DB Integrator: Open Middleware for Data Access

by

Richard Pledereder, Vishu Krishnamurthy,
Michael Gagnon, and Mayank Vadodaria

ABSTRACT

During the last few years, access to heterogeneous data sources and integration of the disparate data has emerged as one of the major areas for growth of database management software. Digital's DB Integrator provides robust data access by supporting heterogeneous query optimization, location transparency, global consistency, resolution of semantic differences, and security checks. A global catalog provides location transparency and operates as an autonomous metadata repository. Global transactions are coordinated through two-phase commit. Highly available horizontal partitioned views support continuous distributed processing in the presence of loss of connectivity. The DB Integrator enables security checks without interfering with the access controls specified in the underlying data sources.

INTRODUCTION

A problem faced by organizations today is how to uniformly access data that is stored in a variety of databases managed by relational and nonrelational data systems and then transform it into an information resource that is manageable, functional, and readily accessible. Digital's DB Integrator (DBI) is a multidatabase management system designed to provide production-quality data access and integration for heterogeneous and distributed data sources.

This paper describes the data integration needs of the enterprise and how the DBI product fulfills those needs. It then presents the DBI approach to multidatabase systems and a technical overview of DBI concepts and terminology. The next section outlines the system architecture of the DBI. The paper concludes with highlights of some of the technologies incorporated in DBI.

DATA INTEGRATION NEEDS

Companies often find themselves data rich, but information poor. Propelled by diverse application and end-user requirements, companies have made significant investments in incompatible, fragmented, and geographically distributed database systems that need to be integrated. Companies with centralized information systems are seeking methods to distribute this data to inexpensive, departmental platforms, which would maximize performance, lower cost, and increase availability.

The DB Integrator product family is specifically designed and implemented to address the following data integration needs:

- Data access. The data integration product must provide uniform access to both relational and nonrelational data regardless of location or storage form. Data access must be extensible to allow the user to write special-purpose methods.

- Location and functional transparency. The location of the data and the functional differences of the various database systems must be hidden to provide end users with a single, logical view of the data and a uniformly functional data access system.

- Schema integration and translation. Users of data integration software must be presented with an environment that lets them easily determine what data is available. Such an environment is frequently referred to as a federated database. A data integration product must be flexible enough to help resolve semantic inconsistencies such as variances in field names, data types, and units of measurement.

- Data consistency. Maintaining data consistency is one of the most important aspects of any database system. This is also true for a federated database.

- Performance. Integrating data from multiple data sources can be an expensive operation. The two primary goals are to minimize the amount of data that is transferred across the network and to maximize the amount of rows that are processed within a given unit of time.

- Security. Access to distributed data must not compromise the security of data in the target databases. The security model must provide authorized access to an integrated schema without violating the security of the autonomous data sources that have been integrated.

- Openness. Any data integration product must accommodate tools and applications with standard SQL (structured query language) interfaces, both at the call level (e.g., Open Database Connectivity [ODBC] for personal computer clients) and the language level (e.g., ANSI SQL).[1,2] It must be able to provide and enable access to data over the most commonly deployed transports such as transmission control protocol/internet protocol (TCP/IP), DECnet, or Systems Network Architecture (SNA).[3]

- Administration. The integrated database must provide flexibility in configuration and be easy to set up, maintain, and use.

Figure 1 illustrates the current set of client-server data access supported by the DB Integrator product family.

[Figure 1 (Client-Server Data Access with the DB Integrator) is not available in ASCII format.]

MULTIDATABASE MANAGEMENT SYSTEMS

A multidatabase management system (MDBMS) enables operations across multiple autonomous component databases. Based on the taxonomy for multidatabase systems presented in Reference 4, we can describe DBI as a loosely coupled, heterogeneous, and federated multidatabase system. DBI is loosely coupled compared to the component databases: The database administrator (DBA) that is responsible for DBI and the DBAs that are responsible for the component databases manage their environments independently of one another. DBI is heterogeneous because it supports different types of component database systems. DBI is federated because each component database exists as an independent entity.

Reference Architecture

The MDBMS provides users with a single system view of data distributed over a large number of heterogeneous databases and file systems. The MDBMS interoperates with the individual component databases similar to the way that the SQL query processing engine in a relational DBMS interoperates with the record storage system. Thus, a relational MDBMS, such as DBI, is typically composed of the following processing units:

- Language application programming interface (API) and SQL parser
- Relational data system
  - Global catalog manager
  - Distributed query optimizer and compiler
  - Distributed execution system
  - Distributed transaction management
- Gateways to access data sources

Catalog Management

One of the key differentiators between MDBMS architectures is the way that the metadata catalog is organized. Metadata is defined as the attributes of the data that are accessible (e.g., naming, location, data types, or statistics). The metadata is stored in a catalog. Two common approaches for catalog management are described below:

- o Autonomous catalog. The MDBMS maintains its own catalog in a separate database. This catalog describes the data available in the multidatabase. For data that resides in a relational database, the metadata definitions of table objects, index objects, and so forth, are imported (i.e., replicated) into the multidatabase catalog. For data that resides in some other data source such as a record file system (e.g., record management system [RMS]) or a spread sheet, the MDBMS catalog contains a relational description of that data source.

- o Integrated catalog. The MDBMS is integrated with a regular database system that is capable of accessing objects (both data and metadata) in remote and foreign databases. A gateway server is responsible for making a foreign database appear as a homogeneous, remote database instance. For data that resides in a relational database, the gateway server stores views of its system relations into that database. For data that resides in a record file system or spread sheet, the gateway server stores the relational metadata description of the data in a separate data store.

DBI Approach

The DBI approach to multidatabase management very closely follows the reference architecture presented earlier. The DBI approach emphasizes the following design directions:

- o Global, autonomous catalog for metadata management

- o Three-tier integration model (described later in this section)

- o Simple, mapped-in gateway drivers to access data sources

- o Support of distributed database features for the Oracle Rdb relational database as well as support of existing Oracle Rdb applications in the multidatabase environment

Global Catalog. DBI is addressable as a single integration server. Integration clients such as tools and applications do not need to deal with the complexities of the distributed data. The DBI global catalog is a repository in which DBI maintains the description of the distributed data. It enables DBI to provide tools and applications with a single access point to the federated database environment. The global catalog enables DBI to tell users what data is available without requiring immediate connectivity to the data or its data source. It can be managed and maintained as an independent database. The maintenance of the

DBI global catalog is not inherently tied to a specific data manager; currently, the DBI catalog may reside in ORACLE, SYBASE, or Oracle Rdb databases.

The use of a global catalog may result in a system with a single point of failure. To eliminate its potential failure within a node, a disk, or a network, standard high-availability mechanisms may be employed. These include shadowed disks with shared access (e.g., clustered nodes) and data replication of the DBI catalog tables with products such as the Digital Data Distributor.[5]

Three-tier versus Two-tier Architecture. With a two-tier data integration model, once the data has been retrieved from the server tier, the actual integration occurs on the client tier. This may result in massive integration operations at the client site. In contrast, the DBI is based on a three-tier architecture that performs most integration functions on a middle tier between the client and the various database servers. The three-tier approach avoids unnecessary transfer of data to the client and is essential to providing production-quality data integration. In another comparison, all clients in the two-tier approach need to be configured to access the various data sources; however, the three-tier approach significantly reduces such management complexities.

Gateway Driver Model. DBI deploys a set of gateway drivers to access specific data sources, including other DBI databases. These drivers share a single operating system process space with DBI to avoid unnecessary interprocess communications. When DBI performs parallel query processing, however, gateway drivers may reside in a separate process space. The core of DBI interacts with the actual gateway drivers (e.g., a SYBASE gateway driver) through the Strategic Data Interface (SDI), an architected interface that is used within the DBI product family as a design center.[6] A gateway driver is implemented as a relatively thin software layer that is SDI compliant and that is responsible for handling impedance mismatches in data models (e.g., RMS versus relational), query language (e.g., different dialects of SQL), and run-time capabilities (e.g., SQL statement atomicity).

Distributed Rdb. One of the design goals for DBI was to enable distributed database processing for DEC Rdb (now Oracle Rdb).[7] From the perspective of an application, DBI therefore looks like a distributed Rdb database system.

DBI CONCEPTS AND TERMINOLOGY

In this section, we present a brief overview of the concepts and terminology relevant to DBI.

DBI Database

A DBI database consists of (1) a set of tables that DBI creates to maintain the DBI metadata (also referred to as the catalog) and (2) the distributed data that is available to the user when connected to the DBI catalog.

A DBA creates a DBI database using the DEC SQL CREATE DATABASE statement. This statement has been extended for DBI to allow the user to indicate the physical database (e.g., a SYBASE database) that will be used to hold the DBI metadata tables.

The creator of a DBI database automatically becomes the owner and system administrator of that database. A DBI system administrator may grant access privileges on the DBI database to other users. Depending on the level of privilege, a user may then perform system administration functions, execute data definition language (DDL) operations, and/or query the tables in the virtual database.

DBI Objects

In addition to regular SQL objects such as tables or columns, DBI uses objects, links, and proxies that are outside the scope of the SQL language standard.

Links and Proxies. The link object tells DBI how to connect to an underlying data source (referred to as the link database). A link object has three components: a link name, the access string used to attach to the link database, and, optionally, security information used by the DBI gateway driver to provide authentication information to the link database system. The proxy object is associated with a link object. It can be used to specify user-specific authentication information for individual links. When users do not want to use proxies for their links, they must specify the authentication information for a specific database at the time they connect to DBI.

Tables. With link and proxy objects in place, the user can import metadata definitions of underlying tables into the DBI catalog. The metadata imported for a table includes statistics, and constraint and index information, all of which are used by the DBI optimizer. The import step is performed with a CREATE TABLE statement that has been extended to allow for a link reference. For example:

```
-- Import "rdb_emp" table into DBI database as "emp"
-- from the link database represented by the link
-- named "link_rdb".

--
CREATE TABLE emp LINK TO rdb_emp USING link_rdb;
```

Views. View objects are useful for making multiple tables from different link databases appear as a single table. In DBI, views serve as powerful mechanisms to resolve semantic differences in tables from disparate databases. DBI supports two types of views: regular SQL views and horizontally partitioned views (HPVs). Regular views are compliant with ANSI SQL92 Level 1; they support full query expression capabilities and updatability.[2] HPVs consist of a view name, a partitioning column, and partition specifications. Figure 2 is an example of an HPV definition.

Figure 2 Example of an HPV Definition

```
        CREATE VIEW emp (emp_id, first_name, last_name, country)
               USING HORIZONTAL PARTITIONING ON (country)

               PARTITION us WHERE country = 'US' COMPOSE AS
                         SELECT employeeid, firstname, lastname, 'US'
                         FROM emp_us

               PARTITION europe WHERE OTHERWISE COMPOSE AS
                         SELECT emp_id, first_name, last_name, country_code
                         FROM emp_eur;
```

HPVs provide a very powerful construct for defining a logical table composed of horizontal partitions that may span tables from disparate data sources. Both retrieval and update operations on HPVs are optimized such that unnecessary partition access is eliminated. In addition, HPVs may be used to implement a shared-nothing computing model on top of both homogeneous and heterogeneous databases.[8]

Stored Procedures. DBI supports stored procedure objects. Stored procedures allow the user to embed application logic in the database. They make application code easily shareable and facilitate DBI to maintain dependencies between the application code and database objects. Furthermore, stored procedures reduce message traffic between the client and the server. Figure 3 is an example of a stored procedure.

Figure 3 Example of a Stored Procedure

```
        procedure maintain_salaries(:state char(2) in,
                                    :n_decreased integer out);
        begin
            set :n_decreased = 0;
            for :empfor as each row of
                 select * from employees emp where state = :state;
             do
                  set :last_salary = 0;
                  history_loop:
```

```
                  for :salfor as for each row of
                      select salary_amount from salary history s
                      where s.employee_id = :empfor.employee_id
               do
                      if :salfor.salary_amount < :last_salary then
                          set :n_decreased = :n_decreased + 1;
                          leave history_loop;
                      end if;
                       set :last_salary = :salfor.salary_amount
                  end for;
            end for;
        end;
```

DBI Database Administration

DBI supports statements that keep the imported metadata consistent with the link database. The extended ALTER TABLE statement may be used to regularly refresh the table metadata information or update the table's statistics. The ALTER LINK statement may be used to modify the link database specification or a proxy for a given link object.

DBI Configuration Capabilities

Figure 4 shows the power of configuration options supported by DBI. Following the three-tier model for data integration, the DBI server may access a very large number of databases, including other DBI databases.

[Figure 4 (DBI Configuration Capabilities) is not available in ASCII format.]

The DBI server is accessible through SQL APIs that are available on popular client platforms. DBI's client-server protocol is supported on all common transports such as TCP/IP, Novell's sequenced packet exchange/internetwork packet exchange (SPX/IPX), DECnet, or Windows Sockets. DBI itself may be deployed on Digital UNIX (formerly DEC OSF/1) and OpenVMS platforms today. Support for additional platforms is being added.

DBI SYSTEM ARCHITECTURE

In this section, we describe the system architecture of the DBI product family and present some of its specific designs.

Interfaces

As shown in Figure 5, the DBI system architecture is anchored by two external interfaces, SQL and metadata driver interfaces/data driver interfaces (MDI/DDI), and two internal interfaces, Digital

Standard Relational Interface (DSRI) and SDI.

The SQL interface is used by DBI clients to issue requests to the integration server. The MDI/DDI interface is used by DBI to call gateway drivers that are provided by a user. The MDI/DDI interface specifies a simple, record-oriented data access interface provided by Digital to assist users in the access and integration of data sources for which no Digital-supplied gateway drivers are available.

DSRI is the interface between DBI's SQL parser and the DBI processing engine.[9] The SDI interface specifies a canonical data interface that shields the DBI core from data-source-specific interfaces and facilitates modular development.[6]

[Figure 5 (DB Integrator Architecture) is not available in ASCII format.]

Components

The component architecture of DBI in Figure 6 closely resembles the multidatabase reference architecture presented earlier:

- The SQL and ODBC client-server environment provides language API and SQL parser functions.

- The API driver and context manager support distributed transaction management and part of the distributed execution system.

- The metadata manager provides global catalog management.

- The compiler supports the distributed query optimizer and compilation.

- The executor supports the remaining part of the distributed execution system.

- The SDI dispatcher and gateway drivers provide the access to data sources.

[Figure 6 (DB Integrator Components) is not available in ASCII format.]

SQL Environment and Server Infrastructure. The SQL parser supports DEC SQL, an ANSI/National Institute for Science and Technology (NIST)-compliant SQL implementation by mapping DEC SQL syntax into an internal query graph representation.[9] In a client-server environment, the DBI server infrastructure is used to manage, monitor, and maintain a DBI server configuration that supports workstation and desktop clients.

API Driver and Context Manager. The API driver is responsible for the top-level control flow of client requests within the DBI core. It currently accepts DSRI calls from applications such as DEC SQL and dispatches them within DBI. The context manager performs demand-driven propagation of execution context to the gateway drivers and maintains the distributed context of active sessions, transactions, and requests.

Metadata Manager. The metadata manager is responsible for the overall management and access to metadata. The services provided fall into the categories of catalog management, data definition, metadata cache management, and query access to DBI system relations. The metadata catalog manager maintains the DBI catalog in the form of DBI-created tables in an underlying database (e.g., SYBASE or ORACLE). The DDL processor executes the data definition statements. The metadata cache manager is responsible for maintaining metadata in a volatile cache that provides high-speed access to metadata objects.

Compiler. The compiler provides services for translating SQL statements and stored procedures into DBI execution plans. A rule-based query optimizer performs query rewrite operations, enumerates different execution strategies, and factors in functional capabilities of the underlying data sources. Each execution strategy is associated with a cost that is based on predicate selectivity estimates, table cardinalities, availability of indices, network bandwidth, and so forth. The lowest cost strategy is chosen as the final execution plan. Above a certain threshold of query complexity, the optimizer switches from an exhaustive search method to a greedy search method to limit the computational complexity of the optimization phase. The compiler generates code that can be processed by the executor component and the gateway drivers.

Executor. The executor component is responsible for processing the execution plan that the compiler produces. These activities include

- Exchanging data between the DBI and the client
- Streaming data between the DBI core and the link databases
- Performing intermediate data manipulation steps such as joins or aggregates
- Managing workspace and buffer pool to efficiently handle large amounts of transient and intermediate data
- Supporting parallel processing

SDI Dispatcher and Gateway Drivers. The SDI dispatcher separates the core of DBI from the gateway driver space. It locates and loads shareable images that represent gateway drivers and routes SDI calls to the corresponding entries in the gateway driver image.

TECHNICAL CONSIDERATIONS

The DBI development team selected several designs and technologies that it believes to be crucial for distributed and heterogeneous data processing. This section summarizes those designs within the following functional units: distributed execution; distributed metadata handling; distributed, heterogeneous query processing; high availability; performance; and DBI server configuration.

Distributed Execution

To support transparent distributed query processing, DBI propagates execution context such as connection context or transaction context to the target data sources. Tools and applications see only the simple user session and transaction that they establish with the DBI integration server.

DBI uses a tree organization to track the distributed execution context. When a user connects to a DBI database, a DBI user session context is created. This session context is subsequently used to anchor active transactions, compiled SQL statements, as well as the metadata cache that is created for every user attaching to DBI. When DBI passes control to a gateway driver, both session and transaction context are established at the target data source.

Distributed transactions must support consistency and concurrency across autonomous database managers. Consistency requires that a distributed transaction manager with two-phase commit logic is available. DBI uses the Digital Distributed Transaction Manager (DDTM) for that purpose and is adding support for the distributed transaction processing (DTP) XA standard integration.[10,11]

Concurrency requires that distributed deadlocks are detected. In a multidatabase system, distributed deadlock prevention is not feasible because no database manager exposes external interfaces to its lock management services -- a procedure required to perform deadlock detection. DBI therefore relies on the simple technique of transaction time-out to detect deadlocks. In addition, a DBI application may choose to specify isolation levels lower than serializability or repeatable read. This is done with the SQL SET TRANSACTION statement. The DBI context manager records the transaction attributes specified and forwards them to the underlying data sources as part of propagating transaction context. Lower isolation levels will, in general, result in fewer lock requests and thus fewer deadlock situations.

Distributed Request Activation. DBI supports SQL statement atomicity. This requires either that a single SQL statement executes in its entirety or, in the case of a failure, that the database is reset to its state prior to the execution of the statement. With DBI, the SQL statement may be executed as a series of database requests at multiple data sources. DBI internally uses the concept of markpoints to track SQL statement boundaries. Gateway drivers are informed of markpoint boundaries, and the driver attempts to map the markpoint SDI operations into semantically equivalent constructs (e.g., savepoints) at the target data source. Some databases support SQL3-style savepoints, which are atomic units of work within a transaction. When DBI decides to roll back a markpoint, the gateway driver may then inform such a data source to roll back to the last savepoint. In the absence of markpoint primitives in the target data source, the gateway driver may elect to roll back the entire transaction to meet the roll-back markpoint semantics.

Gateway Drivers. In contrast with other data integration architectures, the DBI gateway drivers are designed to be simple protocol and data translators. Their primary task is to report the capabilities of the data-source interface (API and SQL language) to the DBI core and subsequently map between the SDI interface protocol and the data-source interface. The gateway drivers typically share process context with the DBI server process, thus avoiding the need for an intermediate gateway server process that would otherwise reside between the DBI server and the data-source server (e.g., SYBASE SQL Server). This reduces the amount of context switching and interprocess message transfer.

The gateway drivers are responsible for mapping the SDI semantics to the interface primitives provided at the target data source. For relational databases such as Oracle Rdb, ORACLE, INFORMIX, SYBASE, or DB2, this requires primarily a mapping to the product-specific SQL dialect and the product-specific data types. For file systems such as RMS, the gateway driver maps the SDI semantics to calls to the RMS run-time library.

Distributed Metadata Handling

In this section, we discuss three areas of importance to the handling of metadata in DBI: catalog management, security, and metadata caching.

Catalog Management. The DBI requirement of database independence implies that DBI cannot require the presence of a particular DBMS for its persistence metadata storage. Rather than devising a private storage and retrieval system, DBI was designed to layer on top of common relational DBMSs.

Static, precompiled native applications are used to access metadata from a given catalog DBMS for two reasons: (1) The pattern of metadata access for the catalog database is known, and (2) The tables housing the DBI metadata in the catalog database are predetermined. Although this approach does not take advantage of the existing gateway drivers, it results in high-performance access to the metadata store.

To simplify the development of a catalog application, the set of primitive operations on the catalog database was isolated, and a catalog application interface (CI) was defined. Catalog applications are developed according to the CI specification and implemented as shareable images. DBI dynamically loads the appropriate catalog application image based on the catalog type specified by a user attaching to a DBI database.

Security. The security support in the currently released version 3.1 of DBI is simple but effective. It uses the security mechanisms of the underlying link database systems in the following areas:

o Authorization to connect to an underlying database through DBI and access data from it.

  Access to the data that is manipulated through DBI is controlled by the underlying DBMS. Typically, underlying database systems control access to data based on the identity of the user attached to its database. DBI supports objects called proxies that enable the client to specify its user identity (i.e., username/password), which is then used to attach to the underlying database.

o Authorization to perform various DBI operations.

  All privileges for a DBI database are for the database itself, rather than for tables or columns. The privileges are based on hierarchically organized categories of users:

  - The DBADM privilege is given to users responsible for setting up and maintaining a DBI database.

  - The CREATE, DROP privilege is granted to interactive users and application developers with database design responsibility who must perform data definition operations.

  - The SELECT privilege is reserved for interactive users and application developers who perform data manipulation operations but do not perform any data definition operations.

When a DBI administrator grants or revokes privileges for a DBI database, DBI, in turn, grants or revokes the appropriate set of privileges on the DBI tables in the database system that manages the DBI catalog. The enforcement of privileges is therefore carried out by that database system. For example, when the SELECT privilege is granted on the logical database, DBI grants the SELECT privilege on the tables that represent the DBI catalog. This ensures that the user has access to the metadata for processing queries. Similarly, when a user is granted the CREATE, DROP privilege on the DBI database, DBI grants SELECT, INSERT, UPDATE, and DELETE on the appropriate tables in the catalog database to the user. This ensures that any DDL actions executed by the user will enable DBI to modify the tables in the catalog database.

Metadata Manager Cache. The in-memory metadata cache serves a dual purpose. First, it facilitates rapid access to the metadata by the DBI compiler. Second, it serves as a data store for the DBI system relations that can be queried by tools and applications. For example, DEC SQL obtains metadata for semantic analysis of SQL statements by querying the DBI system relations.

The metadata cache is structured as a single hash table representing a flat namespace across all DBI objects. An open hashing scheme is employed in which the hash-table entries hang off the buckets in the hash table in a linked list.

To optimize the use of the cache as well as to accelerate the attach operation, the metadata manager initially obtains only minimal, high-level metadata information from the catalog database; for example, only names of tables are fetched into the cache during the DBI database attach operation. Subsequently, the metadata manager obtains further metadata information from the catalog database on a demand basis.

DBI allows the creation of new metadata objects. These operations are typically performed within markpoint and transaction boundaries to enforce proper statement and transaction demarcation. The metadata manager maintains a physical log in cache to denote transaction and markpoint boundaries. The log is an ordered list of structures, each representing a DDL action, a pointer to the cache structure that was changed, and either the previous values of fields that were updated or a pointer to a previous image of an entire structure. When a markpoint or transaction is committed, the corresponding log part is reset; when a markpoint or transaction is rolled back, the log is used to restore the cache to its state prior to the start of the markpoint or transaction.

An object in cache can become stale when another user attaches to the DBI database and causes an object's metadata to be changed in the catalog database. To ensure consistency of the cached version of an object's metadata with the actual version in the catalog database, the metadata manager uses a time stamp to check the currency of the cached object when performing incremental fetching of the object's metadata. If the object in cache is stale, the object is not accessible in the session, and an error message is issued to the user indicating that the object in cache is inconsistent with the catalog database. In a production environment, this would be a rare event, given the low frequency of data definition operations.

The metadata cache is also the data source for the DBI system relation queries. The metadata manager navigates the cache structures to obtain data for the system relations, making use of the hash table for efficient access and using DBI's execution component for evaluating search conditions and expressions.

Distributed, Heterogeneous Query Processing

Distributed query processing in a heterogeneous database environment poses certain unique problems. Data sources behave differently in terms of data transfer cost, and they support different language constructs. Many systems employ rudimentary techniques for decomposing a query, frequently pulling in all the data from underlying tables to the processing node, and then performing all the operations in the integration engine. Others simply use syntactic transformations, thereby providing the least common denominator in language functionality. DBI, on the other hand, provides a robust query optimizer that includes decomposition algorithms to reduce the data flow and provide high-performance query execution.

Cost-based Plan Generation. When a query has several equivalent means of producing the result, the plan that has the least estimated cost is chosen. Statistics for table, column, and index objects are used for estimating result size after various relational operations.[12,13] Data transmission costs from the underlying link database to DBI are taken into account when estimating how much of the query is to be sent to the gateway database. The network transmission cost is measured dynamically for each user session, once per gateway connection. The cost associated with performing a relational operation is also aggregated into the overall cost. This crucial step ensures that the plan is not skewed toward one database engine, which would be the case if only the network transmission costs were taken into account.

Rule-based Transformations. A query result may be produced with different sequences of relational operations. These sequences are generated using rule-based transformations. The starting point is the original operation set in which the query was syntactically represented. From this, permutations are generated to form equivalence sets, which then lead to the various combinations of execution plans that need to be examined for cost. Finally, the least costly plan is chosen for the query. Heuristics are applied to limit the amount of search space.

Capability-based Decomposition. The critical characteristic of a heterogeneous environment is that the data sources are nonuniform in their ability to perform certain operations and in their support of various language constructs. For example, most databases cannot support derived table expressions (i.e., select expressions in the FROM clause of another SELECT statement).

The plan generation and decomposition phases of the optimizer must recognize the underlying databases' capabilities. Consider the query example shown in Figure 7 and the indicated locations of the tables.

Figure 7 Example of an SQL Query

```
select *
  from T1, T2, T3
 where (T1.c1 = T2.c2)
   and (T1.c3 = T3.c3)
   and (T1.c5 = (select avg(T4.c5) from T4)
                  + (select T5.c7 from T5 where T5.c8 = 'a') );
```

T1, T3, T4 and T5 are located in a Oracle database.
Table T2 is located in a DB2 database.

First, with T1 and T3 located in the same database, the optimizer can generate a subplan in which the join between these two tables can be executed in the ORACLE database. An examination of the last (third) AND predicate indicates that all the tables involved in that predicate are located in the same ORACLE database. Due to the limitations in ORACLE's SQL language support, however, it cannot evaluate the combined expression between two subqueries in the WHERE clause, where the arithmetic result is to be compared to the column T1.c5.

The DBI optimizer employs a more sophisticated alternative. It evaluates the two subqueries separately and then substitutes them in the predicate in the subplan for ORACLE as run-time parameter values. This technique leads to the most efficient plan:

1. Retrieve value for (select avg(T4.c5) from T4) from ORACLE.

2. Assign value to variable X.

3. Retrieve value for (select T5.c7 from T5 where T5.c8 = 'a') from ORACLE.

4. Assign value to variable Y.

5. Assign param_1 := variable X.

6. Assign param_2 := variable Y.

7. Execute the SELECT statement below in ORACLE and fetch the result rows.

```
select *
  from T1, T3
 where (T1.c3 = T3.c3)
   and (T1.c5 = param_1 + param_2);
```

8. Fetch the rows of T2 from DB2 into DBI.

9. Perform the join in DBI between the results of steps 7 and 8.

Query Unnesting. A nested SQL query, in its simplest form, is a SELECT query with the WHERE clause predicate containing a subquery (i.e., another SELECT query). The following are examples of nested SQL queries:

Example 1, Table Subquery

```
select *
  from A
 where A.c1 IN (select (B.c2 + 5)
                  from B
                 where B.c3 = A.c3);
```

Example 2, Scalar Subquery

```
select *
  from A
 where A.c1 = (select max(B.c2)
                 from B
                where B.c3 = A.c3);
```

Using strict SQL semantics, we can evaluate this nested query by computing the results of the inner subquery for every tuple in the outer (containing) query block. The value for the column A.c3 is substituted in the inner subquery, and the resulting value (or values) are computed for the select list and used to evaluate the Boolean condition on column A.c1: this is repeated for every tuple of A. This method of evaluating the results is very expensive, especially in a distributed environment.

Query unnesting algorithms provide other methods of evaluation that are semantically equivalent but much more efficient in both time and space. Unnesting deals with the transformation of nested SQL queries into an equivalent sequence of relational operations. These relational operations are performed as set operations, thereby avoiding the expensive tuple iteration operators during execution and providing large performance gains in most cases. The background and motivation behind the use of unnesting has been presented in several research papers.[14,15]

Depending on the type of operations and constructs found in the nested select block and its parent select block, several different algorithms can be used. Some of these require no special operators over and above the regular join operator. Other transformations require a special semijoin operator. Consider the examples shown in Figure 8.

Figure 8 Query Unnesting Algorithm

```
--
-- Q1  - query that will not require a special join after transformation
--
select snum, city, status
  from S
 where status = (select avg(weight) + 5      -- nesting predicate
                   from P
                  where P.city = S.city);    -- correlation predicate

--
-- Q1-U - the unnested version
--
select snum, city, status
  from S, (select city, avg(weight) + 5
             from P
            group by city) as T1(c1,c2)
 where T1.c1 = S.city
   and S.status = T1.c2;

-- Algorithm:
--
-- 1) Take the inner block's FROM table that has a correlation predicate.
-- 2) Add a Group-By to the inner block containing all attributes of this
--    table that appear in one or more correlation predicates.  The order of
--    the attributes in the Group-By does not matter.
-- 3) Also, add these elements to the select list of the inner block; at the
--    beginning or at the end, whatever is convenient.
-- 4) Next, add this block to the FROM list of the outer block - effectively
--    doing a regular join with the tables in the outer FROM list.
-- 5) Lastly, rewrite the correlation and nesting predicates as shown.
```

In the example shown in Figure 9, a special operator called semijoin is necessary. The semijoin of table R with S on condition J is defined as the subset of R-tuples for which there is at least one matching S-tuple satisfying J. Note that this makes the operator asymmetric, in that (R semijoin S) is not the same as (S semijoin R), whereas the regular join is symmetric. By implementing the special semantics required for this semijoin operator, we can transform the nested query into this join operator that can again make use of high-performance techniques like hash joins within the DBI execution engine.

Figure 9 Algorithm with Semijoin Operator

```
-
- Q2 - query requiring a semi-join
-
select snum
  from S
 where city IN (select city
                  from P
                 where P.weight = S.status);

-
- Q2-U - the unnested version
-
select snum
  from (S  semi-join  P
           on (P.weight = S.status AND S.city = P.city)
       );

-- Algorithm:
--
-- 1) Do a semi-join between S and P using the following (combined) condition:
--      "(P.weight = S.status) AND (S.city = P.city)"
--    In reality, this is actually specified as 2 separate semi-joins between
--    S and P, one with the correlation predicate and one with the form of
--    the nesting predicate.  But these get combined using rules.
-- 2) Project out S.snum from the result
```

Predicate Analysis. When a query against an HPV can be satisfied by simply accessing a single logical partition, then the rest of the partitions can be eliminated from the execution plan. Partition elimination algorithms in DBI are used both at compile time, when the predicates on the HPV query involve comparison of the partitioning column with literals, as well as at query execution time (run time), when the partitioning column is compared with run-time parameters.

During affinity analysis, predicates are situated as close to the inner table operation as feasible. For example, consider the following view definition, and the subsequent select statement on that view:

```
create view V1 (a, b) as
  select T1.c1, avg(T2.c2)
    from  T1, T2
   where (T1.c4 = T2.c4)
   group by  T1.c1;

select * from V1 where (a = 5 and b > 10);
```

The predicate a = 5 (upon further conjunctive normal form [CNF] analysis) can be applied on the base table scan itself as T1.c1 = 5.

Index join is one of the efficient join techniques used in DBI. This join technique minimizes the movement of data from the link databases by taking advantage of the indexing schemes in the link database to facilitate the join process. Consider the following query:

```
select *
  from  T1, T2
 where  T1.c1 = T2.c2 + 5
   and  (...some restrict predicate(s) on T2...)
```

Given an index on column c1 of table T1, and with cardinality and cost estimates permitting, the query optimizer can generate an alternate plan. This plan allows the join to be performed by using efficiently indexed access retrieval for table T1.

High Availability

High availability in DBI results from the use of horizontal partitioned views and catalog replication.

Horizontal Partitioned Views. An HPV is a special kind of view in which DBI is provided with information about how data is distributed among tables in link databases. HPVs offer many advantages over normal views, one of them being improved performance through partition elimination and use of parallelism. The other advantage is high availability.

If a partitioned view has multiple partitions and if some partitions are unavailable when the view is queried, then DBI does not fail the query but returns data from the available partitions. An example is shown in Figure 10. The example creates a partitioned view named ALL_EMPLOYEES, with four columns and three partitions, each of which obtains rows from three different tables. The partitioning is based on a specific column, in this case the CITY column, as specified in the USING HORIZONTAL PARTITIONING ON clause.

Figure 10 Example of a Partitioned View

```
CREATE VIEW ALL_EMPLOYEES(ID, NAME, ADDRESS, CITY)
USING HORIZONTAL PARTITIONING ON CITY
PARTITION P1 WHERE CITY = 'MUNICH'
          COMPOSE AS SELECT ID, LAST_NAME, ADDRESS, 'MUNICH'
                     FROM   MUNICH_EMPLOYEES
                     WHERE  STATUS = 'Y'

PARTITION P2 WHERE CITY = 'PARIS'
          COMPOSE AS SELECT ID, FULL_NAME, ADDRESS, 'PARIS'
                     FROM   PARIS_EMPLOYEES
                     WHERE  STATUS = 'Y';

PARTITION P3 WHERE CITY = 'NASHUA'
          COMPOSE AS SELECT ID, FULL_NAME, ADDRESS, LOCATION
                     FROM   NH_EMPLOYEES
                     WHERE  STATUS = 'Y';
```

Suppose the following query is submitted

```
SQL> SELECT * FROM ALL_EMPLOYEES
              WHERE (CITY = 'MUNICH')
                 OR (CITY = 'NASHUA');
```

First, partition P2 is eliminated at compile time. Now suppose partition P3 is presently not available due to network connectivity problems (Figure 11). For each partition that is unavailable, a message is returned indicating that some rows are missing from the result table: %DBI-W-HAHPV_UNAVAILABLE Partition P3 is currently unavailable. However, DBI still attempts to return as much data as is accessible.

[Figure 11 (High Availability with Partitioned Views) is not available in ASCII format.]

Catalog Replication. To prevent the DBI global catalog from becoming a single point of failure, multiple copies of a catalog table can be maintained by using replication techniques. Catalog table copies can be created easily and maintained using replication tools such as the DEC Data Distributor.[5]

Performance

In addition to its distributed query optimizer, DBI uses a series of techniques to increase the speed of query processing, most notably in the areas of data transfer, memory management, join processing, parallelism, and stored procedures.

Data Transfer. The DBI execution engine performs bulk data transfer using the bulk fetch mechanisms provided by the SDI interface. With bulk data transfer, a single request message to a local or remote data source returns many tuples with a single response message. Bulk transfer techniques are mandatory in a distributed environment; they reduce both message traffic and stall waits due to message delays. The data transfer bandwidth between the DBI engine and the gateway drivers is further increased through the use of asynchronous SDI operations.

Memory Management. An MDBMS needs to be able to process large amounts of data efficiently without exceeding platform- or user-specific operational quotas such as the page file size or the working set limit. In addition, standard operating system paging techniques may easily result in heavy I/O thrashing for database-centric work loads.

The DBI executor places data streams, intermediate query results, or hash buckets into individual workspaces. A workspace is organized as a linear sequence of fixed-size pages. A standard page-table mechanism identifies the allocated pages and records status such as whether a page is present in memory or whether it is paged out to secondary storage. The workspace manager operates as an intelligent buffer manager and paging system that controls fair access to memory across all active workspaces of a given DBI user. A buffer pool manager holds the workspace pages that reside in memory.

The buffer pool manager supports multiple buffer replacement policies, which is important for database workloads that involve sequential access to data that is subsequently no longer needed. The two supported strategies are least recently used (LRU) and most recently used (MRU).[16] Finally, the workspace manager supports write-behind for newly allocated pages. This allows newly allocated pages that have been filled to be written asynchronously.

Join Processing. Highly efficient processing of joins and unions is important in any commercial database; it is crucial for a multidatabase system. DBI supports nested loop join, index join, and hash join. In fact, DBI supports both a regular hash-join mechanism and a hybrid, hash-partitioned variant that is augmented with Bloom filtering.[17,18,19]

For both hash-join variants, the inner table rows are read asynchronously into a DBI workspace. This first pass is used to estimate whether or not to use the hash-partitioned variant. An exact estimate for the number of partitions to use is well worth the overhead of this initial pass.[20] In addition, a Bloom filter with 64 kilobits is populated as part of this pass. The inner table cardinality, an estimate for the outer table cardinality, and an estimate of the presently available memory are used to determine whether the simple hash-join technique is sufficient, or whether the use of the hybrid hash-partitioned join technique is warranted.

In general, hash-partitioned join processing is indicated when the inner table and its hash-table buckets do not fit in memory. In this case, both the build phase for the inner-table hash buckets as well as the probe phase of outer-table tuples against the inner-table hash buckets may incur massive amounts of random I/O. When the hash-partitioned variant is selected, the following steps are performed.

- Each partition receives a separate workspace.

- The inner table is partitioned first. During this partitioning step, a Bloom filter is generated from the join column of inner-table tuples and is applied when the outer table rows are partitioned. This results in a potentially massive reduction of the number of rows that are placed into the outer partitions, thus eliminating expensive I/O operations.

- The workspaces that hold the inner-table partition 1 and the hash-table buckets for that partition are aged LRU, which keeps them in memory for the join operation on the first partition pair.

- The workspaces that hold the remaining inner-table partitions 2 through (n) are aged MRU; these pages become immediately available for buffer replacement selection once they have been filled and their frames unpinned.

- Once the partitioning phase is complete, each pair of inner and outer partitions is joined starting with partition pair 1. The inner partitions are aged LRU, and the outer partitions are aged MRU to keep the inner partition tuples in memory.

The use of flexible buffer replacement strategies is crucial for good buffer cache behavior.

Parallelism. DBI employs two types of parallelism: pipelined parallelism and independent parallelism.[8]

With hash-join processing, for instance, the outer table rows are read by separate DBI execution threads from the underlying database. This means that the outer table tuple stream is effectively generated in parallel with the probe phase processing of the hash-join operator on the inner table rows. The outer-table tuple stream is directed into the hash-join probe phase.

For UNION processing on partitioned views, the individual input streams to the UNION operator are generated by separate DBI execution threads. The streams are provided in parallel and independently to the UNION operator.

Stored Procedures. Stored procedures provide a critical performance enhancement for client-server processing. They allow the DBA to encapsulate a set of SQL statements plus control logic. The client sends one message containing a stored procedure rather than several messages, each containing one SQL statement. This reduces processing delays that otherwise would be incurred due to network traffic.

DBI Server Configuration

In a standard DBI configuration, one execution process is created for each DBI client. As the number of clients increases, more and more operating system resources are consumed. The DBI server configuration addresses this problem.

Server Components. A DBI server configuration includes minimally a monitor process, a dispatcher process, and a set of DBI executor processes. The monitor process supports on-line system management of the server configuration. One or more dispatcher processes manage all client communications context. Dispatchers route client messages to an appropriate DBI executor process through high-speed shared memory queues. Figure 12 shows a typical DBI server configuration.

[Figure 12 (DBI Server Configuration) is not available in ASCII format.]

Server Infrastructure. In the DBI server environment, an ODBC client logically connects to a service object that provides access to a specific DBI database.[1] A service is instantiated by a pool of DBI executor processes that contain the DBI image. The amount of processes of the pool is configurable, both off-line and on-line. This allows the administrator to match the throughput requirements for a given DBI database with the appropriate amount of executor processes.

Multithreading. DBI executor processes may presently be configured as session-reusable or transaction-reusable. Session-reusable means that a client is bound to an executor process for the duration of the entire database session. Transaction-reusable means that multiple clients may share the same executor process; a client is scheduled to a DBI executor for one transaction at a time.

SUMMARY

The DB Integrator product contains many features that enable it to provide open, robust, and high-performance data access. DBI guarantees open data access by supporting de facto and de jure interface standards such as SQL92 and ODBC. Client-server connectivity is available over the DECnet, TCP/IP, and SPX/IPX transports. The MDI/DDI interface allows users to extend the use of DBI to gain access to any number of data sources.

DBI provides robust data access by supporting heterogeneous query optimization, location transparency, global consistency, resolution of semantic differences, and security checks. The DBI query optimizer takes cost factors and capabilities into account to determine the optimal plan. A global catalog provides location transparency and operates as an autonomous metadata repository. Global transactions are coordinated through two-phase commit. Highly available horizontal partitioned views support continuous distributed processing in the presence of loss of connectivity. Definitions of views and stored procedures allow the user to hide semantic differences among the underlying databases. Finally, DBI enables security checks without interfering with the access controls specified in the underlying data sources.

DBI offers high-performance data access through a combination of sophisticated query optimization, advanced query execution algorithms, and efficient use of network resources. The query optimizer decomposes a distributed query by using as many features of the underlying database as possible and by employing state-of-the-art techniques such as query unnesting and partition elimination. The DBI query processor is capable of driving index joins and hybrid hash-partitioned joins. All intermediate data is cached I/O optimized. Connections to remote data sources are established solely on demand. Finally, parallel query execution is supported.

In the future, performance will continue to be an important factor for any data access product as will support for object-oriented data models. By combining data-integration technologies such as DBI with application-integration standards such as Object Request Brokers, a merger of data integration and application integration will be feasible.

ACKNOWLEDGMENTS

The authors would like to recognize everyone who contributed to the DBI project. Jim Gray, Hal Berenson, Dave Lomet, and Gayn Winters helped to establish the product vision. Russ Holden and Dan Dietterich lent their technical guidance and project leadership. The DBI engineering team designed, implemented, and delivered the product on schedule. The DBI management team of Steve Serra, Rich Bourdeau, Arlene Lacharite, and Trish Pendleton contributed their commitment to delivering the vision. We would also like to thank the anonymous referees for their invaluable comments on the content and presentation of this paper.

REFERENCES

1. Microsoft Open Database Connectivity, Programmer's Reference, Version 2.0 (Redmond, Wash.: Microsoft Corporation, 1993).

2. Information Technology--Database Language SQL, ANSI X3H2-92-154/DBL CBR-002 (New York: American National Standards Institute, 1992).

3. "Middleware: Panacea or Boondoggle?," Strategic Analysis Report (Gartner Group, July 5, 1994).

4. A. Sheth and J. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," ACM Computing Surveys, vol. 22, no. 3 (1990).

5. Digital Data Distributor Handbook (Maynard, Mass.: Digital Equipment Corporation, Order No. AA-HZ65Fl-TE, 1994).

6. Strategic Data Interface, Version 3.1 (Maynard, Mass.: Digital Equipment Corporation, 1994). This internal DB integrator specification is not available to external readers.

7. DEC Rdb Documentation Set for DEC Rdb Version 6.0 (Maynard, Mass.: Digital Equipment Corporation, 1994).

8. D. J. DeWitt and J. Gray, "Parallel Database Systems: The Future of High Performance Database Systems," Communications of the ACM, vol. 35, no. 6 (1992): 85-98.

9. Digital DSRI Handbook, Version 5.1 (Maynard, Mass.: Digital Equipment Corporation, 1994). This internal document is not available to external readers.

10. Digital Distributed Transaction Manager, OpenVMS Documentation, Version 5.5 (Maynard, Mass.: Digital Equipment Corporation, 1992).

11. Distributed Transaction Processing: The XA Specification, X/Open CAE Specification: C193, ISBN 1-872630-24-3 (1992).

12. P. Selinger et al., "Access Path Selection in a Relational Database Management System," Proceedings of the ACM SIGMOD Conference (1979).

13. P. Selinger and M. Adiba, "Access Path Selection in Distributed Database Management Systems," IBM Research Report (1980).

14. W. Kim, "On Optimizing an SQL-like Nested Query," ACM Transactions on Database Systems, vol. 7, no. 3 (1982).

15. U. Dayal, "Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates and Quantifiers," Proceedings of the 13th Conference on Very Large Databases (VLDB), Brighton (1987).

16. M. Stonebraker, "Operating System Support for Database Management Systems," Communications of the ACM, vol. 24, no. 7 (1981): 412.

17. G. Graefe, "Query Evaluation Techniques for Large Databases," ACM Computing Surveys, vol. 25, no. 2 (1993).

18. B. H. Bloom, "Space/time Tradeoffs in Hash Coding with Allowable Errors," Communications of the ACM, vol. 13, no. 7 (1970): 422-426.

19. M. Ramakrishna, "Practical Performance of Bloom Filters and Parallel Free-text Searching," Communications of the ACM, vol. 32, no. 10 (1989): 1237.

20. S. Christodoulakis, "Estimating Block Transfers and Join Sizes," Proceedings of the ACM SIGMOD Conference (1983).

BIOGRAPHIES

Richard Pledereder

As a consulting software engineer in Digital's Software Products Group, Richard Pledereder was the system architect on the DB Integrator product family and contributed to the architecture and implementation of common DBI and Rdb features such as SQL stored procedures. Richard also initiated the architecture, design, and development effort of a multithreaded database server environment, which is now part of the DBI/OSF and Rdb/OSF products. He received a B.S. and an M.S. in computer science from the Technical University Munich, Bavaria. Richard also collects tapes of operas by the Bavarian composer Richard Wagner.

Vishu Krishnamurthy

Vishu Krishnamurthy is a principal engineer in Digital's Database Integration and Interoperability Group, where he is currently the project leader for the DB Integrator product. Vishu was the technical leader for the metadata and catalog management components of DBI. Since joining Digital in 1988, he has held senior development positions in the Distributed Compiler Group, in the RdbStar project, and in the DEC Data Distributor project. Vishu holds a B.E. (honors) in mechanical engineering from the University of Madras and M.S. degrees in computer and information sciences and in mechanical engineering (robotics) from the University of Florida.

Michael Gagnon

Mike Gagnon joined Digital in 1981 and worked on the design and development of Digital's transaction processing and database systems. Mike contributed to the development of ACMS, Digital's transaction processing monitor for VMS systems, and more recently he contributed to the development of a distributed heterogeneous database system. When that system was refocused as the DB Integrator product, Mike led the team that produced the execution engine for all relational processing. Mike assumed project leadership responsibility for DBI version 1.0 and led the project through version 3.1.

Mayank Vadodaria

As a principal software engineer in Digital's Database Integration and Interoperability Group, Mayank Vadodaria was the technical group leader for the query processing components of Digital's DB Integrator product family. He was also responsible for Digital's SQL development environment products. He has been instrumental in the design of many key features in the compilation and query optimization within DBI. Mayank holds a B. Tech. from the Indian Institute of Technology, Madras, and an

M.S. in computer science from the University of Illinois at
Urbana-Champaign.

===