POLYHEDRA® ACTIVE QUERIES
How Polyhedra's active query mechanism improves the performance of real-time applications

Contents

Abstract ... 1
Background – the need for a new approach ... 2
Why use a DBMS in a real-time application ... 2
The need for live information ... 2
Three traditional approaches to database monitoring ... 3
An alternative approach: Active Queries ... 5
Efficient server-side handling of Active Queries ... 5
Detecting changes cheaply ... 5
Coping when things get busy ... 5
Optimistic Concurrency through Active Queries ... 6
Monitoring the monitors ... 6
Conclusion ... 7
Appendix 1: a worked example, using ODBC ... 8
Waiting for changes ... 10
Being selective ... 10
'Polling' an active query ... 12

Abstract

If many clients each have to continually poll a database to determine the up-to-date state of the information it needs, the database can become overloaded or the clients will have to reduce their poll frequency - and thus resign themselves to higher latency on the information. To avoid these problems, Polyhedra allows clients to launch active queries, where the client is automatically updated should the results of their queries be changed. This improves the timeliness of the data seen by the client and improves the scalability of the system.

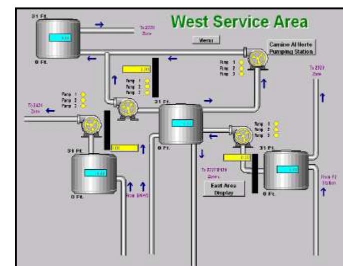
Background – the need for a new approach

Why use a DBMS in a real-time application

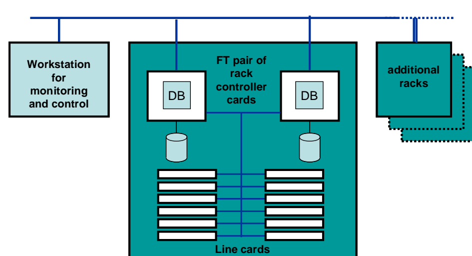
All but the simplest real-time applications are split into a number of software components to make things manageable and to allow different people or teams to be working on the application at the same time. Each component will be responsible for a separate job and communicating with other components via defined interfaces. The data-handling needs of each component will differ, but in general some data will have to be shared between components, and some data will need to be carefully preserved to guard against partial or complete system failure; there may or may not be an overlap between these categories. (Thus, some shared data might be transitory by nature, whereas configuration information will need to be shared and preserved.)

Traditionally, one component might act as the 'data manager' for the application, mediating access to the data and ensuring the critical stuff is preserved; where speed was important, this data manager component would be developed specifically for the application – but this can be complex, and it can be simpler to use a database management system designed for real-time work. In fact, the main reason for adopting commercial database technology is the same as that for adopting any COTS (common off-the-shelf) technology: cost. This shows up in many aspects, including reduced risk, reduced development time, reduced maintenance costs, and fewer specialist skills needed in the development team. As with all COTS technology, the emphasis must be that the chosen product is appropriate to one's needs, otherwise costs will actually increase rather than decrease. In order to keep the footprint down and performance up, a real-time COTS DBMS will be more light-weight than a full, general-purpose DBMS, but will still offer benefits such as transactions, data isolation, data persistence, and industry-standards APIs (such as SQL, ODBC and JDBC) as well as hot-standby capabilities to guard against partial system failure, and the ability to adapt the database structure on the fly to cope with changing needs.

The need for live information



Consider first a SCADA[1] system, controlling (say) a factory or electrical distribution network. Sensors are feeding back readings to the central system, and operators can see what is happening via 'mimic diagrams'. The question arises, how are the mimic diagrams kept up to date? Does the responsibility lie with the components that are obtaining the readings from the sensors, or the data management component?



Next, consider a base station in a telecoms system. The line cards will need to obtain configuration from somewhere; rather than each card having storage capabilities, it is easier to keep the information in cards in the control plane – preferably in a fault-tolerant fashion. External workstations would be able to change the information, but will have no network connection to the line cards and thus would have no direct way of informing them about updates. In a similar fashion, line cards would like to make sure administrative workstations could determine their status.

[1] SCADA stands for *supervisory control and data acquisition*. It generally refers to industrial control systems: computer systems that monitor and control industrial, infrastructure, or facility-based processes. See the Wikipedia article for more details: http://en.wikipedia.org/wiki/SCADA

A final example: in a financial system, the database could hold information about the most recent share prices of stocks, shares or other 'financial instruments', allowing traders to monitor their favourite shares in real time.

In both the above scenarios, there is a general need for a low-cost mechanism that allows components to be able to determine when the data in which they are interested is changed 'under their feet' by other components of the system.

Three traditional approaches to database monitoring

Polling

Notionally the simplest way for an application to detect changes is for it to periodically query the data store, and then compare the results with the previously-retrieved values. This requires no special mechanisms in the data store, or in application(s) which are feeding changes into the data store. However, it can impose a severe load on the data store module, as if an application needs to know about a change within a short interval – say, $1/10^{th}$ of a second if merely updating a screen, or $1/1000^{th}$ or less if the application needs to take mitigating action – then it has to launch queries with that frequency. It also adds to the workload of the client application, as it has to analyse the response to see what – if anything – has changed. In addition, if the data store and the application needing the data are on separate machines, then the network traffic can be significant. In summary, if low latency is required the overheads of use polling are high – and there can be catastrophic degradation in performance at times of peak activity or when the system is expanded sufficiently to cause an overload condition.

External notification mechanisms

Polling can be thought of as a 'passive pull' technology. An alternative is a 'push' notification scheme, where the component that first detects the change is responsible for notifying any other component that needs to be told. The component publishing the information can either know who needs to be informed at design time, or there can be a more dynamic (and probably more generic) 'publish and subscribe' system, with API calls for registering interest in an event and for asking for information to be sent to the parties who had registered to receive information.

The problem with tailored push mechanisms – where the originator/detector of an event tells concerned parties about changes that are relevant to the recipient (without telling them about irrelevant changes) – is that it can be significant effort to adapt to changing requirements during the development phase and beyond. A generic publish & subscribe mechanism avoids this problem, but may not offer sufficiently fine granularity or may not allow precisely the right information to be supplied. Thus, the listener might just get a notification that a value associated with a datum point has changed, and may have to find out for itself whether it is a point of interest, and if so what is the current value. Even if the publish and subscribe mechanism avoids these problems, it still complicates the job of the component publishing the information as it is responsible both for putting the information into the database (for long term storage, say, or to be available to components that are not running or that do not need immediate notification) and for initiating the notification process to subscribers.

Database triggers

Some database systems allow code to be attached to a database, which is triggered on an event. Normally, the trigger code will be written by the database administrator using PL/SQL or in a DBMS-specific trigger language such as Polyhedra's CL language, and the role of the triggers is either to perform some integrity checks (and aborting the transaction if the conditions are not met), or to do some 'knock-on' actions. The integrity checks can be as simple as ensuring particular attributes of the changed records are within a range directly or indirectly defined by other fields in the record, or can be much more complex. Thus, in a telecoms application, an attempt

to create a record for a new connection might be rejected if either of the endpoints are flagged as being in a call – if, however, the new connection is permissible, the trigger on the connection table can set the flags indicating that the two endpoints are in a call. (A separate trigger would clear these flags when the connection record is deleted.)

The uses described above require the trigger code to be run within the transaction that activated it (either as the triggering event is performed, or with other triggered code at the end of the transaction), and obey the ACID properties (see panel to right). In particular, the triggered code should have no side-effects that cannot be rolled back, should the transactions fail. For this reason, one should treat with great caution data management systems that allow code written in C or C++ to be called within transactions, as atomicity cannot be enforced by the system. However, it is OK to run such code post-transactionally as the server can cancel the triggers prior to execution if the transaction is rolled back.

Where a data management system allows post-transactional triggers, such triggers can be used for knock-on actions that take too long to be done in the original transaction – but they can also be used for event reporting: either the database administrator can add C or C++ code to hook in to a generic publish and subscribe mechanism (as described in the previous subsection), or the data management system can have its own inbuilt notification mechanism.

"The ACID Test" for data management

ACID is a commonly-used abbreviation for **A**tomic, **C**onsistent, **I**solated and **D**urable, relating to desirable properties of data management systems that group operations into transactions:

- **Atomic**—Transactions are all-or-nothing—the system never does just part of what is asked.
- **Consistent**—Transactions take the system from one consistent state to another—the system rejects any transactions that break the rules.
- **Isolated**—Transactions operate independently, as though fully serialized, with intermediate, mid-transaction states hidden.
- **Durable**—If a database indicates that a transaction has successfully completed, the application can rely on the changes being preserved - even through a variety of system failure scenarios

There is an implicit property that is also required: that the ONLY way of reading or writing to the data is via transactions.

All of these transactional properties are vital for a reliable data storage system. Most database systems will offer ACID-compliance, though this is much less common in simpler data stores.

Reporting events in this way has the advantage that the client applications generating the updates only have to tell the database; their job is simplified, and, as a central component is taking responsibility for the job of notification, overall system correctness is improved. However, as with the use of a generic notification system described in the previous subsection, it is difficult to ensure the granularity of the notifications is sufficiently fine; in addition, those told about the events will probably need to query the database and work out for themselves what has changed. There is also a maintenance problem, as those who are designing client applications will have to ensure (probably by liaison with the database design team, if they are the only ones that can add triggers) that suitable events are generated for their needs – and these events are updated or deleted if and when their needs change.

In some systems, there will also be a problem with security. Most database systems provide some sort of user-based controls that can be used to determine (down to the level of individual columns of a table) who can view and update the contents of the database. In embedded systems, such controls are often used in a role-based manner, restricting what can be done by individual components, and thus limit the damage should a component go wrong, but they may also be used to stop people discovering operational information that they are not entitled to know about. If user-written code can register to be notified about particular types of events, then people may be able to make inferences about data that they could not directly query the database about.

An alternative approach: Active Queries

In Polyhedra, SQL queries from a client can be static or ‘active’. Active Queries provide automatic client notification of changes in the database in real-time. Active queries are launched in a similar manner to normal (‘static’) queries, except that the database is told the queries are ongoing. The client receives the initial set of results, but instead of forgetting about the query, the server remembers it (until it is cancelled by the client, or the client connection is closed or lost). Whenever the database changes in a way that affects the query, the server notices, and sends a ‘delta’ message to the client.

A delta tells the client enough information to bring it up to date. Thus, it will report added rows, deleted rows and changes to individual rows: for changed rows, it does not send the whole row contents, just the cell values that changed. Consequently, efficient use is made of the network bandwidth, when client and server are on separate machines. In addition, the job of the client application is greatly simplified:

- It is informed when updates occur, with low latency: no need to poll
- If it was just told a change had occurred, it would have to reissue the query to find the new values – and then, depending on the application, it might have to analyse the results to find out what had changed. By contrast, Polyhedra’s active query mechanisms avoid the need to reissue the query, and the client libraries let the application zoom in on the changes.

Once launched, active queries continue until closed by the client, or the client connection is closed, or the query becomes invalid – for example, if the table is dropped, though adding a column to table only invalidates a query if the client had used ‘select *’ rather than specifying the columns of interest.

Efficient server-side handling of Active Queries

Detecting changes cheaply

Conceptually, at the end of every query the server checks to see what active queries are affected by the changes that have been done, so that it can work out which ones need to be sent deltas. Of course, to do it like that would be grossly inefficient, so instead the active query code running within the server makes use of a fine-grain event triggering mechanism provided by the core database engine. This allows the active query mechanism to ignore irrelevant changes:

- the presence of an active query does not affect the performance of other queries, nor the performance of transactions that do not alter the table(s) monitored by the active query;
- active queries on single objects (identified by their primary key) have no affect on the performance of transactions that do not modify the monitored objects; and,
- the affect on the performance of a transaction that does alter data being monitored by an active query is very low, as the work of updating the client is done post-transactionally in a separate thread.

Coping when things get busy

Suppose values are changing rapidly, and the clients cannot keep up with the changes. One example is a process control application, where sensor values are being read (through Analog-to-Digital Converters, or ‘ADCs’) as frequently as possible and the results fed into the database: a characteristic of high-sensitivity ADCs is that two consecutive readings are rarely the same, even if the input value seems stable. You may then have graphic devices or control room ‘mimic diagrams’ being fed in turn from the database, and the network may be busy or the client machine may not be able to handle the workload.

To cope with such situations, if the client is running slowly and cannot keep up with the stream of deltas, the server will start to ‘merge deltas’. Thus, clients will not be told every change, but will be kept as up to date as possible without slowing down the system. This means that the system is responsive even in peak load conditions, and copes well with crashed client applications.

In fact, clients can take advantage of the delta merging mechanism to further reduce the load on the system. At the time the query is launched, the client can specify a ‘minimum delta interval’, in thousandths of a second. Whenever a server sends a delta over an active query with an interval defined, it will ensure the next delta will not be sent until the interval has expired, with all intervening changes merged. A graphic client can choose, say, an interval of a fifth of a second, which is long enough to allow people to read the values without worrying about flicker – yet short enough to appear real-time to those watching the displays. By choosing an appropriate minimal delta interval, and ensuring only the data required is retrieved (e.g., avoiding ‘select *’ and only asking for the columns and rows of relevance to the client), efficient, scalable and responsive systems can be produced.

Optimistic Concurrency through Active Queries

As well as keeping clients up to date with changes in a server, Polyhedra’s active query mechanism also provides an effective way for the client to change the database: it simply has to change the result set of the active query. When it does so, the client library will automatically ask the server to make the corresponding changes. Of course, for this to work, the query has to be ‘updateable’ in that it must be unambiguous as to which record has to be inserted, modified or deleted to make the database match the modified result set: in practice, this means the client cannot modify the database though active queries involving joins or SQL functions. Updates done through the active query mechanism avoid the use of the SQL engine, which improves efficiency.

Whenever a change request is received by the server, the first thing it does is check whether it satisfies the optimistic concurrency constraints: basically, the transaction is only accepted for processing if no other client has made a conflicting change since the last delta was sent. Clearly, the new transaction cannot change a record if something else has just deleted it, but the transaction is also rejected if something else has changed a record attribute that the client wants to alter. Optimistic concurrency is efficient when transactions are expected to succeed, which is usually the case in embedded applications.

Updates through active queries can be grouped into larger transactions. When the client starts such a transaction, the server is automatically told (by the client library code) to block sending deltas on the active queries involved in the transaction. When the client later commits or cancels the transaction, it will – if necessary – receive a merged delta to bring it up to date, but in the case of a successful transaction it does not have to be told about the changes it made!

Monitoring the monitors

Recent editions of Polyhedra have been enhanced to allow the database designer to detect when active queries are set up on particular tables. They can do this by adding records to a configuration table to say which tables are of interest; once set up, records are created in a special table when active queries are established on any of the designated tables, and removed when the queries are cancelled. Each of these special records allows one to find out details of an individual query, and of the connection used to launch the query. Trigger code (written in Polyhedra’s trigger language, CL) can be attached to the special table to respond to the creation and deletion of records, and can take appropriate action. For example, it would be straightforward to kill any connection that tried to launch active queries on a particular table unless the connection was on the same machine:

ENEA

```
script DataQuery
  on  create
      -- (nb: 2130706433 corresponds to net address 127.0.0.1)
      if sourcetable="currency" and \
         machine of session = 2130706433 then delete session
  end create
end script
```

While this particular example may be of limited use(!), the mechanism allows more sophisticated and meaningful actions. For example, suppose that some of the data values are being fed from external devices according to some polling schedule, and the polling schedule for a particular point is specified in its database record. The query monitoring mechanism is sufficiently powerful that it is possible to detect when an active query is looking at a particular point, and trigger code can automatically increase the poll frequency for that point.

Conclusion

As discussed in this document, Active Queries have many benefits over traditional solutions:

- No polling, so low latency achieved with less load on the server
  - less network traffic
  - Scalability of system improved
  - No server load when clients aren't listening
- better handling of overload conditions, more gradual degradation
- No need for separate notification mechanisms
- Clients doing updates are simpler: they don't need to tell others about changes
- Clients doing queries are simpler: they are told what has changed
- No need to set up and manage database trigger code to propagate notification messages

In addition, the bidirectional nature of Active Queries allows an efficient way for clients to update the database on an ongoing basis without involving SQL, and the new mechanism for monitoring the placement of active queries allows, for example, polling frequencies for external data sources to be tuned dynamically.

In summary, Active Queries bring an additional and powerful capability to database developers allowing for faster program development and greater system responsiveness. By avoiding polling they make the system more stable and scalable, and the bidirectional nature provides an easy way for a client to update the database. The uses for Active Queries are varied and many!

Appendix 1: a worked example, using ODBC

We will use the Polyhedra ODBC API to illustrate the use of active queries. This API is available on each of the platforms on which Polyhedra runs. Active queries can also be invoked through each of the other APIs offered by Polyhedra, including JDBC and - for Windows programmers - the OLE DB interface. The example will concentrate on database monitoring through a single query, but could be readily extended to handle multiple queries.

Rather than jump straight in with the full active query example, we shall take a ‘normal’ program, and then extend it to make use of active queries: this will allow us to see how little needs to be changed to adopt the technology, and then how easy it is to make more complete use of the capabilities on offer. Our starting point will be a simple application which connects to a database, queries it, displays the result and stops. Let us first look at the code to start ODBC and connect to a database accessible via port 8001 on a remote machine:

```
SQLHENV  henv;
SQLHDBC  hdbc;
SQLHSTMT hstmt;
/* Allocate an environment handle */
ret = SQLAllocHandle (SQL_HANDLE_ENV, SQL_NULL_HANDLE, &henv);
check_success (ret, SQL_HANDLE_ENV, henv,
               "Failed to allocate environment handle");
/* Allocate a connection handle */
ret = SQLAllocHandle (SQL_HANDLE_DBC, henv, &hdbc);
check_success (ret, SQL_HANDLE_ENV, henv,
               "Failed to allocate connection handle");
/* Connect to the database */
ret = SQLConnect (hdbc, (SQLCHAR *)"192.168.1.101:8001",
                  SQL_NTS, 0, SQL_NTS, 0, SQL_NTS);
check_success (ret, SQL_HANDLE_DBC, hdbc, "Failed to connect to the database");
```

(All ODBC functions start with the letters `SQL`, and almost all of them take a ‘handle’ as one of the parameters, which can be thought of as a pointer to a data structure that is owned and maintained by the ODBC library which uses it to store context information. For clarity, all ODBC function names in the code snippets are emboldened, Polyhedra-specific extensions will be shown in red, and comments in green.)

As always when using ODBC, we first establish an ‘environment’, and are given an environment handle (which we will store in the local variable `henv`); using this handle, we next ask for a connection handle (which we store in `hdbc`) and then open the connection. The function `check_success()` is here assumed to be a user-written function that will check if the ODBC library function return code indicated an error, and if so will obtain the error message by use of the `SQLGetDiagRec()` function, print it, and stop the program.

The key lines establishing a query and printing the results could then read something like:

```
/* Allocate a statement handle */
ret = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &hstmt);
check_success (ret, SQL_HANDLE_DBC, hdbc, "Failed to allocate a statement handle");
/* Execute the query */
ret = SQLExecDirect ( hstmt
                    , (SQLCHAR *)"select code, us dollar from currency"
                    , SQL_NTS);
check_success (ret, SQL_HANDLE_STMT, hstmt, "Failed to execute statement");
fetch_all_data (hstmt); /* Fetch and display the result set */
```

(The function `fetch_all_data()` is assumed to be a user-written function that uses standard ODBC functions such as `SQLFetchScroll()` and `SQLGetData()` to iterate over the result set and print the contents of each row).

ENEA

To complete the program, all that is needed is to make the appropriate ODBC calls to close the connection and release the handles, and then stop:

```
SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
SQLDisconnect (hdbc);
SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
SQLFreeHandle (SQL_HANDLE_ENV, henv);
return 0;
```

To convert this program to one polling the database and reporting the results every 5 seconds, one simply needs to replace the earlier code snippet that set up the query and reported the results by the following code:

```
/* Allocate a statement handle */
ret = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &hstmt);
check_success (ret, SQL_HANDLE_DBC, hdbc, "Failed to allocate a statement handle");
for (;;)
{  /* Execute the query */
   ret = SQLExecDirect ( hstmt
                       , (SQLCHAR *)"select code,usdollar from currency"
                       , SQL_NTS);
   check_success ( ret, SQL_HANDLE_STMT, hstmt
                 , "Failed to execute statement");
   fetch_all_data (hstmt); /* Fetch and display result set */
   Sleep (5);              /* wait five seconds */
}
```

To convert this program to one using an active query, one simply sets a flag on the statement handle prior to launching the query, and then move the call to `SQLExecDirect()` outside the loop:

```
/* Allocate a statement handle */
ret = SQLAllocHandle (SQL_HANDLE_STMT, hdbc, &hstmt);
check_success ( ret, SQL_HANDLE_DBC, hdbc
              , "Failed to allocate a statement handle");
/* Use dynamic cursor, to signal that the query is active */
ret = SQLSetStmtAttr ( hstmt, SQL_ATTR_CURSOR_TYPE
                     ,(SQLPOINTER)SQL_CURSOR_DYNAMIC, 0);
check_success (ret, SQL_HANDLE_DBC, hdbc, "Failed to set dynamic cursor");
/* Execute the query */
ret = SQLExecDirect ( hstmt, (SQLCHAR *)"select code,usdollar from currency"
                    , SQL_NTS);
check_success (ret, SQL_HANDLE_STMT, hstmt, "Failed to execute statement");
for (;;)
{  fetch_all_data (hstmt); /* Fetch and display result set */
   Sleep (5);              /* wait five seconds */
}
```

Of course, this is wasteful client-side: it wakes up every 5 seconds even if nothing has changed, and it prints out the whole result set; if one wanted to report just what had altered, the client application would have to compare the result set with a copy that it had taken. It is also unresponsive to changes: the current state is only reported every 5 seconds. However, it puts a very low load on the server:

- There is no round trip to the server every 5 seconds; instead, the Polyhedra client library merely checks whether there are any incoming messages, and processes them appropriately before allowing the user to iterate over the result set; and,
- If the data is changing rapidly, the server will notice that the client is asleep and will enter delta-merging mode.

ENEA

Waiting for changes

So far, we have used only standard ODBC calls. For the next step, making the client wait for events, we will need to make use of some Polyhedra extensions to ODBC. First, we shall have to enable the Polyhedra event mechanism, and then enable events on the query:

```
/* Enable async event handling */
ret = SQLSetEnvAttr ( henv, SQL_ATTR_POLY_ASYNC_EVENTS_ENABLE
                    , (SQLPOINTER)SQL_POLY_ASYNC_EVENTS_ENABLE_ON, 0);
/* Enable async statement events on the query */
ret = SQLSetStmtAttr ( hstmt, SQL_ATTR_POLY_ASYNC_EVENTS_ENABLE
                     , (SQLPOINTER)SQL_POLY_ASYNC_EVENTS_ENABLE_ON, 0);
/* optional: set a minimum interval between deltas, in milliseconds */
ret = SQLSetStmtAttr ( hstmt, SQL_ATTR_POLY_MINIMUM_DELTA_INTERVAL
                     , (SQLPOINTER)10, 0);
```

(For brevity, we have omitted from the above - and from all subsequent examples – code to call `check_success()` *after each ODBC function call to see whether it worked correctly.)*

The names of the form `SQL_ATTR_POLY_…` are defined in a supplementary header file supplied as part of the Polyhedra release kits. The main loop can now become:

```
/* Fetch and display initial result set */
fetch_all_data(hstmt);
/* enter the main loop */
for (;;)
{   ret = SQLHandleMsg (henv); /* wait for a delta */
    fetch_all_data (hstmt);    /* Fetch and display result set */
}
```

(Of course, it is not necessary to have reorganized the code to have a call to `fetch_all_data()` in front of the loop as well as inside it – but introducing this change here will make the next section simpler to follow!)

The overall effect of these changes is that the main loop is woken up every time there is an incoming delta – though if changes are occurring really fast, delta merging will occur to ensure that deltas are at least 10 milliseconds apart (as specified by the `SQL_ATTR_POLY_MINIMUM_DELTA_INTERVAL` statement parameter).

Being selective

The final refinement is to get the ODBC library to tell us what has changed, so that the output can be more selective. To do this, we first add lines (somewhere after the call of `SQLExecDirect()`) to set up variables with global scope that can be used by the ODBC library to report flags and a bookmark…

```
SQLULEN      Bookmark;                 /* Buffer for bookmark      */
SQLUSMALLINT RowStatusArray[1];        /* Array for row status     */
SQLUSMALLINT ColumnStatusArray[2];     /* Array for column status */
```

… and then we instruct the ODBC library to use them, by inserting the following lines after the call to `SQLExecDirect()`:

```
ret = SQLSetStmtAttr (hstmt, SQL_ATTR_ROW_STATUS_PTR, RowStatusArray, 0);
ret = SQLSetStmtAttr (hstmt, SQL_ATTR_POLY_COLUMN_STATUS_PTR,
                      ColumnStatusArray, 0);
ret = SQLSetStmtAttr (hstmt, SQL_ATTR_FETCH_BOOKMARK_PTR, &Bookmark, 0);
```

We are now ready to adapt the main loop to make use of the bookmark mechanism, with an outer loop and an inner loop (and also an innermost loop to handle the details of each delta). The outer loop will call the `SQLHandleMsg()` function to wait for something interesting to happen:

The inner loop will contain the main code, based around a call of the Polyhedra `SQLGetAsyncEvent()` function to see what events are waiting, and only breaking out of this inner loop when the function reports all outstanding events have been processed. In our case, the main events we expect are an indication of a loss of connection, or an incoming delta. (The other possibility is that the query has failed because it is no longer valid, due to a dynamic change in the schema or the security privileges.) In the case of a delta, the Polyhedra `SQLGetAsyncStmtEvent()` function can be called repeatedly to get information about affected rows.

```
/* outer loop, handling events */
for (;;)
{   SQLUSMALLINT fnId;
    SQLSMALLINT  hType;
    SQLHANDLE    h;
    /* inner loop, getting the next event */
    while (SQLGetAsyncEvent (henv, &fnId, &hType, &h) == SQL_SUCCESS)
        switch (fnId)
        {
          case SQL_API_SQLDISCONNECT:
            /* Database connection lost. Free the handles, etc and stop */
            SQLFreeHandle (SQL_HANDLE_STMT, hstmt);
            SQLDisconnect (hdbc);
            SQLFreeHandle (SQL_HANDLE_DBC, hdbc);
            SQLFreeHandle (SQL_HANDLE_ENV, henv);
            return 0;

          case SQL_API_SQLFETCH:
            /* Data has changed - fetch and display modified result set */
            while (SQLGetAsyncStmtEvent (hstmt) == SQL_SUCCESS)
                fetch_by_bookmark (hstmt);
            break;
        } /* (end of switch, and of inner loop: 'while') */
    ret = SQLHandleMsg (henv); /* wait for the next batch of work */
} /* (end of outer loop: 'for') */
```

Each call to `SQLGetAsyncStmtEvent()` would have left the `Bookmark` variable pointing at the affected row in the result set (ready for the client application to place the cursor on the row via a call of `SQLFetchScroll (hstmt, SQL_FETCH_BOOKMARK, 0)`); the value in `RowStatusArray` will flag what type of change was made to the row, and (in the case of updates) the contents of `ColumnStatusArray[]` will indicate which attributes have a changed value. In the case of a row deletion, the earlier-reported values for the record are still accessible to the client application, to save the user code from having to cache its own copy. (The old values for a deleted row are kept until the application has moved the cursor off that row, or until the next delta is received, whichever comes first.)

Thus a possible definition of `fetch_by_bookmark()`, stripped of all error checking, could be:

ENEA

```
void fetch_by_bookmark (SQLHSTMT hstmt)
{
    SQLRETURN ret;
    SQLCHAR   code[MAX_STRING_LENGTH];
    SQLDOUBLE usd;
    SQLLEN    len;

    ret = SQLFetchScroll (hstmt, SQL_FETCH_BOOKMARK, 0);
    ret = SQLGetData (hstmt, 1, SQL_C_CHAR, &code, MAX_STRING_LENGTH, &len);
    switch (RowStatusArray[0])
    {
    case SQL_ROW_ADDED:
      ret = SQLGetData (hstmt, 2, SQL_C_DOUBLE, &usd, 0, &len);
      printf ("Row Added - Code: %s, 1 Dollar buys: %lf.\n", code, usd);
      break;

    case SQL_ROW_UPDATED:
      if (ColumnStatusArray[2] == SQL_COLUMN_UPDATED)  /* has USD value changed? */
      {
          ret = SQLGetData (hstmt, 2, SQL_C_DOUBLE, &usd, 0, &len);
          printf ("Code: %s, 1 US Dollar now buys %lf.\n", code, usd);
      }
      break;

    case SQL_ROW_DELETED:
      printf ("Currency %s removed from resultset\n", code);
      break;
    }
}
```

The Polyhedra release kits and evaluation kits contain the source code for a complete application – including code for `check_success()`, `fetch_all_data()` and a more complete version of `fetch_by_bookmark()` – that was used as the initial source for the code snippets in this white paper.

The current state of our worked example monitors a single query – so when a delta is received, there is no question about which active query has been updated. It is straightforward to adapt this example to monitor a number of queries: we would allocate each of them their own statement handle and set off the queries prior to entering the main loop, and then within the `case SQL_API_SQLFETCH` clause in the main loop we would use the contents of the variable `h` to see which active query had just received a delta.

'Polling' an active query

In the above example, the call of `SQLHandleMsg()` suspends the thread until the Polyhedra library detects something interesting has happened. In many applications, it is more appropriate to have a main loop doing other things, and occasionally 'peek' at the Polyhedra status to see if anything has happened. You can achieve this by setting a 'timeout' on the environment handle:

```
ret = SQLSetEnvAttr (henv, SQL_ATTR_POLY_EVENT_TIMEOUT, (SQLPOINTER)3, 0);
```

This instructs the Polyhedra library to wait in `SQLHandleMsg()` only for a short time for an incoming event – in this case for about 3 milliseconds, though in practice the delay may be slightly longer, depending on clock resolution and granularity. Note that as before this does NOT involve a round trip to the server; typically all that will happen is a quick check to see if a message had been received or a heartbeat timeout had expired.

ENEA

E&OE: this technical note is believed to be an accurate description of the features and functionality of Polyhedra® as at the time of writing – but as the product family undergoes continual improvement the behaviour in areas covered by this document is subject to change without notice (though our compatibility principles mean that existing code will rarely need alteration and existing applications will interact with new versions of the software).

Enea®, Enea OSE®, Netbricks®, Polyhedra® and Zealcore® are registered trademarks of Enea AB and its subsidiaries. Enea OSE®ck, Enea OSE® Epsilon, Enea® Element, Enea® Optima, Enea® Optima Log Analyzer, Enea® Black Box Recorder, Enea® LINX, Enea® Accelerator, Polyhedra® Flashlite, Enea" dSPEED Platform, Enea® System Manager, Accelerating Network Convergence™, Device Software Optimized™ and Embedded for Leaders™ are unregistered trademarks of Enea AB or its subsidiaries. Any other company, product or service names mentioned above are the registered or unregistered trademarks of their respective owner. © Enea AB 2011