Data Access for the Masses through OLE DB

José A. Blakeley
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399
joseb@microsoft.com

Abstract

This paper presents an overview of OLE DB, a set of interfaces being developed at Microsoft whose goal is to enable applications to have uniform access to data stored in DBMS and non-DBMS information containers. Applications will be able to take advantage of the benefits of database technology without having to transfer data from its place of origin to a DBMS. Our approach consists of defining an open, extensible collection of interfaces that factor and encapsulate orthogonal, reusable portions of DBMS functionality. These interfaces define the boundaries of DBMS components such as record containers, query processors, and transaction coordinators that enable uniform, transactional access to data among such components. The proposed interfaces extend Microsoft's OLE/COM object services framework with database functionality, hence these interfaces are collectively referred to as OLE DB. The OLE DB functional areas include data access and updates (rowsets), query processing, schema information, notifications, transactions, security, and access to remote data. In a sense, OLE DB represents an effort to bring database technology to the masses. This paper presents an overview of the OLE DB approach and its areas of componentization.

1. Introduction

Today, a vast amount of critical information necessary to conduct day-to-day business is found outside the traditional, production corporate databases. Instead, this information is found in file systems, indexed-sequential files (e.g., Btrieve), personal databases (e.g., Access, Paradox), and productivity tools (e.g., spreadsheets, project management, email). To take advantage of the benefits of database technology such as declarative queries, transactions and security, applications must move the data from their original containing system into a DBMS. This process is expensive and redundant. Furthermore, applications want to exploit the advantages of database technology not just when accessing data within a DBMS, but also when accessing data from any other information container. This paper presents an overview of OLE DB, an ongoing Microsoft effort to define a set of OLE interfaces that provide applications with uniform access to data stored in diverse information sources. These interfaces allow a data source to share its data through the interfaces that support the amount of DBMS functionality appropriate to the data source.

1.1 Access to Diverse Sources

Consider a salesman who wants to be able to *Find all email messages he has received from Seattle customers, including their addresses, within the last two days to which he has not yet replied.* This query involves searching the mailbox file containing the salesman's email, as well as a Customers table stored in an Access DBMS to identify customers. OLE DB enables the development of an application that will access both information sources and assist the salesman to answer this query, which can be formulated in an extended SQL syntax[1] as follows:

```
SELECT m1.*, c.Address
FROM MakeTable( Mail, d:\mail\smith.mmf ) m1,
MakeTable( Access, d:\access\Enterprise.mdb, Customers) c
WHERE m1.Date >= date( today(), -2 )
AND m1.From = c.Emailaddr AND c.City = "Seattle"
AND NOT EXISTS (SELECT *
  FROM MakeTable( Mail, d:\mail\smith.mmf ) m2
  WHERE m1.MsgId = m2.InReplyTo);
```

[1] This syntax was designed for illustration purposes only. It is not implemented in any product that we are aware of.

SIGMOD '96 6/96 Montreal, Canada
© 1996 ACM 0-89791-794-4/96/0006...$3.50

We assume that `MakeTable` is a constructor function that coerces the mail file (d:\mail\smith.mmf) into a table. It also exposes the Customers table from an Access database (d:access\enterprise.mdb). The function date() takes a date and a number of days as arguments and produces a date.

In general, OLE DB attempts to make it easy for applications to access data stored in diverse DBMS and non-DBMS information sources. DBMS sources may include mainframe databases (e.g., IMS, DB2), server databases (e.g., Oracle, SQL Server), or desktop databases (e.g., Access, Paradox, Fox). Non-DBMS sources may include information stored in file systems (e.g., Windows NT, Unix), indexed-sequential files, email, spreadsheets, project management tools, and many other sources. Initial motivation and goals for OLE DB and Microsoft's component databases were outlined by Vaskevitch [10].

1.2 Component DBMSs

The benefits of component DBMSs can be discussed from two perspectives: *applications* and *data providers*. First, applications have widely varying database management needs. A decision by an application to use a particular DBMS implies a decision to use a particular storage manager, file access methods, security model, query and scripting language, query processor, and transaction manager. Often, applications do not require or use all the functionality packaged within a commercial, monolithic DBMS. Yet, they are forced to pay additional resource overhead for functionality they do not need. Second, there is a large amount of mission-critical data stored in systems that are not classified as DBMSs. Popular data access APIs, such as Open Database Connectivity (ODBC) [2], impose a high entry bar for data providers by requiring them to expose their data through SQL. This requires a non-SQL data provider to implement the equivalent of an SQL engine in the ODBC driver. OLE DB lowers the entry bar for simple tabular data providers by requiring them to implement only the functionality native to their data store. At a minimum, a provider needs to implement the interfaces necessary to expose data as tables. This opens the opportunity for the development of query processor components (e.g., SQL, geographical) that can consume tabular information from any provider that exposes its data through the OLE DB API (see Figure 1). Notice that it is also valid for SQL DBMSs to expose their functionality in a more layered manner using the OLE DB interfaces.

OLE DB addresses these problems by promoting an approach of componentizing DBMS functionality. OLE DB defines an open, extensible collection of interfaces that factor and encapsulate orthogonal, reusable portions of DBMS functionality. These interfaces define the boundaries of DBMS components such as record containers, query processors, and transaction coordinators that enable uniform, transactional access to diverse information sources. A DBMS becomes a conglomerate of cooperating components that consume and produce data through a uniform set of interfaces. The OLE DB functional areas include data access and updates (rowsets), query processing, catalog information, notifications, transactions, security, and remote data access. By defining a uniform set of interfaces to access data, OLE DB components contribute not only to advance the goals for uniform data access among diverse information sources mentioned above, but also help to reduce the application's footprint by enabling applications to use only the DBMS functionality they need. Initially, OLE DB attempts to provide uniform data access to tabular data. Future versions may provide access to richer models (e.g. object-oriented) and semi-structured data.

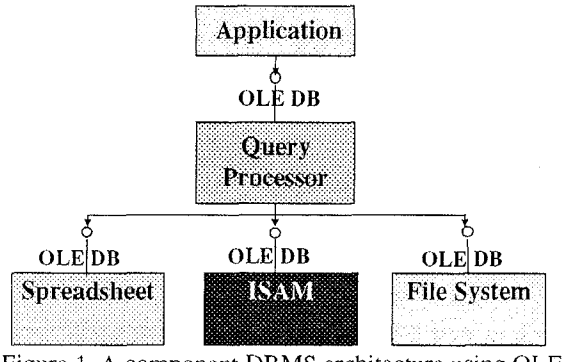

Figure 1. A component DBMS architecture using OLE DB interfaces.

OLE DB leverages the OLE Component Object Model (COM) infrastructure [7], which reduces unnecessary duplication of services and provides a higher degree of interoperability not only among diverse information sources, but also among existing programming environments and tools. Indeed, OLE DB will be the way to access data in a COM environment.

Because OLE DB extends the OLE COM services framework, Section 2 provides an overview of COM. Section 3 describes the architecture and main interface areas of OLE DB. Section 4 presents a comparison with related work, Section 5 presents conclusions and directions for future work.

2. COM Overview

This section provides a summary of key COM concepts. For a detailed discussion of the COM object services framework, the reader is referred to [7, 8].

2.1 Interface

A COM object supports a collection of interfaces. An interface is a strongly typed semantic contract between a client and an object. There is no notion of public state. Clients of a COM object have no knowledge of the internal data structures of the object; all interactions with the object are through its interfaces. COM uses the term "interface" in a sense different from that typically used in object-oriented programming with C++. In C++, interface describes all the functions a class supports. A COM interface is a predefined group of related functions that a COM object implements, but does not necessarily represent all the functions that the object supports. A COM object may support multiple interfaces. A COM object is any structure that exposes its functionality through the interface mechanism. An interface is not an object, nor an object class. Interfaces are closer to the concept of abstract classes in C++. That is, an interface is a C++ class that contains nothing but pure virtual member functions. The interface carries no implementation and only prescribes the function signatures for some other class to implement. C++ applications implement COM objects by inheriting these function signatures from one or more interfaces, overriding each interface function, and providing an implementation of each function.

2.2 IUnknown

IUnknown is a special interface that is the root of all COM interfaces. It contains three methods: QueryInterface, AddRef, and Release. QueryInterface is the method called by a client on a server object to ask whether the object supports an interface. The process of interrogating an object for the interfaces it supports through the QueryInterface method is sometimes referred to as *interface negotiation*. If the server object supports the interface, then it returns the client a pointer to it. A pointer to an interface is a pointer to an array of method pointers—a virtual function table in C++ terms—of all the methods supported by the interface. It enables clients to navigate through the multiple interfaces supported by an object. By convention, OLE prefixes the name of all interfaces with the letter "I." Also, many of the convenience functions provided by COM have the prefix "Co" (e.g., CoCreateInstance).

2.3 Explicit Reference Counting

The AddRef and Release methods represent COM's explicit reference counting mechanism to control the life-cycle of objects. Every time an object hands an interface pointer to a client, for instance as a result of a QueryInterface call, the object increases the reference count of that interface. The client must call Release on each interface pointer it no longer needs. When the reference count of an interface reaches zero, an object can free the memory occupied by the implementation of the interface.

2.4 Memory Allocation

When ownership of allocated memory is passed through an interface, COM requires that the memory be allocated with a specific "task allocator." Most general purpose access to the task allocator is provided through the IMalloc interface instance returned by the function CoGetMalloc.

2.5 Error Handling

COM interface member functions and COM library API functions use a specific convention for error codes in order to pass back to the caller both a useful return value and an indication of status or error information. For example, it is useful for a function to be capable of returning a Boolean result (true or false) as well as indicate failure or success. HRESULT is the result type used by all COM interfaces. An HRESULT is a 32-bit value divided up into four fields. One bit indicates the severity (success or error), 2 bits are reserved, 13 bits indicate the group of status codes this error belongs to (e.g., storage, dispatch, RPC), and 16 bits describe the reason for the error.

2.6 Registry

The registration database, stored in every Windows system, registers every component object available in a machine. Each component class has a unique class ID (CLSID). The entire registry database as well as each entry in the database has a hierarchical structure. The information included in each registry entry includes: the component's CLSID, a friendly name associated with the component, and the form in which the component code is packaged (e.g., as a dynamic link library or executable). OLE DB components will also include special registry keys to indicate the type of provider, and the data formats it can interpret.

2.7 Other COM Services

Other services provided by COM include structured storage, persistent intelligent names (monikers), uniform data transfer, and access to COM objects from programming languages. Each interface definition prescribes an order in which the methods must occur inside the interface virtual table; this provides the basis for binary compatibility of COM objects developed in multiple programming languages.

3. OLE DB Components

Before starting a discussion on the OLE DB components, we need to define a few terms. A *client* is any piece of system or application code that consumes an OLE DB interface; this includes OLE DB components themselves. A *provider* is any software component that exposes an OLE DB interface.

OLE DB providers can be classified broadly into two classes. A *tabular data provider* is any OLE DB provider that owns data and wishes to expose its data in a tabular form via the rowset abstraction, which is defined later in this section. Examples of tabular data providers include relational DBMSs, storage managers, spreadsheets, ISAMs, and email. A *service provider* is any OLE DB component that does not own data, but encapsulates some service by producing and consuming data through OLE DB interfaces. A service provider is both a consumer and a provider. Examples of service providers include command tree to text translators (discussed below), query parsers, query processors, transaction managers, and transaction coordinators. Data is stored in provider-specific formats such as Oracle, SQL Server, Access, Mail, or Excel and it is important to know which providers can access data in what formats. Data providers listed in the registry indicate the data formats they can read.

3.1 Connections

The OLE DB connection model defines how data and service providers are located and activated. Two objects are the basis for the OLE DB connection model: the *data source object* (DSO) and the *session* object. To access an OLE DB provider, a client must first instantiate a DSO. Each data provider is identified by a unique class identifier (CLSID) in the registry and is instantiated by calling the OLE CoCreateInstance function, which creates an instance of the object through the object's class factory. The DSO exposes the IDBInitialize interface which the client uses to provide basic authentication information such as user ID and password (for cases when the data source does not exists in an authenticated environment) as well as the name of the data source (file or database) containing the data to be accessed. Once a DSO has been successfully initialized, the DSO exposes the interfaces: (a) IDBInfo through which a client can query the capabilities of a provider. These capabilities include the interfaces, rowset properties (e.g., scrollability), transaction properties (e.g., isolation levels), SQL dialects, command operations (e.g., left-outer joins, text search operators), and security options a provider supports; and (b) IDBCreateSession through which a session object can be created. The DSO, through which a client initializes access to the data source, may expose the IDBEnumerateSources interface which returns a rowset describing all data sources accessible to the DSO. In some ways, this rowset represents a view of data sources published in the registry. The session object acts as a rowset generator, command generator, and transaction generator. A data provider that does not support commands (described below) exposes the IOpenRowset interface which enables providers to expose their data as rowsets.

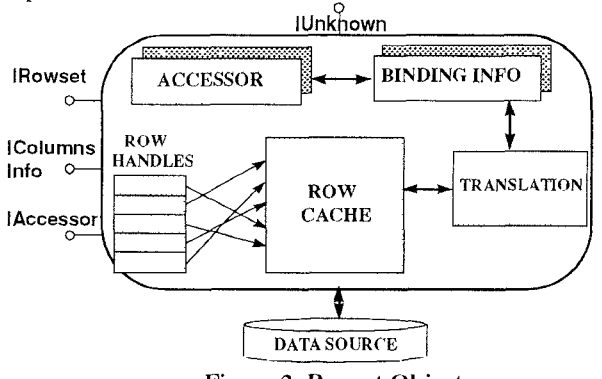

Figure 2. Rowset Object

3.2 Rowsets

Rowsets are the unifying abstraction that enables all OLE DB data providers to expose data in tabular form. Conceptually, a rowset is a multi-set of rows where each row has columns of data. Base table providers present their data in the form of rowsets. Query processors present the result of queries in the form of rowsets. This way it is possible to layer components that consume or produce data through the same abstraction. The most basic rowset object exposes three interfaces: IRowset, which contains methods for iterating through the rows in the rowset sequentially; IAccessor, which permits the definition of groups of column bindings describing the way in which tabular data is bound to client program variables; and IColumnsInfo, which provides information about the

columns of the rowset. Using IRowset, data may be traversed sequentially in a bi-directional manner depending on the properties of the rowset. Figure 2 illustrates the data structures a generic rowset object may support.

Other rowset interfaces capture rowset properties a provider may support. For example, there are interfaces to capture the ability to insert, delete, and modify rows. In addition, there are rowset interfaces that expose richer row navigation models, such as direct access and scrollability. Rowset objects are generally manufactured in one of two ways. First, they represent the result of a query for providers that support query facilities. Second, they are exposed directly as a result of an IOpenRowset::OpenRowset method call to a data source (e.g., table, index, file, in-memory structure).

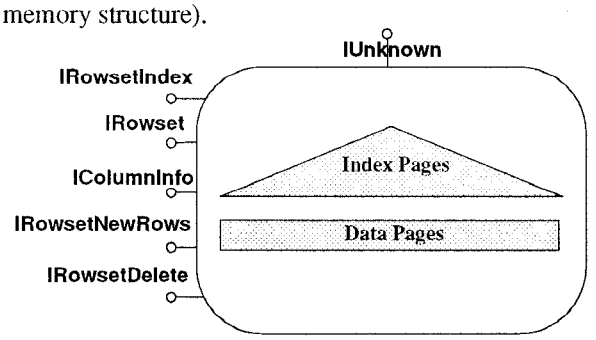

Figure 3. Index Object

Indexes (Figure 3) can be viewed as rowsets whose rows are formed from index entries with the additional property of allowing efficient access to contiguous rows within a range of keys. Indexes are OLE DB objects used primarily by query processor components. Indexes abstract the functionality of B-trees or indexed-sequential files. Indexes in OLE DB represent an interesting way in which common functionality is reused among data with different storage characteristics. Indexes are traversed using the IRowset interface; information about the index entries is obtained via the IColumnsInfo interface; and insertions and deletions are performed through the IRowsetNewRows and IRowsetDelete interfaces, respectively.

3.3 Data Binding and Accessors

Data binding describes the relationship between a field in a client structure and a rowset column value, or parameter in a stored procedure. A client uses the binding structure to communicate to the provider how it expects the data to be transferred from the provider to the client's buffer. The binding structure includes information such as the rowset column being accessed, the data type expected in the client's buffer, the offset within the client's data structure where the provider must place the data, the amount of space available for the column value in the client's buffer, and the precision and scale for values of type numeric. For columns containing OLE objects, the client can also request the interface that must be returned to the client when the object is fetched. An accessor defines a group of binding structures. It permits clients to define simultaneous access to multiple columns and enables providers to optimize access to these multiple columns.

3.4 Commands

In OLE DB, data definition (DDL) and data manipulation (DML) statements are referred to as commands. A command object encapsulates the query processing services available in today's DBMSs. Command objects expose various interfaces representing different pieces of functionality of a query processor including query formulation, validation, and execution. Figure 4 illustrates a typical OLE DB query processor.

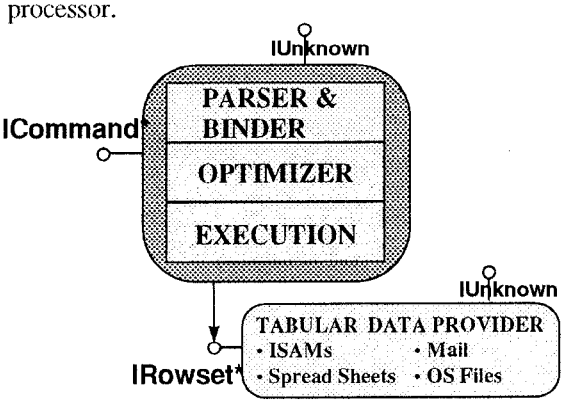

Figure 4. Command Object

The main purpose of a command object is to execute a command. Queries are commands that return results in the form of a rowset. Data definition commands are commands that do not return rowsets (e.g., create table, index, or view). Commands can be expressed in the form of text (typically ANSI SQL-92), or in the form of a command tree. A command tree, which is a mixture of parse tree and algebra expression, is a data structure whose nodes represent operators that consume and produce data. The types of data manipulated by these operators include table (ordered, unique, hierarchical), scalar (constants, row), and Boolean.

Command objects also expose other functionality available in a query processor such as the ability to

validate a command syntactically or semantically, formulate parameterized queries, define optimization goals or resource consumption limits during execution, and request properties and interfaces on the rowsets resulting from commands.

An OLE DB client wanting to use the services of a command object typically follows the following steps: (a) obtain an interface to the command object, typically ICommandText, (b) build a text string representing the request, (c) pass the text to the command object, (d) request optimization goals or specify resource consumption limits during execution (cost limits), (e) request properties to be supported by the resulting rowset objects including interfaces they will expose, and (f) execute the command. The successful execution of a command results in a rowset or an array of rowsets representing a hierarchical result (linked tables). The user can then use the methods of the rowset to navigate the contents of the rowset as required.

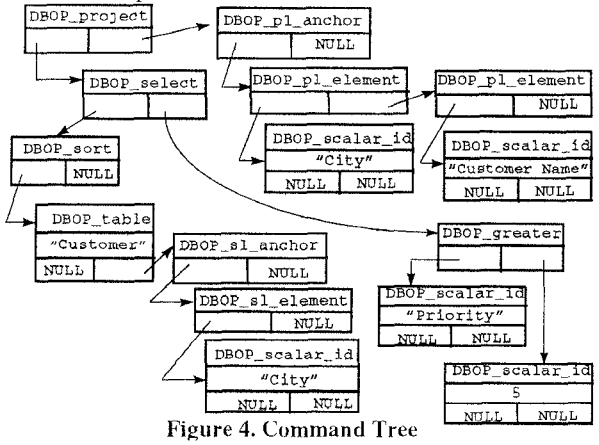

Figure 4. Command Tree

Notice that because command objects consume and produce rowsets, it is possible to compose query processors to process distributed, heterogeneous, or parallel queries. It is also possible to compose specialized query processors (e.g., SQL query processor, text-search query processors, geographical or image query processors). Clients can also formulate queries in the form of command trees. There are two advantages for using command trees as a way to express queries: (a) they provide a single model for applications and tools (e.g., QBE-like design tools) to express queries; (b) command trees are more extensible than text (it is easier to extend the capabilities of a provider by adding a new operator node than to extend the syntax of text). As an example, the text query "`SELECT City, CustomerName FROM Customer WHERE Priority > 5 ORDER BY City;`" can be equivalently expressed by a command tree as shown in Figure 5.

3.4.1 Rowset Properties

During query formulation, OLE DB clients can request certain properties to be satisfied by the rowsets resulting from a query. Common properties include the set of interfaces to be supported by the resulting rowset objects. Any rowset returned from a query exposes the mandatory interfaces IRowset, IAccessor, and IColumnsInfo described earlier. The basic IRowset interface supported by all rowsets allows clients to, at minimum, navigate the rowset in a forward-only manner. Requesting the "bi-directional" rowset property enables a client to navigate the rowset in both directions. Notice that supporting bi-directional rowsets puts additional requirements on the creation of the rowset object to support this functionality efficiently. A forward-only rowset can be fed directly from the query execution plan output. A bi-directional rowset may require materializing the result. In addition to the default interfaces, a client may request interfaces to enable direct positioning within a rowset via bookmarks (IRowsetLocate), scrollability (IRowset Scroll), immediate updatability of rows (IRowsetNew Rows, IRowsetDelete, IRowsetChange), deferred (or batch) updatability of rows (IRowsetUpdate), asynchronous population of the rowset (IRowset Asynch), and the ability to lock individual rows overriding the isolation level of the transaction within which the rowset is accessed (IRowsetLockRows). It is also possible to request additional properties that specialize the behavior of certain interfaces, for example, to make some columns of the rowset updatable and the rest read-only.

3.4.2 Optimization Goals and Execution Limits

As part of the query formulation, OLE DB clients can also specify certain optimization and execution goals to be met by a query processor component. Optimization goals enable clients to specify the critical resources for which it is important to optimize. For example, a client application running on a TV set top box having limited main memory may wish to optimize for low memory consumption at the expense of network messages. A client running on a laptop may wish to optimize battery consumption by suggesting the query processor to deliver the first feasible execution plan obtained within 10 ms of optimization time.

Execution limits enable clients to request that the query processor not spend execution resources beyond a certain point. In a sense, execution limits act as "query execution governors." A client may order the query processor to stop execution after reaching 10 seconds of CPU time, or after computing the 100th result row, or after having transmitted 100 KB of result data over a network. Once the query processor reaches the execution limit it suspends execution of the query; the client may then choose to stop, abort, or resume execution.

The resources for which optimization goals or execution limits may be specified include: CPU, memory, I/O, or network messages. The units include time, number of bytes, number of rows, and so on. Both lists are extensible, enabling OLE DB providers to define their own provider-specific resources or units (e.g., battery consumption).

3.5 Schema Information

OLE DB clients obtain catalog information through the IDBSchemaRowset interface. This interface enables clients to request information about types, tables, columns, indexes, views, triggers, assertions and constraints, statistics, character sets, collations, and domains, as well as some other relationships among these (e.g., constraints and tables, constraints and columns, columns and domains). All catalog information is presented to the client through rowsets. The schemas of these catalog rowsets are based on the ANSI SQL 92 standard. We will not describe the contents of each schema rowset further in this paper.

3.6 Transactions

OLE DB defines a basic set of interfaces for transaction management. A transaction enables clients to define units of work within a provider. These units of work have the atomicity, concurrency, isolation, and durability (ACID) properties [3]. They allow the specification of various isolation levels and optimistic concurrency control policies to allow more flexible access to data among concurrent clients.

3.6.1 Local Transactions

Local transactions refer to transactions running in the context of a resource manager. A provider that supports transactions exposes the ITransactionLocal interface on the session object. A call to the ITransactionLocal::StartTransaction method begins a transaction on the session object. A session object may be inside or outside of a transaction at any point in time. When created, a session is outside of a transaction and all the work done under the scope of the session is immediately committed on each OLE DB method call. When a session object enters a local or coordinated transaction, all the work done under the session, between the StartTransaction and Commit (or Abort) method calls, including other objects created underneath it (commands or rowsets), are part of the transaction. The StartTransaction method supports a rich set of options defining various isolation levels, concurrency control policies, and resource retention behavior (after commit) that clients may request when creating a transaction. OLE DB providers do not need to support all possible transaction options defined. A client can interrogate the transaction capabilities of a provider through the IDBInfo interface.

For providers that support nested transactions, calling ITransactionLocal::StartTransaction with an existing transaction begins a new nested transaction below the current transaction. Calling ::Commit or ::Abort methods on ITransactionLocal commits or aborts the transaction at the lowest level, respectively.

3.6.2 Distributed Transactions

OLE Transactions, another effort within Microsoft which is separate from, but synergistic with OLE DB, defines a model and interfaces for coordinating transactions among multiple (possibly distributed) data providers. Distributed transactions are not discussed further in this paper. Complete details of the distributed transaction coordinator as implemented in the SQL Server product V6.5 are given in [6].

3.7 Notifications

OLE DB uses a well known "OLE controls" notifications model to allow notifications among OLE DB components and clients. Currently, OLE DB defines two notification models: *local* notifications and *data source* notifications.

Local notifications are used by groups of cooperating clients sharing a rowset. All clients and the rowset-generating action are assumed to be working within the **same** transaction. Local notifications enable cooperating clients sharing a rowset to be informed about actions on the rowset performed by their peers. For example, consider a form containing two data controls (widgets); one displaying data in a grid and the other displaying data as a two dimensional histogram. Both controls receive their data from the same rowset, and operate individually as clients of the rowset object. An end-user may communicate an

update to the rowset through the grid control. Local notifications will enable the other control to be informed of these changes as they occur giving it the opportunity to update the histogram as appropriate, ultimately providing a consistent view of the data within the form. Clients register their interest in receiving local notifications though the IConnectionPointCointainer (ICPC) interface. Clients of local notifications support the IRowsetNotify interface, which is called by the rowset object upon occurrence of interesting events.

Data source notifications (watches) are designed to enable clients to be notified about changes to the underlying data source originated by other concurrent clients running under **different** transaction contexts in either read committed or read repeatable isolation levels. Data source notifications under read uncommitted or serializable isolation levels do not make sense; the former leads to unpredictable results, the latter by definition does not allow updates to data being read by other concurrent transactions. Upon notification, the client can request a list of changes from the data provider. An interesting application of data source notifications is the support of replicated data or materialized views. The client code may represent an incremental refresh algorithm that is triggered by the notification. The client code requests the list of changes from the data source (the deltas). Upon arrival of the list of changes, the client code can compute an incremental algorithm to bring the replica or materialized view up to date. Clients register their interest in receiving data source notifications though the IConnectionPointCointainer interface. Clients of data source notifications support the IRowsetWatchNotify interface which is called by the rowset object upon occurrence of changes to the underlying data source.

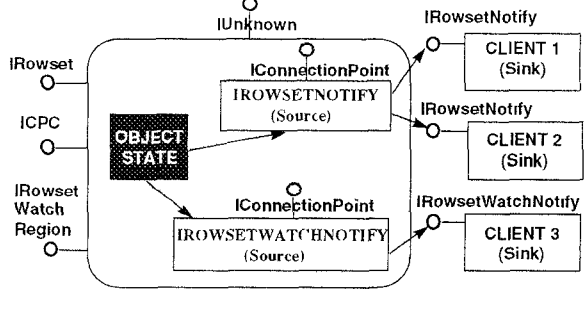

Figure 6. Rowset supporting notifications.

Notifications in OLE DB define a basic mechanism on which to implement active database behavior. IRowsetNotify and IRowsetWatchNotify represent two, among many, possible notification contracts between sources and sinks. Providers may define other notification contracts as needed. Figure 6 illustrates typical plumbing between a rowset supporting the two types of notifications and three clients. Clients 1 and 2 receive local notifications, Client 3 receives data source notifications.

3.8 Remote Data Access

OLE DB enables efficient and transparent access between *clients* and *providers* across process and machine boundaries. In the following discussion, the term "process boundary" refers to a process or machine boundary. We distinguish two forms of remoting: *data* remoting and *object* (or *interface*) remoting. In data remoting, a provider driver exposing OLE DB interfaces is present on the client site. All interaction with the provider is through the driver. All details about communication protocol, data transfer format, and the provider's native interface calls are handled internally by the driver. An example of this approach would be an OLE DB provider accessing a SQL DBMS via ODBC. In other words, the process or machine boundary cuts through the provider's driver. The client/provider relationship is entirely within a process.

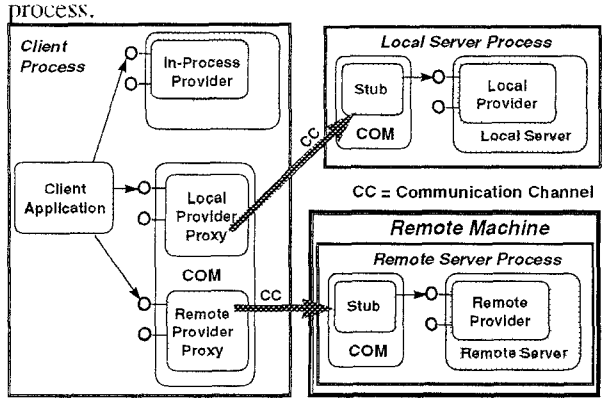

Figure 7. Clients always call in-process code; objects are always called by in-process code.

In interface remoting, an OLE DB provider is running remotely in a different process. Interface remoting provides the infrastructure and mechanisms to allow a method invocation to return an interface pointer to an object created in a different process. The infrastructure that performs the remoting of interfaces is transparent to both the client and the server object. Neither the client or server object are aware that the other party is in a different process. Network OLE, another COM technology leveraged by OLE DB, provides an open specification and reference implementation of such an infrastructure. Figure 7 illustrates the various configurations in which client and provider code may exist.

The proxy/stub agents are agents that take care of marshaling arguments across the process boundary, and optimize access over the network by trying to hide the latency and bandwidth. The design of proxies and stubs is an open-ended optimization problem. Proxies and stubs may vary in complexity. Simple proxy/stub pairs provide marshaling of arguments to method calls. Sophisticated proxy/stubs may provide client-side caching combined with bulk data transfer over the process boundary to minimize network trips, or may even include boxcarring of method calls [3], thereby optimizing network trips across multiple interface method calls.

Providers may choose to implement their remote data access by either approach. An OLE DB Software Development Kit (SDK) will provide reference implementations of data and interface remoting. Data remoting will be offered as an OLE DB provider over ODBC where all process and network communications are managed internally by the ODBC driver. Interface remoting will be offered as a library of generic proxy/stub pairs for all OLE DB objects. Providers wishing to have a tighter control over the performance of their OLE DB implementations across process boundaries are free to implement their own remoting agents using the network protocol or optimizations of their choice.

3.9 Security

OLE DB defines interfaces that support authentication, authorization, and secure administration of access among users, groups, and resources. The role of security in OLE DB is to: (a) expose the relevant security functionality encapsulated in the security models of individual data providers or the underlying operating system on which they run, (b) provide a pass-through layer of security interfaces through which the underlying security functionality can be made available to OLE DB clients, and (c) to provide a unified view of security.

3.9.1 Authentication

OLE DB allows authentication through different mechanisms depending on the layer that enforces authentication: operating system, network, or data provider. In *password-based* authentication, the client authenticates itself to the data provider by supplying a name and password. The data provider enforces authentication in this case. When the client and provider operate in-process, passwords can be passed directly to the provider. In situations where a network connection is involved, then the password must be sent to the provider over the network, which presents additional security problems. *Domain-based* authentication implies the availability of an authentication service provided by the operating system (e.g., Windows NT). In this environment, users authenticate themselves to the domain by providing a password when logging into the system. Once the user is authenticated to the domain, the domain provides identification information on behalf of the user in a trusted manner. *Distributed* authentication assumes the existence of a distributed authentication service such as the one provided by Microsoft's Security Support Provider Interface (SSPI) which is modeled after DEC's General Security Support API. In OLE DB, clients call the IDBInitialize interface to request the type of authentication mechanism they wish to use. They can also request the quality of service they expect when communicating to the provider across a network.

3.9.2 Authorization

Authorization, or access control, is concerned with the enforcement of the privileges defined among users, groups, roles, services, etc. In most cases, authorization is enforced internally by the provider and the client only gets return codes from method invocations determining if the call was successful or a permission violation occurred. OLE DB leverages a set of security interfaces defined by Network OLE to control security of RPC calls between proxies and stubs, the launching of objects, and impersonation of clients by servers.

3.9.3 Administration

Security administration is concerned with managing the granting and revoking of permissions among users, groups, roles, and any provider-specific object (data or code). Data providers supporting commands (e.g., ICommandText) process security administration requests formulated by clients using SQL GRANT and REVOKE statements. Data providers not supporting commands may expose the IDBAccessControl interface which contains methods to get and set permissions among objects, users, and groups. IDBAccessControl is defined for cases where multiple providers, each supporting its own security enforcement mechanisms, are composed or layered. Since each data provider has its own domain space of privileges, users, and objects, OLE DB provides an extensible mechanism to model the difference in models where each type of privilege, user, and object type is defined by a global unique identifier (GUID). In addition, OLE DB defines the IDBTrusteeAdmin interface defined for the administration of trustees. A

trustee may be a user, group, role, agent, or service. The IDBTrusteeAdmin interface defines methods for creating and deleting trustees, populate and test membership in trustee collections, query trustee permissions, and manage trustee affiliations to groups or roles. We decided to add this interface since there was no standard interface (e.g., SQL) to manage trustees in a system.

4. Related Work

OLE DB is related to previous work on extensible DBMS architectures. Genesis [1] was an early effort to factor areas of independently reusable functionality within a DBMS. It contains components defining query language parsing for SQL and QUEL, file access methods, recovery, and concurrency control policies. Genesis provides a compiler-compiler DBMS generation tool that enables a database designer to configure a DBMS from high-level specifications. The tool produces code implementing a DBMS by mixing and matching the various components available in the tool. While Genesis represents a significant achievement in the area of component DBMSs, it has some shortcomings. The interfaces are not open for third parties to reuse or extend; they are only known within the Genesis DBMS generator tool, and the generated components cannot be used outside the tool since its output is a single, monolithic piece of code.

The Open OODB project at Texas Instruments [11] developed an open, extensible architecture for constructing a family of OODBs. The architecture consists of two kinds of services. First, the meta-architecture service houses a collection of support modules providing abstractions for typeless object containers (address space managers), object translation, type repository, and communications. The meta-architecture also includes an infrastructure for defining events, event monitors (sentries), and actions to be called as a result of those events. Sentries represent the mechanism that allows services, built on top of the support modules, to be plugged into the Open OODB framework. Second, the architecture includes an extensible collection of services that implement policies (called policy managers) in various functionality dimensions. There are policy managers for persistence, transactions, queries, remote data access, and change management. The Open OODB project is similar to the componentization goals of OLE DB, but it is different in that it supports object models based on the type systems of C++ and CLOS which are richer than OLE DB's tabular model. However, the Open OODB project does not attempt to address the issues in providing uniform access to diverse information sources.

OLE DB is also related to Open Database Connectivity (ODBC) [2]. They are similar in that they are both APIs designed to provide access to a wide range of data providers. OLE DB and ODBC are synergistic rather than competing technologies. ODBC is an API designed to provide access to SQL data in a multi-platform environment. OLE DB is designed to provide access to data in a COM environment. OLE DB subsumes all the functionality of ODBC, but it also adds new features such as a richer and sharable cursor model via rowsets, notifications, watches, and command trees. A subset profile of OLE DB provides functionality equivalent to ODBC. ODBC requires all providers to expose an SQL model of data. To comply with ODBC, non-SQL providers have to build functionality equivalent to an SQL engine in their drivers to share their data. In contrast, OLE DB factors various layers of DBMS functionality providing interfaces for each layer. As a result data providers need to expose only the OLE DB interfaces that reflect the DBMS functionality they natively implement. For example, a tabular data provider must support rowsets and can ignore the command interfaces. Because ODBC is now an ANSI/ISO standard, we expect that ODBC will continue to be the preferred way to access standard SQL data.

OMG's Object Services Architecture [9] consists of an object request broker and a collection of services that communicate through the broker. Some of the object services include persistence, transactions, naming, queries, relationships, events, and security. The OMG object services architecture is more directly related to OLE/COM. OLE DB is the collection of data access services for COM.

Starburst uses table functions, which are a kind of user-defined functions, as a mechanism to import data residing outside the database and to present it as a virtual table [5]. These virtual tables can be filtered, joined with other tables, and similarly manipulated by the full power of SQL without actually having to store that data in the database. In Starburst, the code implementing any user-defined function must be linked with the rest of the Starburst system. It is not clear from the description in [5] what information the implementor of a table function must expose to allow the resulting function tables to be regarded as regular tables. OLE DB aims at making all these interfaces open and does not require linking a provider's

implementation of a rowset with a query processor component.

5. Conclusions

OLE DB aims to increase the level of interoperability among diverse tabular information sources by providing a uniform abstraction (rowsets) to view this data. It provides an approach to developing a component DBMS technology by defining a collection of open, extensible interfaces describing orthogonal pieces of DBMS functionality, and defining how to build components supporting these interfaces. We presented an overview of the interface areas. The current goal for OLE DB is to provide uniform data access to tabular data providers. Future work includes expanding the access to non-tabular data (e.g., complex objects and self-describing data such as in HTML and SGML documents).

References

1. D.S. Batory, et al., *"Genesis: An Extensible Database Management System,"* IEEE Trans. Software Eng., Vol. 14, No. 11, Nov. 1988, pp. 1,711-1,730.
2. K. Geiger. *"Inside ODBC,"* Microsoft Press, 1995.
3. J. Gray and A. Reuter. *"Transaction Processing: Concepts and Techniques,"* Morgan Kaufmann, 1995.
4. D. Haught and J. Ferguson. *"Microsoft Jet Database Engine Programmer's Guide,"* Microsoft Press, 1996.
5. G.M. Lohman et al. *"Extensions to Starburst: Objects, Types, Functions, and Rules,"* Communications of the ACM, Vol. 34, No. 10, Oct. 1991, pp. 94-109.
6. Microsoft, *"Guide to Microsoft Distributed Transaction Coordinator,"* Microsoft SQL Server, Version 6.5, 1996.
7. Microsoft, *"OLE 2 Programmer's Reference,"* Vols. 1-2, Microsoft Press, 1994.
8. Microsoft, *"The Component Object Model Specification,"* 1996, Microsoft Development Library, CD-14, January, 1996.
9. OMG, *"The Common Object Request Broker: Architecture and Specification,"* The Object Management Group, 1992.
10. D. Vaskevitch. *"Database in Crisis and Transition: A Technical Agenda for the Year 2001,"* Proc. of the ACM Sigmod 1994 Conf., pp. 484-489.
11. D.L. Wells, J.A. Blakeley, C.W. Thompson. *"Architecture of an Open Object-Oriented Database Management System,"* Computer, Vol. 25, No. 10, Oct. 1992, pp. 74-82.

Appendix. A Rowset Traversal Example

The purpose of this appendix is to illustrate a simple OLE DB program that reads a table through a rowset, printing each row along the way. Familiarity with C++, and OLE is assumed. Code to check status and errors after each OLE call is omitted for clarity. The OLE DB interfaces are targeted to DBMS developers, tool builders, and independent system vendors. It is not an interface for application programmers. Bindings to high-level languages like C++ or Visual Basic will be done through higher-level object models such as Data Access Objects (DAO) [4].

```
#include oledb.h
int main()
{
// Define all variables used in the program, not shown
...
// Initialize OLE
CoInitialize(NULL);
 // Instantiate a DSO for an email data provider
CoCreateInstance(CLSID_MailProvider,0,CLSCTX_LOCAL_SERVER,
                 IID_IDBInitialize, &pIDBInit);
// Initialize the DSO for email data provider
pIDBInit->Initialize( "smith", "password", "c:\mail\smith.mmf" );
// Request the IDBCreateSession interface
pIDBInit->QueryInterface(IID_IDBCreateSession, &pIDBCS);
// Create a session returning a pointer to its IOpenRowset interface
pIDBCS->CreateSession(NULL,IID_IOpenRowset,&pIOpenRowset);
```

```
// Open a rowset corresponding to the email file
pIOpenRowset->OpenRowset(NULL,&TableID,0,NULL,IID_IRowset,cPropertyErrors,
                         &pIRowset );
// Get information about column types
pIRowset->QueryInterface(IID_IColumnsInfo, &pIColsInfo);
pIColsInfo->GetColumnInfo(&cCol, &rgColInfo, &pzColNames);
pIColsInfo->Release();
// Establish bindings using a convenience function
CreateBindingsFromInfo(rgColInfo,&cBindings, &rgBindings, &rgData );
// Request the IAccessor interface from rowset
pIRowset->QueryInterface( IID_IAccessor,&pIAccessor );
// Create an accessor, return an accessor handler
pIAccessor->CreateAccessor(DBBINDIO_READ,cBindings,rgBindings,&rgErrors,&hAccessor
);
// Read the rows; 100 rows at a time into the rowset cache
while(SUCCEEDED(hr=pIRowset->GetNextRows(0,NULL,0,100,&cRowsObtained,&rghRows))
      && cRowsObtained)
{
    for( iRow = 0; iRow < cRowsObtained; iRow++ )
    {
    // GetData copies the rows into the local buffers, performing the type conversions
    // specified in the binding structures associated with the accessor
        pIRowset->GetData(rghRows[iRow],hAccessor,rgData );
    // Convenience function to print the data
        PrintData( rgData );
    }
    // Release the rows just printed from the rowset cache.
    pIRowset->ReleaseRows(cRowsObtained,rghRows,NULL,NULL );
}
// Release the accessor handle and the rowset
pIAccessor->ReleaseAccessor(hAccessor);
pIAccessor->Release();
pIRowset->Release();
// Release the session and DSO objects
pIDBC->Release();
pIDBInit->Release();
// Uninitialize OLE
CoUninitialize();
return;
```