## Introduction

<table>
<thead>
<tr>
<th>General Coding Practices</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Types....................................................................</td>
<td>4</td>
</tr>
<tr>
<td>Use String for Text Variables in Java; Char in COBOL...............</td>
<td>4</td>
</tr>
<tr>
<td>Use Integer, Decimal, or Float Types for Numbers..................</td>
<td>4</td>
</tr>
<tr>
<td>Use Type Consistently for a Business Element........................</td>
<td>5</td>
</tr>
<tr>
<td>Data Structures....................................................................</td>
<td>6</td>
</tr>
<tr>
<td>Avoid Structured Data Structures in Java Environments............</td>
<td>6</td>
</tr>
<tr>
<td>Array Processing...................................................................</td>
<td>7</td>
</tr>
<tr>
<td>Use Static Arrays In COBOL Environments..............................</td>
<td>7</td>
</tr>
<tr>
<td>Searching Arrays....................................................................</td>
<td>8</td>
</tr>
<tr>
<td>Write Code That Doesn’t Require Index Checking....................</td>
<td>9</td>
</tr>
<tr>
<td>Subroutines.........................................................................</td>
<td>10</td>
</tr>
<tr>
<td>Use In-Program Functions for Small Subroutines.....................</td>
<td>10</td>
</tr>
<tr>
<td>Use in and out Parameter Modifiers for Service Functions..........</td>
<td>10</td>
</tr>
<tr>
<td>Avoid Using in and out Parameter Modifiers for Local Functions...</td>
<td>11</td>
</tr>
<tr>
<td>Reports..............................................................................</td>
<td>12</td>
</tr>
<tr>
<td>Use Print Forms or Hand Coded Reports for Best Performance........</td>
<td>12</td>
</tr>
<tr>
<td>Other Good Practices..........................................................</td>
<td>13</td>
</tr>
<tr>
<td>Use Substring Notation Instead of String Functions................</td>
<td>13</td>
</tr>
<tr>
<td>Load Data Tables Once.......................................................</td>
<td>13</td>
</tr>
<tr>
<td>Avoid Logging in Main Line Execution Paths...........................</td>
<td>14</td>
</tr>
<tr>
<td>Efficient SQL Database Access.............................................</td>
<td>14</td>
</tr>
<tr>
<td>SQL Record Definitions..........................................................</td>
<td>15</td>
</tr>
<tr>
<td>Treat the Record as a View..................................................</td>
<td>15</td>
</tr>
<tr>
<td>Match EGL Type to Column Type for Simple Data Types...............</td>
<td>15</td>
</tr>
<tr>
<td>Use Text Variables For DATE/TIME/TIMESTAMP Columns...............</td>
<td>16</td>
</tr>
<tr>
<td>Match Null Specification.....................................................</td>
<td>17</td>
</tr>
<tr>
<td>Good Practices for SQL in EGL Programs..................................</td>
<td>17</td>
</tr>
<tr>
<td>Use Result Set Identifier...................................................</td>
<td>17</td>
</tr>
<tr>
<td>Limit the Number of Rows Returned in a Result Set..................</td>
<td>18</td>
</tr>
<tr>
<td>Implement Programmatic Paging for Interactive User Interfaces....</td>
<td>19</td>
</tr>
<tr>
<td>Good Practices for SQL in COBOL Environments........................</td>
<td>20</td>
</tr>
<tr>
<td>Specify singlerow When Retrieving a Single Row.....................</td>
<td>20</td>
</tr>
<tr>
<td>Avoid Using prepare Statement.............................................</td>
<td>20</td>
</tr>
<tr>
<td>Good Practices for SQL Statements In Any Language..................</td>
<td>20</td>
</tr>
<tr>
<td>Use SQL Indices.................................................................</td>
<td>20</td>
</tr>
<tr>
<td>Use SQL Stored Procedures..................................................</td>
<td>21</td>
</tr>
<tr>
<td>Web Coding Practices..........................................................</td>
<td>21</td>
</tr>
<tr>
<td>Disclaimers.........................................................................</td>
<td>21</td>
</tr>
</tbody>
</table>
Introduction

Enterprise Generation Language is a high level application specification that is translated into COBOL or Java for execution in a target runtime environment. EGL is also an enterprise modernization language that was designed as a migration target for fourth generation languages like IBM VisualAge® Generator and Informix 4GL. As such, the language provides alternative ways of implementing some business functions.

This white paper is intended to help customers understand the performance implications of the coding practices they use when their application runs in COBOL or Java environments. The document will periodically be updated as more information surfaces and implications of implementation choices are better understood.

This document contains recommendations for coding general business logic and relational database access in all types of applications with good performance characteristics. Later editions will include best performance practices that relate specifically to Web development and service oriented environments.

The significant conclusions from the document are listed here for readers who want the bottom line quickly. Refer to the body of the document for more detail and supporting information.

Best performance practices for Java runtime are:

- Use string type for all text
- Do not use structured records (records with a fixed storage layout)

Best performance practices for COBOL

- Use char or mbChar text type for applications used with a single national language or Unicode text type for multinational applications
- Use static, fixed length arrays in structured records
- Do not use libraries for small functions that are frequently invoked

Performance practices common to all runtime environments:

- Use libraries instead of called programs for shared subroutines
- If your application needs to access several fields in a record in a dynamic array, copy the record to a single record variable rather than referencing individual record fields with an array index.
EGL Best Practices: Coding For Performance

General Coding Practices

Variable Types

Use String for Text Variables in Java; Char in COBOL

Background

Text variables are used to hold strings of alphanumeric characters. EGL supports five text types: string, Unicode, char, mbChar, and dbChar. String and Unicode variables use UCS2 double-byte representations of Unicode text characters internally to represent the text characters. Char variables use ASCII or EBCDIC single-byte representations of text characters for a specific language or set of languages. Mbchar variables are ASCII or EBCDIC mixed byte length representations of pictorial languages like Chinese, Japanese, or Korean. Dbchar is a double byte representation of the pictorial languages; it is obsolete and supported only for migration of old applications that use it.

Code page conversion is required on every assignment or comparison between a string or Unicode variable on the one hand and a char or mbchar variable on the other.

Observations

- Measurements clearly indicate that string is the best performing text type in Java while char is the choice for speed in COBOL.
- The worst choice is to mix text types in the same application. Especially in COBOL, but even in Java, you pay an extra price for converting text when moving it between variables of different types.

Recommendations

- Choose string for text type if your runtime environment is Java or if there is a serious chance your application will need to be a “multinational” application (that is, support text entered in the language of any country)
- Choose char or mbChar if the application will spend its entire life in COBOL supporting users that speak a common language. If using types char or mbchar for text variables in your program, ensure your COBOL literals are defined with the same types by also specifying program property textLiteralDefaultIsString=no for each program in the application.

Use Integer, Decimal, or Float Types for Numbers

Background

Numeric variables are used to numbers used in calculations. EGL supports several types of numeric variables:
EGL Best Practices: Coding For Performance

<table>
<thead>
<tr>
<th>Types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>smallInt, int, bigInt</td>
<td>Integers as a binary number</td>
</tr>
<tr>
<td>bin</td>
<td>Binary number with assumed decimal point</td>
</tr>
<tr>
<td>smallfloat, float</td>
<td>Floating point numbers</td>
</tr>
<tr>
<td>num, numc</td>
<td>Zoned decimal numbers, one byte per digit, digits are numeric characters, except for last digit which uses first half of byte for a sign</td>
</tr>
<tr>
<td>decimal, money, pacf</td>
<td>Packed decimal numbers, two digits per byte, except for last byte which contains one digit plus a sign. IBM z and I series machines have instructions in the hardware to do calculations with these types; other machines do not.</td>
</tr>
</tbody>
</table>

Observations

- Measurements indicate that if you need to perform a lot of calculations quickly, floating point variables should be used.

Recommendations

- Choose floating point types for numbers used in calculations that don’t have to be precise and for variables that have to contain very large or very small values beyond the limits of the other types. Type float provides more precision (17 decimal digits) than smallfloat (7 digits) with a slight increase in performance cost..
- Use a text variable for numbers that are never used in calculations like phone numbers and id numbers.
- Use decimal or money for values that have decimal places and have to be precise, like your bank account balance.
- Use integer types for loop control, indexing, and associated adding and subtracting. You could substitute decimal here if you are sure your target environment is COBOL.

For example, this code uses smallint variables for loop control:

```plaintext
function sum ( values money[] ) returns ( money )
i smallint; // loop control variable
loopEnd smallInt = values.getsize() ;
total money = 0 ;
for ( i from 1 to loopEnd )
total = total + values[i] ;
end
return ( total ) ;
end
```

Use Type Consistently for a Business Element
EGL Best Practices: Coding For Performance

Use the same type for the variables that represent the same business element everywhere in the application. This prevents costly type conversion from being required on assignments and comparisons between these variables.

You can ensure this occurs by defining a `dataItem` part for the business element. A `dataItem` is a user defined variable type. You can specify the `dataItem` name in place of the EGL types and EGL will determine the actual type from the `dataItem` definitions.

For example, the following `dataItem` to represent the business elements “salary” and “percent expressed as decimal fraction” in the application:

```
dataItem salary money(10,2) end
dataItem percent decimal(3,2) end
```

Here is a library function that uses `dataItem` types salary and percent:

```
function computeNewSalary ( currentSalary salary, raisePercent percent ) returns ( salary )
  return ( currentSalary + ( currentSalary * raisePercent ));
end
```

**Data Structures**

**Avoid Structured Data Structures in Java Environments**

**Background**

EGL records are a specification of a set of variables (record fields) that together describe the attributes of a single entity. The EGL reference uses the term “structured record” to refer to a record definition for which a fixed storage layout is defined. Each field in a structured record must have a fixed length (no strings allowed). Each field has a level number specified which indicates whether it follows and lies within the previous field in the record. The EGL reference does not have a term for records which are without a structure. In this document we will refer to structured records as “fixed” records and records without a structure as “flexible” records (flexible because the storage layout of the record can vary based on what is efficient for the runtime implementation, Java or COBOL, rather be controlled by the developer).

**Observations**

- Measurement scenarios on assignments between records of the same type and “moves by name” between records of the same and different types show that using flexible records provides the best performance across both Java and COBOL environments.
- COBOL environments can handle structured records as efficiently as flexible records.

**Recommendations**
EGL Best Practices: Coding For Performance

- Use flexible records for best performance across all environments
- Use structured records only when necessary to match the layout of existing file records or to redefine the layout of record fields

Examples

Examples of flexible and structured record definitions are show below:

- A flexible record using string type for text and date columns:

  ```
  record flexSQLRecord type sqlRecord
  {
    tableNames = ["benchmarktable"] ,
    keyItems = [ id ]
  }
  id string ;
text string ;
count int ;
age int ;
cash money (16,2) ;
salary money (16,2) ;
start string ;
due string ;
moreText string ;
longText string ;
end
  ```

- A structured record using char type for text and date columns:

  ```
  record structuredSQLRecord type SQLRecord
  {
    tableNames = ["benchmarktable"] ,
    keyItems = [ id ]
  }
  10 id char(10) ;
  10 text char(20) ;
  10 count int ;
  10 age int ;
  10 cash money (16,2) ;
  10 salary money (16,2) ;
  10 start char(10) ;
  10 due char(10) ;
  10 moreText char(30) ;
  10 longText char(100) ;
end
  ```

Array Processing

Use Static Arrays In COBOL Environments

Background
EGL allows arrays to be declared in two ways. One is as a multiply occurring item in a fixed record structure. We’ll refer to this as a static array because its dimension never changes. If the number of active entries can vary, the developer must manage that by including a variable to hold the number of array slots that are being used. For example:

```c
record staticIntArrayRecord
  10 count int ; // count of valid entries
  10 staticIntArray int[1000] ; // static array of 1000 integers
end
```

The other kind of array is defined outside of a structured record. In this case, the array dimension is the initial number of entries. The application can append, insert, and remove entries in this array, so we’ll refer to it as a dynamic array. In the following example, `dynamicIntArray` is a single dimensional array of integers initialized with zero entries:

```c
dynamicIntArray int[0] ;
```

**Observations:**

- COBOL programs work most efficiently when array storage is allocated by the compiler as a fixed length array of fixed length elements. Measurements confirm that both processing static arrays is more efficient than processing dynamic arrays in COBOL environments.
- Java takes longer to load static arrays than dynamic arrays, but processes references to the array elements at equal speed. However static arrays are defined only in structured records which don’t allow strings.

**Recommendations**

- Use dynamic arrays for Java environments or for scenarios where a maximum array size cannot be determined ahead of time.
- Use static arrays in COBOL environments.

**Searching Arrays**

**Background**

EGL supports an “in” expression to search for values in arrays. The expression takes a value and a starting index as input and returns the index of the first element in the array whose value equals the input value.

**Observation**

Measurements indicate the “in” operator provides better performance than a “loop and compare” search, especially in COBOL environments.
EGL Best Practices: Coding For Performance

Write Code That Doesn’t Require Index Checking

Background

EGL COBOL build option `checkIndices` has a default value of yes. If specified, the COBOL generator adds checks to your generated program to check for indexes out of range or references to dynamic arrays that are null. Index checking is done for both array references and substring indexing.

Recommendations

- Code your program to be “safe” so that it never uses out of range indices or references null arrays. Then you can specify `checkIndices = no` and have a better performing program.

- Make sure you start with an empty dynamic array (array exists, but doesn’t have any entries) rather than a null array (array doesn’t exist). Declare your array variable with a dimension of 0 or attribute clause or assign a “not null” array to the variable. For example:

```cobol
function arrayExample ( )
  array1 int[] ; // null array
  array2 int[0] ; // empty array
  array3 int[] {} ; // empty array
  array4 int[] = [ 1, 2 ] ; // array with two entries
  size int ;
  size = array1.getSize() ; // exception – null array
  size = array2.getSize() ; // size = 0 – array is empty
end
```

- When using indices, you can use the array `getSize` function to make sure that your index won’t be out of range. For example,

```cobol
function sumArray ( intArray int[] )
  i, iEnd, sum int ;
  iEnd = intArray.getSize() ;
  for ( i from 1 to iEnd)
    sum = sum + intArray[i] ;
end
```

- When using substring notation, you can use the `strlib.characterLen` function to make sure your substring index won’t be out of range.

Warning

If you don’t use safe coding practices for array indexing or array references and you still try for better performance by specifying `checkIndices = no`, the program behavior will be unpredictable. You won’t get a clear error message describing the problem if something goes wrong.
Subroutines

Use In-Program Functions for Small Subroutines

Background

EGL provides four methods of implementation for subroutines:

- Called programs
- Functions defined within the same program and usable only from that program. This includes function parts automatically included in a program when referenced by the program.
- Functions defined within a library that can be invoked by users of the library
- Services

The performance characteristics of services invoked across the network are not dealt with in this document. If the service is deployed as a “local” service, its performance characteristics are like those of a library.

Libraries also contain global variables that can be accessed directly by programs that use the library. All the programs that use the library share the same instance of the global variable.

Observations

- Calling a library is faster than calling a program
- Calling a program instead of a library function is very costly in Java environments.
- Java invocation of a library function or access to a library variable is as fast or faster than invoking a function or accessing a variable in the same program
- The opposite is true in COBOL. It much faster to invoke a function or access a variable in the same program than to use a library function or variable.
- Access to global library variables in COBOL is implemented as a library function call.

Recommendations

- For COBOL environments, implement frequently invoked small subroutines in the same program as in which they is called.
- Otherwise use libraries instead of called programs for shared subroutines.

Use in and out Parameter Modifiers for Service Functions

Background

EGL allows parameters in the function prototype to be declared as in, out, or inout. “in” specifies that the argument corresponding to the parameter is input only; the argument value is not modified when the function returns even if the parameter variable is modified in the function.
“out” specifies that the argument is a return value only. The value in the argument is not passed to the function when the function is invoked. “inOut” specifies that the parameter is both an input value and a return value. The argument value is passed to the function on entry; the argument is updated with the parameter value on return.

**Recommendations**

- Specify “in” as a modifier for any service parameter that does not need to be returned to the program invoking the service. This prevents the parameter value from being passed back across the network when the service returns.
- Specify “out” for any parameter used only for returning a value to the caller. This avoids passing the argument value across the network to the service when the service does not use the value.

**For Example:**

```egl
function getCustomerName ( 
    customerID string in,
    customerName string out
) 
. 
. 
end
```

Note: This practice does not apply if the service is always generated as a local EGL service where the values do not travel across the network. In this case, follow the best practices for libraries described below.

**Avoid Using in and out Parameter Modifiers for Local Functions**

**Background**

EGL allows local and library function parameters to be declared as *in*, *out*, and *inout* as described previously for services.

**Observations**

The Java environments incur overhead from making a temporary object for arguments where the parameter was declared with an *in* or *out* modifier before the value is passed to the function.

**Recommendations**

Avoid the need for specifying *in* or *out* modifiers by ensuring that the function code does not modify or alter an input only parameter or assume any value is on entry for an output parameter. Instead, either specifically set (or leave as default) the *inout* modifier and get a better performing function invocation in Java environments:
function getCustomerName ( 
  customerID string inout, 
  customerName string inout )

c.

c.

c.

c.

c.

Reports

Use Print Forms or Hand Coded Reports for Best Performance

Background

EGL provides several ways of creating reports:

- Print forms
- Simple text reporting facility
- Handlers for populating reports created using Business Intelligence and Reporting Tools (BIRT)
- Reports coded using EGL text formatting functions

For more information on reports, refer to the RBD Information Center. Look for the EGL Language Reference in Contents under Developing, Developing EGL applications.

You can find information on BIRT Reports and EGL text reports under EGL reports in the topic “Creating Applications with User Interfaces” in the EGL Language Reference. Find Print forms under Text UI in the same topic.

Observations

- In Java, the most efficient report programs are handcoded using structured records and formatting functions
- In COBOL, the most efficient report programs use print forms with a resource association of SEQ (COBOL I/O operations) instead of the default SEQRS (EGL runtime I/O operations)

Recommendations

- Use BIRT when you want to create sophisticated output in PDF or HTML format, including graphics, tables, graphs, and charts
- Use print forms for high volume reporting in COBOL environments
- Use simple text reporting for low volume reporting in Java environments
Other Good Practices.

Use Substring Notation Instead of String Functions

Background

EGL provides substring notation to request assignments and comparisons between substrings of text variables. The notation lets you specify the position of the first and last characters in the substring. Here is an example:

```plaintext
if ( aChar10[8:10] == bChar10[3:5] )
aChar10[1:3] = bChar10[6:8];
end
```

VisualAge Generator (VAG), a predecessor of EGL, did not support subscript notation for text fields. Rather it provided system functions which wrapped the corresponding string function from the C language. Versions of these functions (vgLib.copyStr and vgLib.compareStr) still exist in EGL to support migration of VAG applications.

Observations

- In all cases except one, substring notation is simpler to code and performs better than using the copyStr and compareStr functions.
- The exception (using substring notation for comparisons in COBOL) is being evaluated as a performance defect.

Recommendation

- Always use substring notation instead of the copyStr and compareStr functions.

Load Data Tables Once

Background

EGL provides the dataTable part to support loading of predefined arrays of initialized data. A typical use would be abbreviations of state capitals.

The table contents may be updated by the program so the EGL runtime be default loads a new copy of the table for each user in a transactional environment.

The table can be defined with attributes of resident and/or shared. Resident means the table remains loaded in the runtime region even after a program that uses the table completes. Shared applies especially to the CICS environment where a single copy of the table is loaded and shared by all transactions running in the CICS region.

Recommendation
EGL Best Practices: Coding For Performance

- If the contents of an EGL data table contents are “read only”, always define the table as *shared*.
- Also define it as *resident* unless it is needed conditionally and infrequently.

For example:

```plaintext
dataTable aSharedTable type BasicTable {
  shared = yes,
  resident = yes
}
3 aColumn char(3);
{ contents = [
    "abc",
    "xyz"
  ]
}
end
```

Avoid Logging in Main Line Execution Paths

**Background**

Many customers include logging or tracing capabilities in large applications. At times we've seen situations where applications perform poorly due to the resource being consumed by application traces that developers forgot to remove before running the performance evaluation.

**Observation**

The cost of checking a “logging active” switch is very low compared to the cost of actually writing a log entry.

**Recommendation**

Minimize performance problems related to logging overhead by following one simple rule: make sure any logging done from the main line path of your production applications can be turned on or off without regenerating your program.

**Efficient SQL Database Access**

SQL performance is a topic that is difficult to cover in a book, much less a short paper. Fortunately SQL coding practices for efficient database access apply to EGL applications just as they do to any language with embedded SQL. The good advice in the many books on SQL performance applies here.

This section gives information specific to the EGL implementation of SQL, plus a few of the SQL practices that are generally applicable.
SQL Record Definitions

Treat the Record as a View

Background

An SQL view is a virtual table defined to the database manager. It represents a predefined query presented to the developer as a table that can access from a program. The view is subset of information extracted from the database that the application can process.

EGL SQL records derived from the DB schema include all the table columns in the record definition, some of which your application may need to access and some of which may be not relevant to your application.

Recommendations

- If there are columns in the DB table that are not used in your application, then clone the SQL record derived from the DB manager and remove the columns from the record definition that your application does not need. Performance will improve because unneeded columns won’t be brought in when the table is read.
- Alternatively, if you want to keep one record definition to use in different contexts where different columns are required, edit each SQL statement derived from the record definition in order to remove references to columns and associated record host variables for the columns that are not used in that context.

Match EGL Type to Column Type for Simple Data Types

Background

IBM DB2® and JDBC drivers perform data conversion when moving data between program variables and database columns if the variable and column types are compatible. Performance however is best if column type matches the type that database manager or driver is retrieving.

Recommendation

Use the following EGL types for EGL variables associated with columns in SQL tables for best performance:

<table>
<thead>
<tr>
<th>SQL Column Type</th>
<th>EGL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAR, VARCHAR</td>
<td>char or mbchar – COBOL string - Java</td>
</tr>
<tr>
<td>GRAPHIC, VARGRAPHIC</td>
<td>string or Unicode (Unicode table)</td>
</tr>
<tr>
<td>SMALLINT</td>
<td>dbchar (not a Unicode table)</td>
</tr>
<tr>
<td>INTEGER</td>
<td>smallint</td>
</tr>
</tbody>
</table>

11/10/2009 15
Use Text Variables For DATE/TIME/TIMESTAMP Columns

Background

EGL provides date, time, timestamp, and interval data types. EGL supports special date and timestamp calculations using these data types. EGL also supports functions and patterns for displaying date, time, and timestamp information in a wide variety of formats.

However if the application does not use date, time, or timestamp expressions and always displays the date using the same consistent format, the use of these data types can be costly.

Recommendation

For best performance with dates, times, and timestamps, choose one of the database manager data formats for the data element and use that format consistently throughout your application. Then use either char or string types for the date, time, or timestamp columns and set these build descriptor options to the same value specified for the DBM:

- defaultDateFormat
- defaultTimeFormat
- defaultTimeStampFormat

The DBM will format the data element in the chosen format when the element is retrieved from the table into a char or string host variable. EGL will format current date/time/timestamp values and values entered by the user in the specified formats. No other formatting will be required for the data element in the program.

If you want the application to support display of dates, times, or timestamps with different formats or to use date or timestamp arithmetic operations, then use date, time, and timestamp EGL types.

<table>
<thead>
<tr>
<th>SQL Column Type</th>
<th>EGL Type (best performance, assumes consistent format)</th>
<th>EGL Type – use with variable formats or date / timestamp arithmetic</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>char(10) or string</td>
<td>date</td>
</tr>
<tr>
<td>TIME</td>
<td>char(8) or string</td>
<td>time</td>
</tr>
<tr>
<td>TIMESTAMP</td>
<td>char(n) where n = 20 to 26, or string</td>
<td>Timestamp (“yyyyymmddhhmssffffff”) where fffffff is optional</td>
</tr>
</tbody>
</table>
EGL Best Practices: Coding For Performance

Match Null Specification

Background

SQL columns and EGL variables can be specified as being able to be null, that is to have no value. For some business attributes, a value of blank or 0 in a variable can indicate a value has not been specified for the attribute. However for other attributes, values of blank or 0 are valid values. For example, consider the attribute “number of children”. It’s useful to be able to distinguish between the attribute value being 0 (the person has no children) or null (the number of children is unknown).

Observations

The cost of null processing appears to negligible in Java environments, but has measureable overhead in COBOL environments.

Recommendation

Define SQL record field as nullable if and only if corresponding table column is nullable. When a SQL table is created, columns can be defined as “NOT NULLABLE”. These means a value is required for these columns. For an EGL variable, the ? modifier specified with the variable type indicates that the variable is nullable. If there is a mismatch between the two declarations, unnecessary processing can occur, or worse, SQL errors may be encountered.

Here is an example of a record. The columns holding id and name were defined as “NOT NULLABLE” when the table was created. The desc field is optional and defined as nullable.

```eql
record anSQLRecord type sqlRecord {
  tableNames = ["anSQLTable", "T1"],
  keyItems = [ id ]
}

id string ; // id is required (not nullable)
name string ; // name is required
desc string? ; // description is optional (nullable)
}
```

Good Practices for SQL in EGL Programs

Use Result Set Identifier

Background

A result set is the set of rows selected by an SQL query. The EGL open statement sends a query to the database manager. The EGL for each or get next are used to retrieve the rows in the set one at a time for processing.

VisualAgeGenerator connected the statements using the SQL record name specified on both statements. However in scenarios where there were multiple opens in the program for the same
EGL Best Practices: Coding For Performance

record, the generated program had to be smart enough to connect the statements to the result set that was currently opened.

EGL allows a handle called a result set identifier to be specified on the open statement that selects the result set and the foreach or get next statements that process the set.

Recommendation

Always specify the result set id on open and related positional get or foreach statements. For example:

```plaintext
aRec anSQLRecord ;
try
  open aResultSet for aRec ; // result set id specified
  while ( aRec not noRecordFound )
    get next aRec from aResultSet ; // id referenced
  end
onException ( ex anyException )
  writeStdOut ( "SQL Error = " + ex.message ) ;
end
```

Limit the Number of Rows Returned in a Result Set.

Background

A SQL query may select thousands of rows in a result set and prepare to return them to your program when your application is only set up to display 20 rows at a time.

Recommendation

For best performance, customize the SQL SELECT associated with the EGL open result set statement to limit the number of rows returned to the number needed in your application. For example if your page is limited to displaying 20 rows, only retrieve 20 rows from the database.

The best way to do this if you using a DB2 database is to add a FETCH FIRST n ROWS ONLY directive to your query. If you expect that the average number (m) of rows retrieved is less than n, add an OPTIMIZE FOR m ROWS directive as well.

For example, here is a library function that returns the next 20 rows in the database following the last displayed row id. Note that his example is using the get statement to read an array of records. The SQL statement in the example could be used just as well with an open statement.

```plaintext
function getNextFlexRowDynamicArray( aRecs flexSQLRecord[] inOut )
  dim int = aRecs.getSize ( ) ;
  id char(10) ;
  if ( dim == 0 )
    id = "" ;
  else
    id = aRecs[dim].id ;
```
EGL Best Practices: Coding For Performance

```ecl
end
get aRecs with
#sql{
    select
        id, text, Count, Age, cash,
        salary, START, due,
        moreText, longText
    from benchmarktable
    where id > :id
    order by id asc
    fetch first 20 rows only
} ;
end
```

If you use a database manager other than DB2, check the SQL reference for your DBM to find out the appropriate directive to use.

If you want to use syntax that is common to all DBMs, use the `maxsize` attribute when defining the array variable. The DBM can return a result set with more than 20 rows, but the generated program won’t fetch more than 20 rows from the result set. For example:

```ecl
bRecs anSQLRecord[] { maxSize = 20 };
try
    get bRecs ;
onException ( ex anyException )
        writeStdOut ( "SQL Error = " + ex.message ) ;
end
```

Implement Programmatic Paging for Interactive User Interfaces

**Background**

In transactional environments like the IBM Websphere Application Server or CICS or IMS, many users are supported out of a limited supply of heap or region memory. More users can be supported when the program supporting each user reads only those rows that are currently going to shown on the user interface.

**Recommendation**

If your application is implemented as a JSF handler or Web transaction, or as a TUI with a 3270 interface running on CICS or IMS, you will want to implement programmatic paging for lists in scenarios where the number of rows the returned by a database query could be greater that the number of rows displayed on the page or screen.

For TUI applications, the number of row displayed is determined by the size of the screen. For JSF and Web transaction pages, the maximum number of rows you want to display can be estimated using a “rule of thumb” that states that the maximum number of cells displayed in a data table should be less than 10000. This means that if you want to display 20 elements in a row, then the maximum number of rows on the page should be 10000 / 20, or 500.
EGL Best Practices: Coding For Performance

Look for examples how to implement programmatic paging in the RBD/EGL Foundation Language Course on the EGL Café.

**Good Practices for SQL in COBOL Environments**

If your program is ever going to be generated to run as a COBOL program, consider the following practices:

**Specify singlerow When Retrieving a Single Row**

**Background**

Static SQL used in COBOL programs provides a special form of the SQL Select statement call single row select which is more efficient when you know your query is going to retrieve only one row (for example, a key equal query for an column that doesn’t allow duplicate values). EGL uses this form in the generated application if the singlerow modifier is used on the EGL get statement.

There is no special API for single row selection in Java. Specification of the singlerow modifier is ignored in Java environments.

**Recommendation**

Use the singlerow option on get statements intended to retrieve a single row with a key that is unique in the table. For example:

```
get aRec singlerow ;
```

**Avoid Using prepare Statement**

**Background**

SQL statements in COBOL provide the best performance when they can be prepared by the DB2 precompiler prior to the COBOL compilation. Using the EGL prepare statement to prepare SQL statements at runtime bypasses the DB2 precompiler.

**Recommendation**

Don’t use the prepare statement in applications intended for COBOL environments.

**Good Practices for SQL Statements In Any Language**

**Use SQL Indices**
Background

For frequently executed queries or queries against tables with large numbers of rows, make sure you are operating on table columns which have indices defined.

Recommendations

- Avoid using SQL statement operations like ORDER BY unless the ordering uses an index already sorted in the order you need. Operations that result in sorting are:
  - ORDER BY
  - GROUP BY
  - TABLE JOINS
  - UNION

- Watch out for query keywords that don’t use the index. SQL WHERE clause keywords that disallow index access are heavy consumers of resources. Examples are:
  - Computations in the WHERE clause
  - LIKE operator with leading % or underscore in the pattern

Use SQL Stored Procedures

Background

Some database intensive processing can be implemented as a database stored procedure and offload processing to an underutilized database server. EGL provides the ability to code a stored procedure call in the EGL program using the EGL open statement for stored procedure calls that return a result set and the EGL execute statement for stored procedure calls that use simple parameters.

Recommendation

For your most performance-intensive pages, consider using SQL stored procedures for database access.

Web Coding Practices

Watch this space for information in the future…

Disclaimers
EGL Best Practices: Coding For Performance

The information contained in this documentation is provided for informational purposes only. While efforts were made to verify the completeness and accuracy of the information contained in this documentation, it is provided “as is” without warranty of any kind, express or implied. IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this documentation or any other documentation. Nothing contained in this documentation is intended to, nor shall have the effect of, creating any warranties or representations from IBM (or its suppliers or licensors), or altering the terms and conditions of the applicable license agreement governing the use of IBM software.

IBM, the IBM logo, VisualAge, zOS Language Environment, zSeries and DB2 are trademarks of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.