mcobject®

precision | data management

# SQL or Navigational Database APIs: Which Best Fits Embedded Systems?

**Abstract**: For embedded systems developers, the choice of database application programming interfaces (APIs) often boils down to the high-level SQL language and Call Level Interface, and navigational APIs integrated with C++ and other languages. Which API is best? This paper examines the familiarity and ease-of-use often cited as benefits of SQL. A sample application is implemented with SQL and then with a navigational API, to explore the issues of programming ease, maintainability, determinism and learning curve. Special attention is given to the significance of SQL optimizers in evaluating database APIs.

Supplied by

PhaedruS SystemS

**E-mail: info@phaedsys.com**
**www.phaedsys.com**

## Introduction

As embedded systems grow smarter, developers are turning to commercial database management systems to support new, data-intensive features. In doing so, developers confront a variety of database application programming interfaces (API)—and must choose the right one for their projects.

One choice is SQL, a high-level language developed for business systems, which has been extended into the embedded systems environment. Since its introduction in the 1970s, SQL has gained popularity for its (to some extent) vendor independence. SQL also offers a higher level of abstraction to programmers by separating database access language from the physical database implementation.

Developers can also choose navigational APIs that are more closely integrated with the third-generation programming languages, such as C and C++, used in such projects. Many database vendors offer navigational APIs either alongside SQL or as the sole interface for their database products. Clearly, these alternative APIs serve a purpose, or address some needs, or they would not survive. Intuitively, developers may sense that these APIs offer greater efficiency and precision, due to their origin in powerful, widely used programming languages.

Which API to use? This paper examines two reasons application developers might consider SQL for an embedded application. The first is familiarity, since SQL is associated with some of the best-known Enterprise databases such as Oracle and DB2. The second is the hope that development will be easier with higher-level SQL. In the sections below, a sample application is implemented with SQL and then with a navigational API, in order to explore the issues of programming ease, maintainability, determinism, and learning curve.

## SQL and Navigational APIs Defined

SQL is a "set" oriented language. In other words, it works with a set of result rows. For example, a simple query, SELECT * FROM TABLE-A, will generate a result set, or the set of rows matching the query criteria.

In contrast, navigational APIs work on one record at a time. A function in the API is used to locate a record in the database, then another record, and another, through a looping procedure. Application logic determines whether the current row is a member of the set of interest. "Navigational" is a general term. In practice, navigational APIs use a number of navigation methods (sequential, indexed, or in the case of hierarchical DBMSs, pointer-based).

## Comparison Application

To demonstrate the different programming techniques, consider a simple program that audits Internet traffic, such as might be found in an intelligent network infrastructure device (a caching device or firewall, for example) or a corporate Web monitoring application.  Requirements are as follows:

Maintain a chronological record of URL visits
Report all URLs visited
Report URLs by User
Report # accesses for specified URLs

Any given URL can be visited by one or more users, and any user can visit one or more URLs.  This creates a many-to-many relationship.  For simplicity, we won't worry about decomposing the URLs to avoid storing the home address of www.mcobject.com/index.htm and www.mcobject.com/partners.com multiple times.  The database design employed for this paper is not intended to reflect an optimal design, but to aid in comparing SQL to a navigational API.

The SQL used in this comparison is ANSI SQL and the C API is ODBC. McObject's eXtremeDB, a database designed to be used in intelligent, connected devices and embedded systems, provides the navigational API.

For the test database, the SQL database definition language (DDL) is:

```
create table url
(
   path       char(31) primary key
);
create table visitor
(
   vname      char(31) primary key
);
create table visit
(
   path           char(31) references url,
   vname          char(31) references visitor,
   when_visited   timestamp
);

create index vvisitor on visit( vname );
create index vindex on visit( path, when_visited );
```

The corresponding eXtremeDB DDL is:

```
declare database urlmon[20000];

class URL
{
  char<32>               path;

   unique tree <path>    by_path;
};

class visitor
{
  char<32>               vname;

   unique tree <vname>   by_vname;
};

class visit
{
  char<32>         path;
  char<32>         vname;
  unsigned<4>      when_visited;

  tree <path, when_visited> to_path;
  tree <vname> to_visitor;
};
```

In both schemas, the *visit* record supports the many-to-many relationship between URL and VISITOR records. eXtremeDB could have used OID (object identifier) and ref (references) to implement the relationship, but to keep the examples as similar as possible, we have employed the relational approach of primary and foreign keys.

The *visit* class and table have redundant copies of *path*, which is the foreign key of *url.path*. It is indexed with *when_visited* to support queries like "show me who accessed this URL in the last hour". Because *path* is the first component of the index, any DBMS should be able to use it to optimize the join "url.path = visit.path".

The following SQL/ODBC code fragment demonstrates an implementation of the first requirement, to report all URLs in the database.

```
int ReportURLs()
{
   HSTMT  StmtHdl;
   char  *selecturl     = "select path from url";
   char  path[32];
   SDWORD path_ind;
   int stat;

   if ((stat = SQLAllocStmt(ch, &StmtHdl)) != SQL_SUCCESS)
      return stat;

   stat = SQLPrepare(StmtHdl, (UCHAR*) selecturl, SQL_NTS);
   if (stat != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      SQLFreeStmt(StmtHdl, SQL_DROP);
      return stat;
   }
   stat = SQLBindCol(StmtHdl, 1, SQL_C_CHAR, path,
                     sizeof(path), &path_ind);
   if (stat != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }
   if ((stat = SQLExecute(StmtHdl)) != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }

   puts("\n");
   for ( ; ; ) {
      if ((stat = SQLFetch(StmtHdl)) != SQL_SUCCESS)
         break;
      printf("%s\n", path);
   }

   if (stat != SQL_NOTFOUND)
      OnError(eh, ch, StmtHdl);

   if((stat=SQLFreeStmt(StmtHdl, SQL_DROP)) != SQL_SUCCESS)
   {
      OnError(eh, ch, StmtHdl);
      return stat;
   }
   return SQL_SUCCESS;
}
```

The code allocates an ODBC statement handle and causes the SQL database to parse the SQL SELECT statement, prepare an execution plan, and actually execute the statement. Within a loop, each of the result set rows are fetched and the URL path is printed.

The equivalent eXtremeDB implementation is shown next.

```
int ReportURLs()
{
   MCO_RET           rc = 0;
   mco_cursor_t      UrlCsr;
   mco_trans_h       trn;
   URL               UrlHandle;
   char              path[32];

   mco_trans_start( db, MCO_READ_ONLY,
                    MCO_TRANS_FOREGROUND, &trn );

   /* initialize cursor */
   rc = URL_by_path_index_cursor( trn, &UrlCsr );
   if ( rc != MCO_S_OK ) {
      mco_trans_commit( trn );
      return rc;
   }
   puts("\n");

   for(rc = mco_cursor_first(trn, &UrlCsr);
      rc == MCO_S_OK;
      rc = mco_cursor_next(trn, &UrlCsr))
   {
      rc = URL_from_cursor( trn, &UrlCsr, &UrlHandle );
      rc |= URL_path_get( &UrlHandle, path, sizeof(path));
      printf("%s\n", path);
   }
   mco_trans_commit( trn );
   return MCO_S_OK;
}
```

The eXtremeDB code begins a transaction (all database access, read or write, occurs within the scope of a transaction in eXtremeDB) and instantiates a cursor that will be used to iterate over the URL objects in the database. This is done by setting up a loop with *mco_cursor_first* to initialize the loop and *mco_cursor_next* in the loop increment. Each iteration of the loop obtains a handle to the current URL object, and retrieves and prints the *path*. When *mco_cursor_next* tries to read beyond the last URL object, it returns MCO_S_CURSOR_END, causing the loop to terminate. The transaction is then closed.

The amount of programming required to achieve these results with either API is roughly comparable, though the eXtremeDB implementation requires slightly less coding (there is no need to bind host variables) and will execute faster because there are no parse or execute stages. However, the SQL steps of allocating a statement handle and preparing and executing the query, and the eXtremeDB steps of starting a transaction and instantiating a cursor, are roughly comparable.

The key difference between the implementations is one of transparency, or the connection between the original requirement, and the implementation code. In the SQL example, except for the SQL *select* statement, it is impossible to see programmatically what the application is doing, other than processing some *select* statement. The eXtremeDB implementation, on the other hand, is quite clear. A cursor for the *by_path* index of the URL class is instantiated and used to iterate over the URL objects in the database. For each URL, a class handle is initialized from the cursor and used to retrieve the URL's path. There is no disconnect between the application code and the operations being carried out.

In a more complex example, the various SQL-ODBC API function calls will be far removed from the associated text of the SQL statement, making it more difficult for a programmer who is not intimately familiar with the code to relate the program code to the functional requirements. This increases the risk of introducing defects and increases the cost of maintaining the application during its life cycle.

The next example demonstrates a more complex requirement that requires joining, or navigating, all three class/table types to list all users and, for each user, every URL visited.

The SQL implementation is shown first:

```
int ReportURLbyUser()
{
   HSTMT  StmtHdl;
   char   path[32],
          vname[32];
   SDWORD path_ind,
          vname_ind;
   int    stat;
   char   *select     = "\
select path, vname \
from url, visitor, visit \
where url.path = visit.path \
and visitor.vname = visit.vname \
order by vname";

   if ((stat = SQLAllocStmt(ch, &StmtHdl)) != SQL_SUCCESS)
      return stat;

    stat = SQLPrepare(StmtHdl, (UCHAR*) select, SQL_NTS);
    if (stat != SQL_SUCCESS) {
        OnError(eh, ch, StmtHdl);
        SQLFreeStmt(StmtHdl, SQL_DROP);
        return stat;
    }

    stat = SQLBindCol(StmtHdl, 1, SQL_C_CHAR, path,
                      sizeof(path), &path_ind);
    if (stat != SQL_SUCCESS) {
        OnError(eh, ch, StmtHdl);
        return stat;
    }

    stat = SQLBindCol(StmtHdl, 2, SQL_C_CHAR, vname,
                      sizeof(vname), &vname_ind);
    if (stat != SQL_SUCCESS) {
        OnError(eh, ch, StmtHdl);
        return stat;
    }

    if ((stat = SQLExecute(StmtHdl)) != SQL_SUCCESS) {
        OnError(eh, ch, StmtHdl);
        return stat;
    }

   puts("\n");
   for ( ; ; ) {
      if ((stat = SQLFetch(StmtHdl)) != SQL_SUCCESS)
```

```
            break;
         printf("%s\t%s\n", vname, path);
   }

   if (stat != SQL_NOTFOUND) {
      OnError(eh, ch, StmtHdl);
   }

   if ((stat = SQLFreeStmt(StmtHdl, SQL_DROP)) !=
              SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }
   return SQL_SUCCESS;
}
```

Despite the requirement to join three tables, the code is not substantially different from the previous code fragment. The addition of a second selected column requires one more call to SQLBindCol.

The equivalent eXtremeDB implementation is shown next:

```
int ReportURLbyUser()
{
   MCO_RET            rc = 0;
   mco_cursor_t       UrlCsr, VisitorCsr, VisitCsr;
   mco_trans_h        trn;
   URL                UrlHandle;
   visit              VisitHandle;
   visitor            VisitorHandle;
   char               path[32],
                      vname[32];
   int                eq;

   mco_trans_start( db, MCO_READ_ONLY,
                   MCO_TRANS_FOREGROUND, &trn );

   /* initialize cursor */
   rc = visitor_by_vname_index_cursor(trn, &VisitorCsr);
   rc |= visit_to_visitor_index_cursor( trn, &VisitCsr );
   rc |= URL_by_path_index_cursor( trn, &UrlCsr );
   if ( rc != MCO_S_OK )
   {
      mco_trans_commit( trn );
      return rc;
   }
   puts("\n");
   for(rc = mco_cursor_first(trn, &VisitorCsr);
      rc == MCO_S_OK;
      rc = mco_cursor_next(trn, &VisitorCsr))
   {
      rc = visitor_from_cursor( trn, &VisitorCsr,
                                &VisitorHandle );
      rc |= visitor_vname_get( &VisitorHandle, vname,
                               sizeof(vname));

      for(rc = visit_to_visitor_search( trn, &VisitCsr,
                           MCO_EQ, vname, sizeof(vname));
         rc == MCO_S_OK;
         rc = mco_cursor_next(trn, &VisitCsr) )
      {
         // use _compare method to ensure we haven't
         // advanced to the next visit_vname
         rc = visit_to_visitor_compare( trn, &VisitCsr,
                               vname, sizeof(vname), &eq ))
         if( rc || eq )
            break;
         rc = visit_from_cursor( trn, &VisitCsr,
                                 &VisitHandle );
         rc |= visit_path_get(   &VisitHandle, path,
                                 sizeof(path) );
         rc |= URL_by_path_find( trn, path, sizeof(path),
                                 &UrlHandle );
         rc |= URL_path_get(     &UrlHandle, path,
                                 sizeof(path) );

         printf("%s\t%s\n", vname, path);
      }
   }
   mco_trans_commit( trn );
   return MCO_S_OK;
}
```

Again, the amount of coding is comparable between the two implementations. The steps using eXtremeDB's navigational API are a little different, requiring a loop within a loop to achieve the same result as the SQL join. In this case, the outer loop iterates over the *visitor* objects in alphabetical order of *vname*. For each of the *visitor* objects, the second loop iterates over the *visit* objects by using the *visitor.vname* as the search value for the *visit_to_visitor_search* method.

For each iteration of the inner loop, *visit_to_visitor_compare* is called to determine if the current *visit* object's *vname* field is equal to the search value. This is to test whether *mco_cursor_next* has stepped beyond the set of relevant objects, and is equivalent to determining the SQL ‘set’ for the *visitor->visit* join. If the comparison passes, a *visit* handle is initialized from the cursor, the *visit.path* field is retrieved, and it is used as the search value to find the associated URL object. This is the equivalent of the *visit<-path* join.

The application code in which the navigational API implements the equivalent of a SQL three table join is neither more complex nor more voluminous. In fact, the use of an API whose naming scheme is driven by database design creates self-documenting code, making it easy to follow the processing logic. In contrast, the SQL-ODBC API functions (SQLPrepare, SQLBindCol, SQLExecute, SQLFetch) have no direct association to the contextual database and do not contribute to the readability and maintainability of the code.

The final example demonstrates how to implement aggregation with SQL and with the eXtremeDB navigational API. For each URL stored in the database, the application reports the number of times the URL has been visited by all users. The SQL implementation is shown first:

```
int ReportURLOverTime
{
   HSTMT  StmtHdl;
   char  path[32];
   SDWORD count,
          path_ind,
          count_ind;
   int stat;
   char  *select     = "\
select path, count(*) \
from url, visit \
where url.path = visit.path \
group by path";

   if ((stat = SQLAllocStmt(ch, &StmtHdl)) != SQL_SUCCESS)
      return stat;

   stat = SQLPrepare(StmtHdl, (UCHAR*) select, SQL_NTS);
   if (stat != SQL_SUCCESS) {
       OnError(eh, ch, StmtHdl);
       SQLFreeStmt(StmtHdl, SQL_DROP);
       return stat;
   }

   stat = SQLBindCol(StmtHdl, 1, SQL_C_CHAR, path,
                     sizeof(path), &path_ind);
   if (stat != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }
   stat = SQLBindCol(StmtHdl, 2, SQL_C_LONG, &count,
                     sizeof(count), &count_ind);
   if (stat != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }

   if ((stat = SQLExecute(StmtHdl)) != SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }

   puts("\n");
   for ( ; ; ) {
      if ((stat = SQLFetch(StmtHdl)) != SQL_SUCCESS)
         break;
      printf("%s\t%d\n", path, count);
   }

   if (stat != SQL_NOTFOUND) {
      OnError(eh, ch, StmtHdl);
   }

   if ((stat = SQLFreeStmt(StmtHdl, SQL_DROP)) !=
                  SQL_SUCCESS) {
      OnError(eh, ch, StmtHdl);
      return stat;
   }
   return SQL_SUCCESS;
}
```

The SQL code joins the URL and VISIT tables, counting the number of VISIT rows joined with each URL row. Other than the SQL SELECT statement, the implementation is not so different than the previous two SQL implementations.

The eXtremeDB navigational implementation follows:

```
int ReportURLOverTime()
{
   MCO_RET        rc = 0;
   mco_cursor_t   UrlCsr, VisitCsr;
   mco_trans_h    trn;
   URL            UrlHandle;
   visit          VisitHandle;
   char           path[32], vpath[32];
   int            count;

   mco_trans_start( db, MCO_READ_ONLY,
                     MCO_TRANS_FOREGROUND, &trn );

   /* initialize cursor */
   rc = URL_by_path_index_cursor( trn, &UrlCsr );
   rc |= visit_to_path_index_cursor( trn, &VisitCsr );
   if ( rc != MCO_S_OK )
   {
      mco_trans_commit( trn );
      return rc;
   }

   puts("\n");

   for(rc = mco_cursor_first(trn, &UrlCsr);
      rc == MCO_S_OK;
      rc = mco_cursor_next(trn, &UrlCsr))
   {
      rc = URL_from_cursor( trn, &UrlCsr, &UrlHandle );
      rc |= URL_path_get( &UrlHandle, path, sizeof(path));
      printf("%s\n", path);

      for(count = 0,
         rc = visit_to_path_search( trn, &VisitCsr,
                          MCO_EQ, path, sizeof(path), 0);
         rc == MCO_S_OK;
         rc = mco_cursor_next(trn,&VisitCsr),count++)
      {
         // ensure we haven't advanced to the next
         // visit_vname
         rc = visit_from_cursor( trn, &VisitCsr,
                                 &VisitHandle );
         rc |= visit_path_get( &VisitHandle, vpath,
                               sizeof(vpath) );
         if( rc || strcmp( path, vpath ) )
            break;
      }
      printf("%s\t%d\n", path, count);
    }
    mco_trans_commit( trn );
    return MCO_S_OK;
}
```

This eXtremeDB implementation resembles the earlier navigational example, consisting of a loop within a loop to affect the 'join' of *path->visit* in order to count the number of *visit* objects for each *path*. This is how eXtremeDB navigates one-to-many relationships. In this case, because *visit.to_path* is a compound index, we cannot use the _compare method because no comparison value for *when_visited* is provided. So, we simply retrieve the *visit.vpath* field and use strcmp to test for the end of the *visit* objects for a path.

## Programming ease

Which API results in simpler programming? The eXtremeDB navigational implementations above require about as much labor, measured in lines of code, as the SQL equivalent. Given an understanding of either API, neither is more complex. However, a SELECT with a large number columns will require one SQLBindCol function per column, and a parameterized statement will require a call to SQLBindParameter for each argument to the statement. A SELECT to fetch ten columns with three parameterized filters will require thirteen such function calls which, while not complex, do add to the volume of application code. Crafting correct SQL statements for complex operations, such as a correlated sub-query, requires a depth of understanding of SQL that most non-specialists (such as embedded systems developers) lack, and therefore adds to the learning curve and detracts from the maintainability of the application.

## Maintainability

Because the database objects being acted upon are used in the API, the eXtremeDB navigational API implementation yields application code that is self-documenting. Given an understanding of the underlying data model, a developer who is unfamiliar with the implementation will be able to read it, understand it, and maintain it.

With SQL, the body of application code is often quite distant from the text of the SQL statement(s). This lack of integration adds difficulty by requiring the maintenance programmer to conceptualize two distinct logical systems, while predicting the interaction between the two. This "mental juggling" is essential to insure that a change to one system does not impair the other.

For example, tinkering with the SQL statement to address new application requirements can impair performance if it causes the optimizer to choose an inferior execution plan. This, in turn, necessitates a database re-design, such as adding or dropping indexes. These iterations of application code changes and database changes add to the maintenance cost. With a navigational API, however, the programmer *by definition* writes the execution path and knows whether the database design supports the requirement, and will modify the schema at the same time, if necessary.

## Determinism

In the examples above, the navigational approach is deterministic in the sense that when the application is compiled, it is known exactly how the data will be traversed. In contrast, the SQL optimizer has a number of choices to consider and the data navigation is not determined until run-time. For the three-table join shown above, the SQL optimizer will choose from six possible join combinations. A four-table join presents 24 possible join combinations. Many factors, such as the presence of indexes, the distribution of values within indexes, and the number of rows in the tables, determine the choice of execution plans. Depending on the optimizer, the execution plan can change from execution to execution as the metrics driving this determination change.

This means performance may slip as distribution of data in the database changes. SQL databases have various techniques to cope with this. An ‘update statistics’ operation is a common one, but can be time consuming and in some database systems it must be explicitly invoked, which puts a maintenance burden on the end-user.

## Learning Curve

As suggested above, developers often fall into a trap from not understanding their database vendor’s SQL optimizer. Unfortunately, the consequences are often not felt until the end of a project when, to everyone’s dismay, performance is abominable when tested with real-life data.

Failing to understand the optimizer and the reason for its execution plan can mean re-writing significant portions of the application’s SQL. For example, to get around optimizer limitations, the developer may have to break down complex queries into simpler ones.

This “hidden” learning curve—consisting of the time required to fully comprehend the behavior of any SQL optimizer—is not trivial. Important questions to ask about the optimizer include:

**Does the database use a rules-based or cost-based optimizer?** Rules-based optimizers determine execution plans based upon pre-determined rules and without consideration for the actual contents of the database. Cost-based optimizers are more complex and consider the cardinality of indexes, the number of rows in a table and other factors to attempt to calculate the I/O cost of potential execution plans.

One problem with cost-based optimizers is that the number of possible join combinations increases by N-factorial for N tables joined in the query, and each index on each table creates a possible navigation path that the optimizer must evaluate. An optimizer can easily spend more time analyzing all the plans than it would take to execute one of the early plans considered (even if it was not, ultimately, the most efficient). Cost-based optimizers should include a way to halt evaluation of execution plans and simply go with the best one found so far.

**Can the optimizer perform an index intersection?** If not, the developer can wind up, at the end of a project, rewriting what was thought to be one or more simple queries. For example, a query with a filter on two columns ("columnA = 3290 and columnB like 'u%' order by columnB, columnC") would need to be separated into two queries, one for each of the conditions. The application would then find the intersection of the two result sets, and sort the results via a quick sort or other algorithm in order to gain acceptable performance. The database should do this by itself, but many cannot perform an index intersection, requiring this workaround.

**Does the database support clustered indexes?** Clustered indexes represent the best approach to sorting. Non-clustered indexes should be used for filtering. The reason for this is that a clustered index physically orders data rows according to the index. Hence, retrieving a row set in the order of the clustered index requires only scanning the data table (possibly from some starting point determined by a filter), minimizing disk I/O and leading to superior performance.

In contrast, consider the overhead entailed in sorting with a non-clustered index: with a one megabyte table and 100K of cache, there is enough cache for 10% of the table. The index returns the indexed columns, sorted in the desired order, but they point to random pages in the data file where the rest of the table columns are found. This results in a less than one-in-ten chance of a data page referenced by the index being in cache (less than 1/10, since the index pages also take up cache space). This causes a tremendous amount of I/O to return the row set in sorted order by using the non-clustered index.

I/O is tremendously expensive in terms of performance and should clearly be avoided. Therefore the optimizer should do a table scan (or use a filter if one is available) and sort the results in memory. Even if the row set is large and the sort algorithm needs to swap to disk occasionally, this results in less I/O than thrashing the index and data pages through the cache.

(Note that the discussion above doesn't apply to main-memory databases, which eliminate I/O.)

**Does the database support covered queries?** The scenario above can be avoided if the columns selected are also the ones to be sorted, a secondary index exists for the columns, and the optimizer has the ability to perform a covered query.  A covered query is one that does not need to access the table pages because all necessary data exists in the index pages.

## Conclusion

Clearly, when efficiency is important, knowing standard SQL is a small part of the learning curve in working with a specific SQL database.  The developer must also learn the capabilities of the DBMS optimizer and know it will support the desired level of performance. Similarly, future developers performing maintenance or enhancement will need to understand this optimizer, as well as master the application/database interaction that is somewhat masked by the SQL-ODBC API. These requirements often outweigh SQL's presumed ease-of-use, tipping the embedded systems database choice toward the more transparent, self-documenting and deterministic navigational API.

In addition, performance considerations can be magnified by embedded system CPUs that, for economic reasons, are often a fraction of the clock speed of contemporary workstation and server CPUs.  Query optimization is a heavily CPU-intensive task. Choosing the navigational API eliminates this source of overhead, supporting real-time performance in network infrastructure, telecommunication switches and other real-time devices.