# SQL/MED — A Status Report

Jim Melton
Oracle, Sandy, UT 84093
jim.melton@acm.org

Jan-Eike Michels
Vanja Josifovski
Krishna Kulkarni
Peter Schwarz
{janeike, vanja, krishnak}@us.ibm.com
schwarz@almaden.ibm.com

## Guest Column Introduction

In March, 2001, we delivered a (partly) guested column covering the topic of Management of External Data [1]. The column you are reading right now reports on the on-going development of the SQL/MED standard and is authored by all but one of the authors of that earlier column.

We trust that our readers will benefit from this update on an interesting and important part of SQL.

*Jim Melton and Andrew Eisenberg*

## Introduction

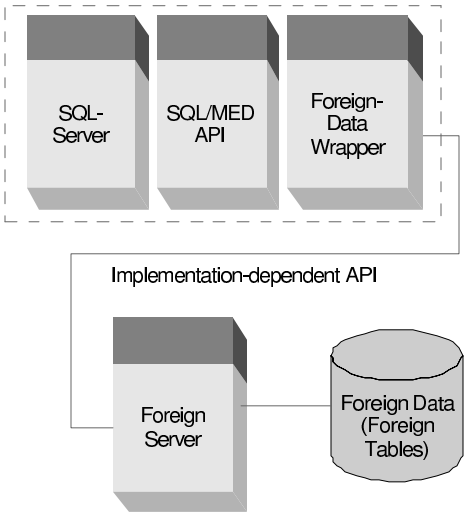
As discussed in [1], a new part of the SQL standard, known as SQL/MED came into existence in early 2001. (MED stands for "Management of External Data".) SQL/MED offers syntax extensions to SQL as well as a set of routines for use in developing and managing applications that access both SQL data and non-SQL (also known as external) data.

SQL/MED specifications can be divided into two broad parts. The first part, called the *wrapper interface*, offers facilities to view external data managed by one or more external sources (formally known as *foreign servers*) simply as a set of SQL tables (formally known as *foreign tables*). External data may be stored in file systems, in HTML-formatted web pages, in XML documents, or in some other specialized repositories. The second part of SQL/MED, called *datalinks*, offers facilities to let an SQL-server control the management of referential integrity, recovery, and authorization of data residing in one or more file systems.

The wrapper interface provides the ability to use the SQL interface to access non-SQL data and, if desired, to join that data with SQL data. An application issuing an SQL query to an SQL-server supporting the wrapper interface can reference both tables managed by that SQL-server and foreign tables known to that SQL-server. The SQL-server is responsible for decomposing such a query into multiple fragments, connecting to one or more software entities (formally known as *foreign-data wrappers*) that interface with the foreign servers responsible for managing the data that corresponds to foreign tables referenced in the query, devising an execution plan for each fragment, initiating the execution of those plans, receiving the result data from each of the foreign-data wrappers, and finally completing the query execution and returning the result to the application.

The interaction between the SQL-server and a foreign-data wrapper is based on a request/reply paradigm. The SQL-server builds a request representing the query fragment. The foreign-data wrapper analyzes the request and returns a reply that describes that portion of the request that can be handled by the foreign server. The SQL-server must compensate for any part of the query fragment that cannot be executed by a foreign server.

**Figure 1 — Components of the Wrapper Interface**

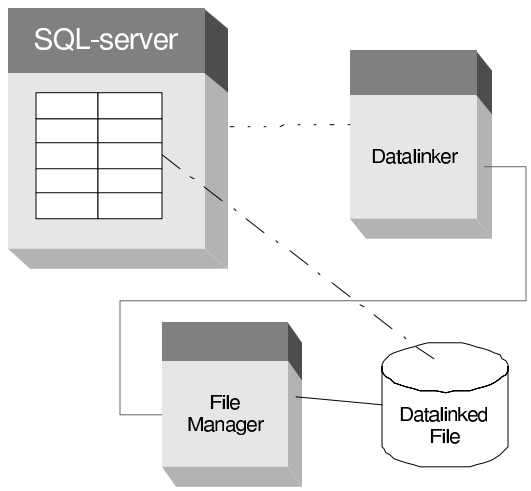



The wrapper interface illustrated in Figure 1 includes SQL extensions for defining foreign servers, foreign tables, and foreign-data wrappers and two distinct sets of routines: *foreign-data wrapper interface SQL-server routines* and *foreign-data wrapper interface wrapper routines* (together commonly called *foreign-data wrapper interface routines*). While an SQL-server conformant to SQL/MED must implement

the SQL extensions and the foreign-data wrapper interface SQL-server routines, a foreign-data wrapper conformant to SQL/MED must implement the foreign-data wrapper interface wrapper routines. Figure 1 (which originally appeared in a slightly different form in [1]) depicts the relationships among SQL-servers, foreign-data wrappers, and foreign servers.

Datalinks are useful for applications that require referential integrity, recovery, and authorization mechanisms, typically provided by database management systems, for the data stored in external files without the need to store their contents directly in the database. Applications using datalinks are expected to access the data stored in files using the native interface of file systems rather than via SQL. This part of the specification depends on a software entity (formally called a *datalinker*) that interfaces to a file system. The datalinks part of the specification consists of an SQL built-in data type, DATALINK, and a set of built-in operators to operate on values of DATALINK type. Figure 2 (which also originally appeared in a slightly different form in [1]) depicts the relationships among the SQL-server, the datalinker, a file system and a "datalinked" file[1].

**Figure 2 — Datalink-related relationships**




An extensive description of both the wrapper interface and datalinks specification as standardized in [2] can be found in [1].

While the initial version of the SQL/MED specification published in early 2001 provided a fairly complete set of facilities to access and manage external data, additional work by the standardization committees in the last two years promises a much richer standard. In fact, [1] hinted at the limitations of the initial version and possible extensions to the specification to overcome those limitations. The primary focus of this article is to explain these new extensions. Though this new version is not published yet (it is currently in the Final Committee Draft, or FCD, stage of the ISO standardization process and is expected to be published in early 2003), no significant changes are expected to occur between now and its eventual publication date. In the remainder of this paper, we refer to the SQL/MED specification published in 2001 [2] as the *previous version of SQL/MED* and the SQL/MED specification currently in FCD stage [3] as the *current version of SQL/MED*.

We describe the enhancements to the wrapper interface in next section and the extensions to datalinks in the section titled "Enhancements to Datalinks".

## Enhancements to the Wrapper Interface

As described in [1], communication between an SQL-server and a foreign-data wrapper can occur in either two modes: decomposition mode or pass-through mode. Further, in decomposition mode, a query is broken into multiple fragments by the SQL-server, each to be executed by a particular foreign server. The interaction between an SQL-server and a foreign-data wrapper in decomposition mode occurs in two phases: planning phase and execution phase. During the planning phase, the SQL-server and a foreign-data wrapper cooperatively produce an execution plan for a given query fragment, while during the execution phase the agreed-upon plan is executed by each of the foreign-data wrappers and the resulting data is returned to the SQL-server.

There are essentially three major enhancements to the wrapper interface in the current version of SQL/MED:

1. The ability for an SQL-server to communicate complex query requests to foreign-data wrappers.
2. The ability for an SQL-server to communicate the query context (that is, information to identify requests belonging to the same query) to foreign-data wrappers.
3. The ability for an SQL-server to ask foreign-data wrappers for query execution costs.

All these extensions affect the query-planning phase in decomposition mode. We describe below each of these enhancements in detail.

The examples used in the following sections use the simple schema, containing two foreign tables, managed by a single foreign server, shown in Example 1.

[1] A "datalinked" file is a file that is referenced by a value stored in a DATALINK column.

**Example 1 — Sample schema**

```
EMP  (name      VARCHAR(16),
      street    VARCHAR(30),
      phone     VARCHAR(10),
      city_id   INTEGER,
      resume    VARCHAR(32000) );

CITY (id            INTEGER,
      longitude     FLOAT,
      latitude      FLOAT,
      name          VARCHAR(30),
      state_name    VARCHAR(30) );
```

## *Communicating complex query requests*

With the wrapper interface specified in the previous version of SQL/MED, an SQL-server was limited to communicate query requests of the form SELECT <column list> FROM *FTN*, where *FTN* is the name of a foreign table and each element of <column list> refers to a column of that table. In this scenario, a foreign server was assumed to be primarily a data source. However, there are many cases where a foreign server can provide computational power as well as being a source of data. A foreign server can provide either a subset of the functionality of an SQL-server or the functionality of a foreign server can overlap the functionality of an SQL-server. In these cases, it may often be more efficient to pass complex queries for execution by foreign servers. The wrapper interface in the current version of SQL/MED is enhanced in a number of ways to take advantage of the features supported by foreign servers.

### Queries containing WHERE clauses

The wrapper interface in the previous version of SQL/MED does not allow an SQL-server to communicate queries containing a WHERE clause to foreign-data wrappers. The wrapper interface in the current version is enhanced with four new routines to deal with the WHERE clause and its Boolean expressions. Example 2 illustrates the use of such a mechanism:

**Example 2 — Query with WHERE clause**

```
SELECT resume
FROM EMP
WHERE name = 'John Doe';
```

The advantage of letting the foreign server execute the predicates is pretty obvious. In the above example, the foreign-data wrapper can return a single resume to the requesting SQL-server instead of returning the resumes and the names of all the employees for the SQL-server to perform the selection. This will reduce the amount of data sent to the SQL-server tremendously, leading to better performance. On the other hand, if the foreign server is unable to perform the selection, the foreign-data wrapper will exclude the predicate from the reply, signaling to the SQL-server that it must compensate by applying the predicate in the SQL engine.

In addition to better performance, the ability to pass queries containing predicates can actually make a larger number of foreign servers accessible. For example, some foreign servers may not be able to provide their data without a unique key. Such foreign servers cannot be made accessible using the wrapper interface of the previous version.

### Queries containing multiple table references in FROM clauses

Though the wrapper interface in the previous version of SQL/MED could deal with multiple table references in the FROM clause, the previous version restricted the queries flowing from an SQL-server to a foreign-data wrapper to contain exactly one table reference in the FROM clause. That restriction has now been lifted, so queries containing multiple table references in the FROM clause can be communicated to individual foreign-data wrappers. Example 3 illustrates the use of such a mechanism:

**Example 3 — Multi-table query**

```
SELECT CITY.id, CITY.latitude,
       CITY.longitude
FROM EMP, CITY
WHERE EMP.name = 'Jane Doe' and
      EMP.city_id = CITY.id;
```

Using the wrapper interface in the previous version of SQL/MED, the SQL-server would have needed to transmit two query fragments to the foreign-data wrapper, each retrieving rows from one of the foreign tables used in the query. With the enhanced interface supported in the current version of SQL/MED, the entire query can be sent in a single request, assuming the foreign server is able to perform joins and predicate evaluations. If a foreign server is unable to perform the join, it will indicate so by excluding one table reference and the join predicate from the reply. Here again, the SQL-server must compensate for the capability that the foreign server lacks and perform the join after retrieving the necessary data from the foreign server.

### Queries containing complex value expressions in SELECT and WHERE clauses

The wrapper interface in the previous version of SQL/MED limits the value expressions contained in the SELECT clause of the queries flowing from an SQL-server to foreign-data wrappers to column references only. The wrapper interface in the current version of SQL/MED is enhanced to allow complex value expressions both in the SELECT and the WHERE clause of the queries. Value expressions can now be column references, constants, or parameters, all connected by one or more operators[2]. Example 4 illustrates the use of such a mechanism:

**Example 4 — Query with operators**

```
SELECT name || ' ' || phone
FROM EMP
WHERE name = 'John Doe';
```

Conceptually, value expressions are modeled as *typed operator trees*. Internal nodes of the trees are operators, while the leaf nodes represent column references, constants, or parameters. (A parameter value expression is useful for performing nested loop joins with the inner table being a foreign table.) Each operator node has a set of child nodes (sub-expressions). The wrapper interface is enhanced with six additional routines that are used to traverse the operator trees.

In the current version of SQL/MED, the foreign-data wrapper must either agree to evaluate a value expression as a whole, or to reject it. For example, if the foreign-data wrapper was asked to evaluate an expression such as "(C1+C2)*C3" (C1, C2, and C3 all being columns of a foreign table), it may either commit to evaluate the whole expression or indicate that it cannot evaluate the expression at all, but it is not allowed to commit to the evaluation of just (C1+C2).

### Queries containing user-defined function invocations in value expressions

It is often possible for a foreign server to offer functionality that is similar to that provided by an SQL-server. In particular, it is possible for a foreign server to offer an executable function that is equivalent to a user-defined SQL-invoked function that exists at an SQL-server. In some cases, it may be more efficient to execute the function at the foreign server rather than at the SQL-server. The current version of SQL/MED offers a mechanism called "routine mapping" to enable SQL-servers to choose the best option for executing a specific function referenced in the query.

A routine mapping associates an SQL-invoked function with a routine at a foreign server. Generic options provide the necessary information for the foreign-data wrapper to identify the function when it encounters it during query processing.

Routine mappings are created at an SQL-server via a CREATE ROUTINE MAPPING statement, which is given the signature of an SQL-invoked function that exists at the SQL-server and the name of a foreign server, as shown in Example 5.

**Example 5 — Creating a routine mapping**

```
CREATE ROUTINE MAPPING FN1_AT_FS1
  FOR SCH.FUN1(VARCHAR, INTEGER)
  SERVER FS1
  OPTIONS (REMOTE_NAME 'FN1',
           REMOTE_SCHEMA 'TEST')
```

Example 6 illustrates the use of a mapped function:

**Example 6 — Using routine mappings**

```
SELECT resume
FROM EMP
WHERE fun1(name, city_id) = 100;
```

The value expression corresponding to the function invocation in the WHERE clause carries the information about the routine mapping. The foreign-data wrapper receiving the above query can then use the routine mapping information to figure out the specific function that needs to be executed by the foreign server.

As with other objects in the SQL-environment, a routine mapping can be modified if the value of a generic option changes, or it can be completely dropped if the mapping is no longer needed (for example, when the function implementation ceases to exist either locally or remotely).

## *Communicating the query context*

During the planning of a complex query involving several foreign tables from the same foreign server joined by predicates, an SQL-server may generate several requests to the corresponding foreign-data wrapper. These requests will likely reference the same value expressions and predicates from the query. Often, the analysis of a value expression (or parts of it) is independent of the other expressions and can be reused between requests. In order to allow for such reuse, the wrapper interface is enhanced to provide a new routine called `AdvanceInitRequest()`. This routine contains all the parameters of `InitRequest()` routine provided in the previous version of SQL/MED and an additional parameter called

[2] The term *operators* includes both SQL built-in operators and user-defined functions.

QueryContextHandle, representing the query context. All calls to AdvanceInitRequest() with the same handle as the argument for QueryContextHandle parameter represent fragments of the same query.

## Communicating query execution costs

Evaluating a query over data in the SQL-server and a set of foreign servers requires complex query planning. While SQL-servers differ in their query planning ability, there are some common basic features. Most of the modern SQL-servers plan queries by examining several different query execution plans, estimating the time needed to execute each and picking the one with the shortest execution time. In order to expand this paradigm on queries over data in foreign tables, the SQL-server needs to acquire estimates for the query fragments executed by the foreign servers. Both the size of the result set of the query fragment execution and the query execution time are the minimum required information to perform the query planning. By way of illustration, consider following query in Example 7.

**Example 7 — A Query with a Join**

```
SELECT CITY.latitude,
       CITY.longitude
FROM EMP, CITY
WHERE EMP.city_id = CITY.id;
```

There are several ways for the SQL-server to execute this query. If the foreign server is capable of performing joins, then the whole query can be sent to the foreign server. If for each employee there is only one city, this is a feasible execution strategy. If, on the other hand, each employee is listed in up to ten cities, the result size will be up to ten times the size of the employee table. Much less data will be shipped if the SQL-server first receives the data from both foreign tables, and then performs the join operation internally. Other join methods are possible using parameterized queries, order information, and so forth.

Since the SQL-server has no knowledge of the execution model used in a particular foreign server, the foreign-data wrapper needs to provide the estimates for the result set size and execution time in order for the SQL-server to produce an efficient execution plan. SQL/MED favors an estimation model based on execution time and result size estimates, as opposed to the traditional database model where the "query cost" is divided into CPU, I/O, and network operations. In the latter case, the estimates might be hard to get when the foreign server is a third-party system with design unknown to the foreign-data wrapper writer.

The wrapper interface in the current version of SQL/MED is enhanced with four new routines that return an estimate of the cardinality of the query result, an estimate of the cost to execute the entire result of the query, an estimate of the cost to execute and return just the first row of the result, and the estimated cost of re-executing the query.

## Wrapper interface in action

In this section, we present an example query involving joins and predicates to illustrate the enhanced wrapper interface. Assume an application issues to an SQL-server the query seen in Example 8.

**Example 8 — A More Complex Query**

```
SELECT CITY.id, CITY.latitude,
       CITY.longitude
FROM EMP, CITY
WHERE EMP.name = 'Jane Doe' and
      EMP.city_id = CITY.id;
```

Assume further that both the EMP and CITY tables are managed by the same foreign server and the corresponding foreign-data wrapper is able to deal with multiple table references in the FROM clause and can perform predicates of the form "column_name = constant" and "column_name = column_name".

As described in [1], the query planning starts with the SQL-server establishing a connection to the foreign-data wrapper. Once the connection is established, the SQL-server invokes the foreign-data wrapper's AdvanceInitRequest() routine, passing a handle (a Request Handle) to a structure that describes the query as an argument. In general, an SQL-server might send different fragments of the original query for analysis to the wrapper in the process of the query planning and optimization. To keep the discussion simple, assume that the SQL-server asks the foreign-data wrapper to plan the entire query.

Once the request is received, the foreign-data wrapper analyzes first the FROM clause of the request, then the WHERE clause and finally the SELECT list. Analysis of each clause starts with determining the number of items in the clause. For the FROM clause, the foreign-data wrapper invokes the GetNumTableRefElems() routine with the RequestHandle as an argument to find out the number of table references present in the FROM clause. In the above example, the SQL-server returns 2 as the result, as there are two table references in the FROM clause. In SQL/MED, Table Reference Handles are used to represent the table references. The foreign-data wrapper invokes the GetTableRefElem() routine passing

the Request Handle and an integer value, *k*, as arguments, which returns the *k-th* Table Reference Handle. This handle can then be used as an argument in the invocation of routines such as GetTableRefTableName() to return the table name, and GetTRDHandle() to obtain a descriptor that contains the information about the columns and their data types.

As described in [1], generic options can be associated with foreign-servers, foreign-data wrappers, foreign tables, and columns of foreign tables. These options are essentially attribute/value pairs that describe information that is specific to a given object and are stored at the SQL-server as part of the metadata about those objects. For example, each of the tables in our example may be associated with a URL of a web site, specified as a generic option. The foreign-data wrapper can retrieve such generic options by invoking GetTableOptByName() routine by passing a Table Reference Handle as an argument. Furthermore, generic options associated with the columns of each of the tables can be retrieved using the GetTableColOptByName() routine by passing a Table Reference Handle and column name as arguments. Since our foreign-data wrapper is assumed to be able to handle multiple table references in the FROM clause, the FROM clause is accepted.

Once the FROM clause is analyzed, the foreign-data wrapper examines the predicates in the WHERE clause by invoking GetNumBoolVE() routine passing Request Handle as the argument to find out the number of predicates; and subsequently invoking GetBoolVE() routine multiple times to retrieve the Value Expression Handle for each such predicate. Expressions in SQL/MED are uniformly represented using trees. Each node in the tree is a value expression and has an associated Value Expression Handle. There are four kinds of value expressions: operators, column references, parameters and constants. The foreign-data wrapper can obtain the kind of a value expression by calling the GetValueExpKind() routine with a Value Expression Handle as the argument. For each operator node, the number of operands and their Value Expression Handles can be obtained invoking the GetNumChildren() and the GetVEChild() routines. Each value expression is typed and the foreign-data wrapper can check the type of a node invoking the GetValueExpDesc() routine that returns a data type descriptor. In the example above, since both Boolean expressions are of the form our foreign-data wrapper can deal with, it accepts the WHERE clause.

The foreign-data wrapper then starts analyzing value expressions in the SELECT list, which begins with finding the number of expressions in the SELECT list invoking the GetNumSelectElems() routine with a Request Handle as the argument. The foreign-data wrapper then obtains the Value Expression Handle for the *k*-th select list element by invoking the GetSelectElem() routine with the Request Handle and *k* as arguments. Routines that apply to value expressions in the WHERE clause are applicable here as well. In the above example, all select list elements are column references and our foreign-data wrapper accepts all of them.

Once the query is analyzed, the foreign-data wrapper returns two handles a Reply Handle and an Execution Handle, as described in [1]. The SQL-server examines the response from the foreign-data wrapper by using an interface similar to the one used by the foreign-data wrapper to determine the part of the query that the foreign-data wrapper is willing to execute. In the above example, the SQL-server determines that our foreign-data wrapper is capable of executing the entire query. The SQL-server can then ask the foreign-data wrapper for cost information, perform query optimization, and then settle on an execution plan. The SQL-server than invokes the Open() routine with the Execution Handle as the argument to ask the foreign-data wrapper to initiate the execution. Once the query is executed, the foreign-data wrapper sends the resulting data to the SQL-server. The communication of the results to the SQL-server happens via descriptors as described in [1].

## Enhancements to Datalinks

As already stated in [1], DATALINK is an SQL data type that allows storing in an SQL column a reference to a file that is located in a file system external to the database system.

SQL/MED allows a variety of options to be specified for columns and attributes of the DATALINK type. With these options, it can be determined how strictly the SQL-server controls the file. The possibilities range from no control at all (the file does not even have to exist) to full control, where removal of the datalink value from the database leads to a deletion of the physical file.

In the previous version of SQL/MED, the datalinks functionality supports two modes of "write permissions" for datalinks, namely FS (implying "File System") and BLOCKED.

When WRITE PERMISSION FS is specified, the system (that is, the combination of SQL-implementation, datalinker, and file manager) allows users to update a file while the file remains linked to the database. However, this mode does not provide file data

recovery, which means that, if the disk crashes or a user needs to restore the database, there is no backup data to recover. In cases of transaction failure, this may cause inconsistency between the file data and the database data. Moreover, the write access permission is determined by the file system permissions currently assigned to the file. WRITE PERMISSION FS does not support a token-based access model like the one provided with READ PERMISSION DB (implying "DataBase").

On the other hand, WRITE PERMISSION BLOCKED provides data recovery functionality for an SQL-mediated file (through the means of the datalinker), as long as the RECOVERY option is specified as YES. However, a user cannot update the file while the file is currently linked. Updating the content of a file that is linked with the WRITE PERMISSION BLOCKED requires three distinct steps:

1. Unlinking the file
2. Modifying the file
3. Re-linking the file

It should be noted that while the file is not linked to the database in step 2, it is not protected against unwanted modifications or even deletion. However, many usage scenarios require the ability to update the content of a file while it is datalinked. The current version of SQL/MED now includes this functionality, called "update-in-place". This feature provides the ability to use datalinked files for such functions as library check-out and check-in, as well as a way to back out any uncommitted file changes and restore to the previous committed version. One of the key requirements is to provide an access control scheme for accessing datalinked files, as well as a capability for updating files in a consistent way. Using datalinks and this new feature, when a disaster or crash occurs, the user can rely on the SQL-implementation to restore all the data — both SQL-data and file data — to a consistent state.

Update-in-place provides token-based access to the file similar to the mechanism used for READ PERMISSION DB. When READ PERMISSION DB is specified, the database server controls which users are authorized to access the file. File data recovery can be supported in an implementation-dependent way, as long as the RECOVERY option for the datalink is specified as YES.

The update-in-place functionality is made available by a new WRITE PERMISSION option called the ADMIN option, followed by either the keywords REQUIRING TOKEN FOR UPDATE or the keywords NOT REQUIRING TOKEN FOR UPDATE. "ADMIN" represents the fact that the SQL-server and the datalinker together decide whether a given user is authorized to update a file. "REQUIRING TOKEN FOR UPDATE" indicates that the token, which was included in the file reference when the user requested it from the SQL-server, is needed to update the column containing the datalink value in question. Conversely, "NOT REQUIRING TOKEN FOR UPDATE" indicates that this token is not needed for the update of the column. Since the update process involves more than one system (database server and file server), the same token can be used as an authentication method to indicate who can complete the whole file update process. On the other hand, to use this feature, an application has to remember the token throughout the process. In the case where applications may already have their own authentication mechanisms, one might not want to maintain the token. Using the NOT REQUIRING TOKEN FOR UPDATE option may be a good alternative for such applications.

In order to write to a file linked by a datalink value, two new string value functions are available that return a character string representation of the datalink value that includes also a write token. These functions, called DLCOMPLETEWRITE and DLPATHWRITE, are used by an application to retrieve the URL of the file that is to be updated. The former function returns the full URL, while the latter one returns only the file name and path, without the address of the file server.

In addition to the existing datalink constructor function DLVALUE, two new datalink value constructors are being added to SQL/MED: DLNEWCOPY and DLPREVIOUSCOPY. The purpose of the DLNEWCOPY constructor is to create a datalink value by which the SQL-server can tell that the content of the file referenced by that datalink is different (*i.e.*, the content has changed, but not the URL) from the value that was previously referenced by the datalink. By contrast, the purpose of the DLPREVIOUSCOPY constructor is to construct a datalink value by which the SQL-server knows that the content of the file might have changed, and that the user is not interested in maintaining the changed file but would like to revert to the file that was originally referenced by the datalink. Each of the two new datalink constructors has two input parameters. The first parameter is a character string that contains the location of the file. The second parameter is an indicator of whether the write token is included in the first argument; '1' indicates that the token is included, while '0' means that a token is not included. The datalinks constructed with the two new constructors are only valid in an SQL UPDATE statement.

## Examples

Several examples are shown below that demonstrate the use of the update-in-place feature. All examples assume

that the table shown in Example 9 has been defined and populated.

**Example 9 — Table for demonstrating update-in-place feature**

```
CREATE TABLE EMPLOYEE (
   ID          INTEGER NOT NULL,
   NAME        VARCHAR(20),
   DEPT_NO     SMALLINT,
   TITLE       VARCHAR(50),
   PHOTO       DATALINK
      FILE LINK CONTROL
      INTEGRITY ALL
      READ PERMISSION DB
      WRITE PERMISSION ADMIN
         REQUIRING TOKEN FOR UPDATE
      RECOVERY YES
      ON UNLINK RESTORE,
   RESUME      DATALINK
      FILE LINK CONTROL
      INTEGRITY ALL
      READ PERMISSION DB
      WRITE PERMISSION ADMIN
         NOT REQUIRING TOKEN
         FOR UPDATE
      RECOVERY YES
      ON UNLINK RESTORE,
   PRIMARY KEY (ID)
)
```

### How to update-in-place

An HR administrator wishes to update the picture of the employee with ID = 50100. She connects to the SQL-server and SELECTs the PHOTO column to retrieve the URL, including the write token.

```
SELECT DLURLCOMPLETEWRITE(PHOTO)
FROM EMPLOYEE
WHERE ID = 50100;
```

Assume the returned value is "HTTP://HR_SRV.XYZ.COM/hr/emp_pict/xxxx;emp50100.gif", where xxxx is the actual write token (recall that SQL/MED does not specify the format of write tokens). This file reference is then used to open the file and copy the new picture over the existing one. Assuming the new picture is in /hr/tmp/emp50100.gif, the following Unix® shell command accomplishes this task:

```
cp /hr/tmp/emp50100.gif
   /hr/emp_pict/xxxx;emp50100.gif
```

When the file copy is completed, an UPDATE statement is issued to notify the SQL-server that the new version of the file is ready to be linked.

```
UPDATE EMPLOYEE
   SET PHOTO = DLNEWCOPY (
'HTTP://HR_SRV.XYZ.COM/hr/emp_pict/xx
xx;emp50100.gif', 1 )
   WHERE ID = 50100;
```

### How to "rollback" unwanted file changes

The HR administrator has updated the new picture as in the example above. However, she wants to roll back the changes, but the file system (or "file manager" in SQL/MED terms) has no "undo" option. She can issue the following UPDATE statement to replace the modified file by the original file that was linked to the database earlier.

```
UPDATE EMPLOYEE
   SET PHOTO = DLPREVIOUSCOPY (
'HTTP://HR_SRV.XYZ.COM/hr/emp_pict/xx
xx;emp50100.gif', 1 )
   WHERE ID = 50100;
```

The reader might wonder how the old copy of the file can be restored when the file manager has no means to do it. This functionality requires that the datalinker and the SQL-server work closely together to provide the means to restore the previous copy. The functionality that is needed to accomplish this operation is not much different from that required to do "point in time" recovery which is already included in the first version of SQL/MED.

### How the SQL-server interacts with the Datalinker

This example illustrates how the SQL-server interacts with the datalinker to provide the update-in-place functionality. Since the interaction between the SQL-server and the datalinker is not standardized in SQL/MED, but is left implementation-dependent, this example shows only one possible way in which it could be done. Different systems might choose to implement it differently.

The application connects to the SQL-server and retrieves a URL, with a write token, for a file:

```
SELECT DLCOMPLETEWRITE( PHOTO )
   INTO :url
FROM EMPLOYEE
WHERE ID = 66101
```

The URL is now stored in the host variable named url.

The SQL-server checks with the datalinker to determine whether the current user has authority to update the specified file.

If the current user has authority to update the file, the SQL-server returns a file reference with an embedded write token. Here is an example of the returned value:
"HTTP://XYZ.COM/a/b/xxxx;file1.txt"
where xxxx is the write token.

The application uses this file reference to open the file. For example, it might issue the following file system call in a C program.

```
fptr =
    fopen("/a/b/xxxx;file1.txt");
```

The datalinker *intercepts* this file system call and checks to see whether the write token is valid. If the write token is valid, the datalinker allows the "fopen" operation to proceed and return a pointer to the file descriptor. The application then uses that file descriptor to read/write data to the file.

When the file update is completed, the application UPDATEs the same row with the original file reference (including the write token) to notify the SQL-server that the new version of the file is ready and any implementation-dependent archiving process can be started upon COMMIT.

Example 10 illustrates the use of the SQL UPDATE statement.

**Example 10 — Updating datalinked value**

```
UPDATE EMPLOYEE
   SET PHOTO = DLNEWCOPY(:url, 1)
WHERE ID = 66101;
```

The SQL-server forwards the request to the datalinker, which checks the validity of the write token and possibly triggers the file archive process to backup the modified file. Upon successful completion, the SQL-server returns a successful completion indication to the application.

## SQL/MED's Future

Even though the current version of SQL/MED specification adds significant new capabilities compared to the previous version, the wrapper interface is still limited to read-only access to data managed by foreign servers. Enhancing the wrapper interface with the ability to insert, update, and delete the contents of foreign tables is likely to be the focus for the next version of the standard. Handling additional query capabilities such as grouping and evaluation of aggregate functions by foreign servers may also be candidates for future work.

## Summary

In this month's column, we have reviewed the coming new version of the SQL/MED standard, outlining new capabilities that give more powerful access to foreign servers and the foreign tables that they manage, as well as new datalink facilities.

Like most standards, SQL/MED continues to evolve. When the next edition is nearing completion, we plan to provide yet another update — but don't expect this for at least another couple of years.

## References


[1] J. Melton, J.-E. Michels, V. Josifovski, K. Kulkarni, P. Schwarz, K. Zeidenstein, SQL and Management of External Data, *SIGMOD Record*, 30(1): 70–77, March 2001
http://www.acm.org/sigmod/record/issues/0103/JM-Sta.pdf

[2] ISO/IEC 9075-9:2001, *Information technology — Database language — SQL — Part 9: Management of External Data (SQL/MED)*, International Organization for Standardization, June 2001

[3] FCD (Final Committee Draft) 9075-9:200*x*, *Information technology — Database language — SQL — Part 9: Management of External Data (SQL/MED)*, currently under ballot
http://sqlstandards.org/SC32/WG3/Progression_Documents/FCD/4FCD1-14-XML-2002-03.pdf


## Web References


[1] International Committee for Information Technology Standards (INCITS)
http://www.incits.org/

[2] INCITS Technical Committee H2 (Database)
http://www.incits.org/tc_home/h2.htm

[3] International Organization for Standardization (ISO)
http://www.iso.org/

[4] ISO/IEC JTC 1/SC 32
http://www.jtc1sc32.org/