linked into the DBMSI. These functions include a collection of standard operations supported by InterViso, plus locally written special purpose functions. If the data are to be moved to another location, this adds very little overhead. If the data will be used locally, it does mean that they must be read into the program, translated, and then stored back into a temporary relation in the database rather than simply using the DBMS to perform a "select into."

Data in qualification receives three possible levels of translation. In the best case, a federated constant can be rewritten into an equivalent local constant, resulting in the best possible performance. For this to work, a suitable federated to local transformation must have been defined and the predicate must be equality or its negation. In an intermediate case, the conversion of a local value to federated units can be expressed within the query for comparison to other federated values. This case involves more computation, but they are handled close to the data. In the least desirable case, local data have to be passed from the DBMS to the schema translator for evaluation of the qualification. For large relations with selective tests, this results in transferring large amounts of unwanted data between the local DBMS and the schema translator.

The InterViso schema translator differs from the one in Mermaid by supporting a richer set of transformation primitives, and it pays more attention to pushing computation into the local DBMS when possible. In this way it maximizes performance where possible, while still providing a high level of functionality for all databases.

5.2 DBMSI Library Calls

One of the most difficult issues facing InterViso is how to support every DBMS and version of DBMS that can store data. It is prohibitively expensive to own the necessary hardware and to purchase all DBMS products. Therefore, we have decided to support access to only relational DBMS products and commercial gateway products accessible from Unix. On occasion, DII consulting has assisted in developing translators to special purpose systems which are not SQL based and which can not be accessed by commercial gateways.

Each InterViso DBMSI has two DBMS specific components: the language translator and the subroutine call interface. The language translator was an important part of Mermaid and several very diverse types of language were supported. However, the evolution to standard SQL has meant that the language translator has become a single module that has conditional compilation for variants of SQL. Every DBMS

and gateway product supports SQL, but there are differences in each product. The differences include the way that transactions are submitted (e.g., `BEGIN/END` vs `COMMIT WORK`); the use of pattern matching (`LIKE`); the naming of relations by user or by database; and the way to bulk load data.

The DBMS API subroutine calls have little standardization between vendors, although there is a movement toward standardization and newer releases of the DBMSs are slowly converging toward the standards. InterViso uses a table driven interface that defines the calls and parameters for each function that may be needed. This code is supplied to end users in source form, because it sometimes varies between releases of the same DBMS and will continue to vary as the DBMS calls evolve toward the standard.

6. Access Control

InterViso provides at least the level of access control offered by commercial, centralized DBMSs. It also provides excellent protection of databases, but the price is complexity in administering the access control. Access control in a distributed, heterogeneous system is much more complex than the same level of control in a centralized system. There are more processors and system administrators involved, more levels of checking, and different models of access control.

Administrative complexity arises due to the necessity for local control over local databases. In a tightly coupled, distributed system, it is possible to have a central system administration. However, in a federated system such as InterViso, there are data and system administrators for InterViso and for each underlying database. Access through the InterViso system is a right that is granted to individual users. By having each user represented as himself (or part of a small group), existing controls and accounting for machine and database resources can be applied locally without knowledge of InterViso. Access to the InterViso system does not give the user automatic access to the databases accessed by InterViso, and access to the underlying databases does not give a user access to InterViso.

When a user does not have permission to run InterViso or to access the DD/D and underlying databases, the system should lock out the user. Good security demands telling the user very little about the cause of the failure, while fixing it requires seeing which entries are missing or incorrect. InterViso attempts to give good diagnostics without compromising security.

Several federated databases may be defined over different groups of databases and subsets of the databases. Any single database may belong to many federated databases. This capability is another way to control access much as views can be used to control access in the local database. Federated databases carry the concept of views to the federated database level. For example, some group of users may need access to several databases containing information about commercial shipping, while another group of users is interested in military activities. Some databases may contain information about both and, therefore, different subsets of the databases

may participate in different federated databases. If a field does not exist in the federated schema, no InterViso user can access it.

The InterViso access control was described by Templeton et al. (1987*a*). The first step in installing a new user is to determine which federated databases the user will be allowed to access. The user then needs to obtain a login ID on each computer that contains a local database or DD/D for the federated database and on the computer on which he will run his InterViso user interface. In addition, the user needs a login and passwords for each DBMS that provides its own access control and must be granted access permission to the databases as well. The user's login name on each of these host computers and databases and an encrypted version of the matching password is stored in the DD/D using the interactive iv admin program.

If this level of access control and auditing is not required, some of it may be bypassed by assigning multiple real users to the same login IDs on the remote computers. For example, a user "mermaid" might be defined and given a login on a remote computer. All users of InterViso who access that computer will show "mermaid" as their login name on that computer.

The DD/D database contains an iv admin database in which all logins and errors are recorded. The system administrator may also turn on logging of every query that is submitted. The logs may be audited by security officers or accounting personnel.

7. Open Systems Standards

There is a major mismatch between access to legacy systems and use of open systems standards which were defined later. InterViso attempts to provide a standardized view of the system and the databases even when the local database does not use standard SQL and standard calls. At the same time, InterViso should take advantage of standards that do exist.

Mermaid was developed before there were standards for networks, operating systems, SQL, or DBMS calls. We made several fortunate guesses at the direction of standards and our first implementation was on Unix (on the PDP 11) in C, accessing Ingres using the QUEL language and the Britton-Lee using the IDL language.

The original design assumed that there would be many front-end languages, many DBMS query languages, and many network protocols. The modular design has made it possible to modify the system as new standards have been adopted. InterViso was designed after many standards were in place, so it has adopted SQL, TCP/IP, and Unix. However, at the same time, it has remained open to operation with non-standard systems, and it has maintained the modular design so that it can continue to evolve to meet or use the new standards.

The SQL accepted by InterViso is ANSI SQL89 but limited to the portable subset of the commands that InterViso can execute: `SELECT`, `INSERT`, `UPDATE`, and `DELETE` and `BEGIN/END`. Two additions have been made for InterViso: `INTO`

and AT. SELECT INTO allows the user to specify a new relation into which the result is stored. Many dialects of SQL support a similar function. The new relation can then be used by the creator as though it were a fundamental part of the federated database and joined to existing tables. The AT allows the user to specify the site for an operation. In the case of SELECT INTO, it specifies the location of the result as in:

```
SELECT INTO monday AT myhost * FROM sighting WHERE date='04-18'
```

The default for SELECT is to use any copy of a replicated relation and all copies of a fragmented relation. The requirement for INSERT, UPDATE, and DELETE is to reliably apply the update to all copies of a replicated relation and to one or more copies of a fragmented relation. The AT clause may be used to override this, except for updates to replicated relations which must affect all copies. The example shows the location for the INSERT being given with AT:

```
INSERT INTO sighting AT SATVIEW (shiptype, lat, lng, date)
    VALUES ('CG', '002300N','016000W', '06-12')
```

In the federated schema example, the same result may be achieved with the update to the single site table sat_rpt, except that more precision is allowed. When there is a difference in precision, it may be desirable to define update tables that have a schema close to the underlying table:

```
INSERT INTO sat_rpt (shiptype, lat, lng, date)
    VALUES ('CG', '002300N','016000W', '06-12-14-03')
```

InterViso's SQL excludes CREATE TABLE or GRANT. All data definition is done through the DD/D Builder. All access control is done through an administration utility. The limits on transactions were discussed above.

The InterViso API was developed before the SQL Access Group draft standard DBMS Call Level Interface (CLI) was proposed (SQL Access Group, 1992). A new API has been written that follows the 1993 Microsoft ODBC standard. It currently runs as a layer above the original API. Our experience with ODBC reflects the problems that software vendors have in attempting to implement standard software. We originally started implementation following the snapshot CLI standard, but then it appeared that the Microsoft ODBC standard was the one that the application development vendors were following, so we decided to switch. Shortly after we completed the API, Microsoft revised it, which meant that we had to revise our code. At about the same time, the CLI standard was updated with significant differences from the draft and from ODBC. We are now testing various front-end tools with InterViso and we find variations in how calls are made and in what level tools expect the InterViso system to implement.

Standards will eventually make it easier to assemble systems from diverse components, but the industry is not there yet. However, customer's expectations have been raised so that they believe that little systems integration work is required.

8. Lessons Learned

The evolution from Mermaid to InterViso has been influenced by customer demands. Many of the new features in InterViso have been discussed above. This section reiterates some of the features in Mermaid that have proved to be important and then summarizes the new features with a discussion about why they were added.

8.1 Mermaid Features

Mermaid was designed for change with functional layers that have well-defined interfaces. In 1982, this was called "modular design," while today one could call this "object oriented" programming. This has proved to be extremely important since it has allowed the system to change over time.

Debugging features were designed in from the beginning. The development and deployment of distributed code could not have been done without good debugging features. Commands can be entered interactively or in configuration files to turn debugging on to a specified level in either a process or a functional area. A functional area is code such as network code or parser code that exists in many processes. The debug output is written to a local file on the computer where a process runs.

The Mermaid query optimizer is an area that has changed little. Mermaid was the first system with a query optimizer that can support replicated and fragmented data. Support for replicated and fragmented tables at the federated level has been critical for many applications of InterViso. When legacy databases are integrated, it is likely that there are redundant data, because data are stored where they are used, and many applications use the same data. Fragmented data occurs when different organizations have common reporting requirements but collect data on different entities.

Access control was an important component of Mermaid even as just a prototype. Organizations absolutely require good access control for any distributed application that retrieves data from existing, autonomous databases. While Mermaid provided the same types of access control, there have been improvements under InterViso. The most important changes have been the creation of administrative tools for password management, strengthening of the encryption used to protect passwords, and the creation of audit trails to record significant events.

A major area of contention during the design of Mermaid was whether it should start a process per user or handle multiple user requests through a single process. The advantage to handling multiple users in a single process is that less load is put on the computer, and it makes some global coordination easier. The advantages to a process per user are:

1. Access control is strengthened since it requires a login for the user and operations may be logged by user.
2. The operating system accounting routines are used so that users may be charged for database access.
3. The user process sends the commands to the DBMS through the API for the DBMS, most of which will process only a single command at a time. If

the Mermaid code were to handle multiple users, it would have to spawn processes for each command or else single thread the commands to the DBMS.

Mermaid and InterViso use a process per user. This was definitely the correct decision. Computers have become more powerful, so the extra processes have not been a problem. The user's ability to control access and monitor access has been critical.

8.2 InterViso Features

The first addition to InterViso was the IVBuild program to develop the DD/D. The DD/D development tool in use with Mermaid supported only an Ingres DD/D and it could be used only by a person who understood the Mermaid DD/D format. IVBuild is DBMS independent so the customer can store the DD/D in Ingres, Oracle, or Sybase. It comes with a manual that describes the process of developing the DD/D in enough detail that a DataBase Administrator who is familiar with database concepts and with the underlying databases can use IVBuild, without requiring a detailed understanding of DD/D implementation details.

Early Mermaid customers and potential customers did not want to allow updates through any remote program. However, once customers started to use InterViso, they found that they did want to update the underlying databases. Most updates are still made with the existing application programs, but corrections to the data are important.

Mermaid's user interface was either batch or Sun windows. There was no API. The PC has become the primary desktop computer and X-windows has become the standard for Unix workstations. InterViso's user interface options were therefore extended to support X-windows for the interactive interface which allows the windows to be displayed on a workstation or on a PC. The first API was written for InterViso in 1990 before the CLI standard was published. It ran on only the Unix workstation. InterViso now has the ODBC API that supports application programs on the PC.

The performance of InterViso has been adequate except where extensive schema translation is required. The ability to `SELECT INTO table AT site` was added so that frequently accessed data can be translated once and stored as a snapshot where it can be rapidly accessed.

InterViso's DBMSI processes were originally shipped as binary modules. This meant linking in the DBMS subroutine library at DII. Not only did it violate DBMS license agreements, but it made it harder to support all possible DBMS releases. We reorganized the linking process so that the DBMSI is now partially compiled and linked at the customer site. This has caused some new problems, because sometimes different Unix routines are picked up at the customer site than were used to test InterViso.

Many customers have projects to define data element standards for future development. InterViso enforces data element standards at the federated level.

The user organization specifies the standard way to view a data element including the standard name, data type, length, units, precision and/or format. The IVBuild program then automatically makes the federated view conform to the standards. This means that users of InterViso start to write programs and submit queries using the standard view even if the data are not yet standardized.

Maintenance of passwords is the most difficult part of installing InterViso. Passwords must be maintained for each user, and for each host and DBMS. InterViso has an interactive program for entering passwords. They are maintained centrally and then distributed with a utility. InterViso also needs to be able to fail gracefully and with good diagnostics when bad passwords are used.

The processing of a query makes tables in the databases hold intermediate results. If a computer or network connection fails, the table may be left. If a subsequent query uses the same name for a table, it will fail. Mermaid uses simple names for intermediate tables and requires manual cleanup if a failure left an intermediate table in the database. InterViso logs names and then has a cleanup program that will locate and remove the intermediate tables. It also uses randomized names to minimize the likelihood that temporary table names will collide.

9. Conclusion

InterViso has made it possible to access most types of structured data. It provides the "glue" between the user interface and the existing databases. It has been designed to operate with existing software so that it can provide services not supplied by other software products, and it can take advantage of the capabilities of other products. The strengths of InterViso are its ability to resolve differences in schema and data, and its open, standards-based interfaces to DBMS gateway products and to user interface products. InterViso will continue to evolve as more types of data are accessed and as open systems standards evolve.

InterViso is based on the research prototype, Mermaid, but it has been refined and extended to meet the demands of customers. In many cases the solutions are more pragmatic than elegant, but the world of legacy databases is messy and very inelegant.

We see years of improvements ahead as more types of data are brought into the federation, as standards evolve, and as users accept the technology.

References

Ahmed, R., DeSmedt, P., Du, W., Kent, W., Ketabchi, M., Litwin, M., Rafii, A., and Chen, M.-C. The Pegasus heterogeneous multidatabase system. *Computer,* 24(12):19-27, 1991.

Bernstein, P., Goodman, N., Wong, E., Reeve, C., and Rothnie, J. Query processing in a system for distributed databases (SDD-1). *ACM Transactions on Database*

Systems, 6(4):602-625, 1981.

Breitbart, Y., Silberschatz, A., and Thompson, G. An approach to recovery management in a multidatabase system. *The VLDB Journal,* 1(1):1-39, 1992*a*.

Breitbart, Y., Garcia-Molina, H., and Silberschatz, A. Overview of multidatabase transaction management. *The VLDB Journal,* 1(2):181-239, 1992*b*.

Chen, A., Brill, D., Templeton, M., and Yu, C. Distributed query processing in a multiple database system. *Proceedings of the International Computer Symposium,* Taiwan, 1986.

Collet, C., Huhns, M., and Shen, W.-M. Resource integration using a large knowledge base in Carnot. *Computer,* 24(12):55-63, 1991.

Elmagarmid, A.K. and Du, W. A paradigm for concurrency control in heterogeneous distributed database systems. *Proceedings of the Sixth International Conference on Data Engineering,* Los Angeles, 1990.

Epstein, R., Stonebraker, M., and Wong, E. Distributed query processing in a relational database system. *Proceedings of the ACM SIGMOD,* New York, 1978.

Kim, W. On optimizing an SQL-like nested query. *ACM Transactions on Database Systems,* 7(3):443-469, 1982.

Kim, W. and Seo, J. Classifying schematic and data heterogeneity in multidatabase systems. *Computer,* 24(12):12-18, 1991.

Litwin, W., Mark, L., and Roussopoulos, N. Interoperability of multiple autonomous databases. *ACM Computing Surveys,* 22(3):267-293, 1990.

Levy, E., Korth, H.F., and Silberschatz, A. An optimistic commit protocol for distributed transaction management. *Proceedings of the ACM SIGMOD,* Denver, 1991.

Lohman, G., Mohan, C., Haas, L., Lindsay, B., Selinger, P., and Wilms, P. Query processing in R*. IBM research report RJ 4272, April, 1984.

Mehrotra, S., Rastogi, R., Breitbart, Y., Korth, H., and Silberschatz, A. The concurrency control problem in multidatabases: Characteristics and solutions. *Proceedings of the ACM SIGMOD,* San Diego, 1992.

Microsoft. *Microsoft Open Database Connectivity Software Development Kit, Version 2.0, Programmer's Reference.* Redmond, OR: Microsoft Press, 1993.

Perrizo, W. Rajkumar, J., and Ram, P. HYDRO: A heterogeneous distributed database system. *Proceedings of the ACM SIGMOD,* Denver, 1991.

Pu, C., Leff, A., and Chen, S.-W. Heterogeneous and autonomous transaction processing. *Computer,* 24(12):64-72, 1991.

SQL Access Group. *SQL Call Level Interface,* Preliminary specification, X/Open Co., Ltd., UK, October, 1993.

Sheth, A. and Larson, J. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys,* 22(3):183-236, 1990.

Templeton, M. Schema translation in Mermaid. *Heterogeneous Database Workshop,* Chicago, 1989.

Templeton, M., Ward, P., and Lund, E. Pragmatics of access control in Mermaid. *Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering,* 10(3):33-38, 1987*a*.

Templeton, M., Brill, D., Chen, A.P., Dao, S., Lund, E., MacGregor, R., and Ward, P. Mermaid: A front-end to distributed heterogeneous databases. *Proceedings of the IEEE, Special Issue on Distributed Database Systems,* pp. 695-708, May 1987*b*.

Thomas, G., Thompson, G., Chung, C.-W., Barkmeyer, E., Carter, F. Templeton, M., Fox, S., and Hartman, B. Heterogeneous distributed database systems for production use. *ACM Computing Surveys,* 22(3):237-266, 1990.

Veijalainen, J. and Wolski, A. Prepare and commit certification for decentralized transaction management in rigorous heterogeneous multidatabases. *Proceedings of the Eighth International Conference on Data Engineering,* Tempe, AZ, 1992.

Williams, R., et al. ?? R*: An overview of the architecture. IBM Research Report RJ3325, December, 1981.