Best Practices for Informix Developers

Art S. Kagel
ASK Database Management

Best Practices for Informix Developers

Abstract:

We will explore the various ways to skin the development cat, examining several typical application module scenarios. Along the way, we will look at how to develop better applications that encounter fewer processing bottlenecks and take best advantage of IDS's features.

Best Practices for Informix Developers

- Application Development Languages
- Application Performance
- New Informix & Informix-Related Features

Best Practices for Informix Developers

Application Development Languages

IBM Development Language Support:

- Informix C Software Development Kit (CSDK)
  Current release CSDK 4.10.xC4
- Embedded SQL in COBOL (ESQL/COBOL)
  Latest Version: 7.25.xD4
- IBM Universal Driver for JDBC & .NET
- Enterprise Generation Language (EGL)
- Informix Compiled 4GL (Current: 7.51)
- Informix 4GL Rapid Development System
- Informix SQL (Current: 7.51)

Best Practices for Informix Developers

Application Development Languages

IBM Development Language Support:

Informix C Software Development Kit (CSDK) features:

- ESQL/C compiler driver and libraries
- CLI / ODBC library
- OLE DB Interface
- Informix JDBC Driver
- Informix .NET Provider
- Object Interface for C++ (No longer supported)

Best Practices for Informix Developers

Application Development Languages

IBM Development Language Support:

Informix SQL (ISQL) features

- Perform Forms Manager
  Simple data display and maintenance forms
- ACE Report Writer
  Text report generator using a forms-like interface
- SQL Menu Manager
  Menus to tie multiple ACE and Perform modules and other application modules together into simple applications

Best Practices for Informix Developers

Application Development Languages

IBM Development Language Support:

Informix 4GL features

- Native code and p-machine based compiler options
- Latest release integrates to WEB/SOAP (7.50+)
- High level 4th Generation Language with embedded SQL
- Event driven forms control modules
- Event driven report control modules

Best Practices for Informix Developers

Application Development Languages

Third Party Informix 4GL Language Support:

Three vendors support extended x4GL language:

Four Js

Querix

Aubit 4GL – Open Source

Best Practices for Informix Developers

Application Development Languages

Third Party Informix 4GL Language Support:

4Js Genero
GUI - Fully graphical XML based display system
Windows, Mac OS/X, Linux
Full WEB Integration
Ajax, HTML5
Web services/components, SOA
Mobile Client
Language extensions (including embedded Java)
Virtual Machine/Byte code compiler
Deploy single executable on any platform
Supports all major RDBMS servers (but no SQL dialect mapping)
Graphical report writer (proprietary)
Graphical IDE tailored to 4GL language (Proprietary - Eclipse-like)
4GL Code Generator Option

Best Practices for Informix Developers

Application Development Languages

Third Party Informix 4GL Language Support:

Querix

- **Lycia II Development Workbench (Eclipse based IDE)**
  - **Lycia WindowBuilder - New graphical form designer**
  - **Other productivity plugins: CVS, BIRT, Jasper, etc.**
- **ESQL/C Compiler included for x4GL/ESQL-C integration**
- **Support for most RDBMS servers**
- **Dynamic SQL dialect conversion**
- **WEB, JAVA, SOAP, REST, ADO/.NET, SAX integration**
- **BIRT & Jasper report writers integrated**
- **Language extensions including full Genero extension import**
- **Graphical Thin Clients**
  - **Desktop**
  - **Web**
  - **Mobile**
- **Create WEB services from any 4GL function**

Best Practices for Informix Developers

Application Development Languages

Third Party Informix 4GL Language Support:

Aubit4GL

- **Support for most RDBMS servers**
- **Near complete 4GL source compatibility**
- **Character, XML, GUI User Interfaces**
- **PDF report output extension**
- **Language extensions**
- **Embedded/inline C & ESQL/C support extension**
- **Dynamic SQL dialect conversion (customizable)**
- **Open Source and free to develop and distribute**
- **Commercial support options available**
- **Many tools included**
  - **ISQL-like forms and reports**
  - **Portable dbaccess clone**
  - **More...**

Best Practices for Informix Developers

Application Development Languages

Scripting Language Support:

- Shells and other scripting languages with sqlcmd or dbaccess
- Perl with DBD/DBI Informix
  http://search.cpan.org/~johnl/DBD-Informix-2008.0513/Informix.pm
- Ruby / Informix
  http://ruby-informix.rubyforge.org/
- PHP Informix
  http://php.net/manual/en/book.ifx.php
- GO Programming Language
- Python

Best Practices for Informix Developers

Application Development Languages

MongoDB Wire Listener Protocol

Supports development stacks for MongoDB:
MEAN - MongoDB, ExpressJS, AngularJS and Node.js
Library level APIs for: C, C++, C##, Java, PHP, etc.

Best Practices for Informix Developers

Application Development Languages

REST Protocol Listener:

Supports any development tools that support HTML

Best Practices for Informix Developers

Application Development Languages

On to the meat

?

Best Practices for Informix Developers

Application Performance Tips

INSERTs of large numbers of rows should use INSERT CURSOR:

```
EXEC SQL
    DECLARE ins_1 CURSOR FOR
       INSERT INTO mytable( col1, … )
     VALUES (?, ?, … );
EXEC SQL
    OPEN ins_1;
for (n_ins=0;;) {
    EXEC SQL PUT ins_1 USING var1, var2, ....;
    n_ins++;
    .....
    if (n_ins > 100) EXEC SQL FLUSH ins_1;
}
```

Best Practices for Informix Developers

Application Performance Tips

INSERTs of large numbers of rows should use INSERT CURSOR:

```
/* Number of rows successfully flushed by PUT to a full buffer
   or by FLUSH */

    num_inserted = sqlca.sqlerrd[2];
```

Best Practices for Informix Developers

Application Performance Tips

- Increase INSERT CURSOR performance by increasing the buffer size:
- Environment:

  export FET_BUF_SIZE=32765

  Range: 4096 – 2^31 (Default: 4096)
- Global variable:

  extern int FetBufSize = 32765;
  (Still limited to 16 bits – open PMR)

Best Practices for Informix Developers

Application Performance Tips

Complex Upsert or Merge Operations:

- Failed updates are cheap
- Failed inserts are expensive

Best Practices for Informix Developers

Application Performance Tips

Complex Upsert or Merge Operations:

Rule of thumb:

- If fewer than 70% of operations are INSERTs, try UPDATE first
  - If zero rows were updated then INSERT.
- If 70% or more are INSERTs, try INSERT first
  - If duplicate key error then UPDATE.

Best Practices for Informix Developers

Application Performance Tips

Complex Upsert or Merge Operations:

Or you can use the new MERGE statement!

```
LOAD FROM "data.fil" INSERT INTO TEMP stager( … );
MERGE INTO DataTable AS d
USING stager AS s
ON d.key1 = s.key1 AND d.key2 = s.key2 …
WHEN MATCHED UPDATE SET d.attr1 = s.attr1, d.attr2 = s.attr2
WHEN NOT MATCHED INSERT (key1, key2 ..., attr1, attr2, …)
   VALUES (s.key1, s.key2 …, s.attr1, s.attr2, …
;
```

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking

WHAT IS OPTIMISTIC LOCKING?

Assume no two users will be modifying the same row at the same time.

If it happens that a row which a session is modifying has been changed by another session, then deal with it at that point.

Reduces lock contention.

Best Practices for Informix Developers

The biggest hurdle to good multiuser application performance is contention!

The biggest hurdle to good data quality is poor contention management techniques!

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking

Basic Optimistic Locking Protocol:

- Fetch row without lock (no FOR UPDATE clause)
- Present a copy of the row for the user to modify
- Fetch current stored version of row with lock (with FOR UPDATE OF CURRENT clause)
- Validate original version against current version
  - If no changes – update and commit
  - Else rollback work and notify user

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking Implementation

Version I:

Every table has a DATETIME YEAR TO FRACTION(5) column* included that defaults to CURRENT and an UPDATE trigger that maintains the column when the row has been updated.

Only have to fetch this timestamp column to validate the current version of the row against the original.

* To get sub-second timings you MUST turn on USEOSTIME in the ONCONFIG
Actual resolution depends on your OS & processor architecture!

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking Implementation

Version II:

Add CRCOLS to every table

- Adds two columns cdrserver & cdrtime which are automatically maintained by IDS.

Add REPLCHECK column to every table

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking Implementation

Version III:

Add VERCOLS to every table

Adds two columns, ifx_insert_checksum and ifx_row_version uniquely identify the version of a row. Changes in the value of the ifx_insert_checksum column will identify rows that have been deleted and reinserted. The ifx_row_version column is incremented anytime the row is updated.

Best Practices for Informix Developers

Application Performance Tips

Avoid lock contention using Optimistic Locking Implementation

Use COMMITTED READ isolation with LAST COMMITTED option in combination with other methods to reduce lock contention to zero.

Even if there is a badly behaved application holding locks, or a user happens to query a row just as it is being updated, the user will not be locked out or have to wait but will return the latest committed version of the row. Optimistic locking protocol prevents the user from updating an out-of-date version.

Best Practices for Informix Developers

Application Performance Tips

Take advantage of Array Fetching.

- IDS ALWAYS sends as many rows as fit in a communications buffer (FetBufSize 4K-2GB) in a single operation.
- ESQL/C, ODBC, 4GL, Java, and other library interfaces deblock the communication buffer to return a single row from this buffer with each CURSOR FETCH until it is drained.
- Lots of overhead to have the library perform this deblocking for you.

Best Practices for Informix Developers

Application Performance Tips

- Array Fetching returns the entire block of rows into your application's memory space into arrays of memory one for each column fetched.
- You can usually deblock the data more efficiently than the Informix libraries since you can write data specific code!

* There is a bug that prevents using Array Fetch with LVARCHAR columns and UDTs (SQCODE = -1831)

Best Practices for Informix Developers

Application Performance Tips

Implementing Array Fetching:

1. Set FetBufSize or FET_BUF_SIZE large enough to hold a reasonable but large number of rows
2. Set FetArraySize to the number of rows that fit in the FetBufSize buffer
3. Prepare and Describe the query into an sqlda data structure
4. Place a pointer to an array of memory big enough to hold FetArraySize values into the sqlda.sqlvar[col_ord].sqldata for each field being returned
5. Execute: FETCH USING (*sqlda)
6. Pull the data out of the array to reference a single row

Best Practices for Informix Developers

Application Performance Tips

Implementing Array Fetching:

```
FetBufSize = 32765;
FetArrSize = FetBufSize / rowsize;
EXEC SQL PREPARE my_stmt FROM "SELECT tabid, tabname FROM systables";
EXEC SQL DESCRIBE my_stmt INTO sqlda_str;
...
sqlda_str.sqlvar[0].sqldata = malloc( FetArrSize * sizeof (int) );
sqlda_str.sqlvar[1].sqldata = malloc( FetArrSize * 129 );
…
EXEC SQL DECLARE mycurs CURSOR FOR my_stmt;
EXEC SQL OPEN mycurs;
```

Best Practices for Informix Developers

Application Performance Tips

Implementing Array Fetching:

```
while(1) {
   int *tabid = (int *)sqlda.sqlvar[0].sqldata;
   int *tabnm[129] = (char *)sqlda.sqlvar[1].sqldata;
   EXEC SQL FETCH mycurs USING DESCRIPTOR :sqlda_str;
   if (sqlca.sqlcode = 100) break;
   nrows_returned = sqlca.sqlerrd[2];
   for (row=0; row < nrows_returned; row++) {
      printf( "Tabid: %d Name: %s.\n", tabid[row], tabnm[row] );
      …
   }
   …
}
```

Best Practices for Informix Developers

Application Performance Tips

Two externally controlled optimizations are available

Best Practices for Informix Developers

Application Performance Tips

Deferred Prepare Optimization:

- Enabled with SQL command or environment variable:
  - SET DEFERRED_PREPARE ENABLED;
  - export IFX_DEFERRED_PREPARE=1
- Defers actual PREPARE of a statement until it is either executed or a cursor is opened against it. The PREPARE statement only registers the statement to be prepared later.
- Can reduce the number of round trips the application makes to the server.
- Not much gain if a statement is to be prepared once and reused many times.
- Since the PREPARE doesn't actually happen until the OPEN, any DESCRIBE must be executed following the OPEN not the PREPARE.
- Syntax and undefined object errors are also delayed until the OPEN.
- Can't use Deferred Prepare with Array Fetch.

Best Practices for Informix Developers

Application Performance Tips

OpenFetchClose optimization:

- Enabled by setting environment variable:
  export OPTOFC=1
- Delays cursor declare and open until the first FETCH.
- Automatically closes the cursor when it has been drained.
- If combined with Deferred Prepare the PREPARE happens at the time of the first fetch as well.
  - **DESCRIBES must then follow the FETCH (but only for the first iteration).**
  - **Syntax and undefined object errors will only be returned by the first FETCH not by the PREPARE or OPEN.**
- Can't use Array Fetch feature with OpenFetchClose optimization.
- Closing a cursor frees it also, so reopening the cursor will return an error.

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

TEXT and BYTE type column data can be returned three ways. In order from slowest to fastest these are:

- Fetch to memory (user or library allocated)
- Fetch to a file or file descriptor
- Fetch using user defined open, read/write, and close functions

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

To fetch using User Defined Functions:

1. DESCRIBE the statement
2. Modify the locator structure (loc_t) for the Blob column so that the loc_loctype field contains the define LOC_USER
3. Define the blob location functions and assign a pointer for each to the appropriate loc_t field:

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

To fetch using User Defined Functions:

1. loc_open()
   - Called when the cursor is opened.
   - Prepare for data transfer here.
2. loc_read()
   - Called from INSERT and UPDATE statements to 'read' in blob data from user space.
   - Called many times until the entire blob has been transferred.
3. loc_write()
   - Called from SELECT statements to 'write' blob data into user space.
   - Called until the entire blob has been transferred.
4. loc_close()
   - Called when the cursor is closed.
   - Finalize blob handling here.

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

An open function:

```
int openit( loc_t *loc, int flags, int bsize )
{
    if ((flags & LOC_WONLY) && loc->loc_size < -1) {
       fprintf( stderr,
                "Blob opened with length: %d.\n", loc->loc_size );
       return -1;
    }
    loc->loc_status = 0;
    loc->loc_xfercount = 0L;
    global_length = 0 ;
    return 0;
}
```

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

A Write Function:

```
/* BLOB Fetch function. */
int writeit( loc_t *loc, char *buf, int nbytes )
{
    unsigned long toread;
    if ((bpos + nbytes) > bbsize) {
       bbsize += (2 * (nbytes < 1024 ? 1024 : nbytes));
       blobbuff = (char *)realloc( blobbuff, bbsize );
    }
    if (loc->loc_size != -1)
       toread = min( nbytes, (loc->loc_size - loc->loc_xfercount) );
    else
       toread = nbytes;
    memcpy( &blobbuff[bpos], buf, toread );
    bpos += toread;
    loc->loc_xfercount += toread;
    global_length += toread;
    return toread;
}
```

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

A Read Function:

```
/* BLOB Insert function. */
int readit( loc_t *loc, char *buf, int nbytes )
{
    unsigned long towrite;
    towrite = min( nbytes, (loc->loc_size - loc->loc_xfercount) );
    memcpy( buf, &blobbuff[bpos], towrite );
    bpos += towrite;
    loc->loc_xfercount += towrite;
    global_length += towrite;
    return towrite;
}
```

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

A Close Function:

```
int closeit( loc_t *loc )
{
    loc->loc_status = 0;
    if (loc->loc_oflags & LOC_WONLY) {
        /* Fetching, cleanup. */
        loc->loc_indicator = 0;
        loc->loc_size = loc->loc_xfercount;
    }
    return 0;
}
```

Best Practices for Informix Developers

Application Performance Tips

Simple Large Objects

Setup:

```
col->sqldata = (char *)&(locators);
locators.loc_loctype = LOCUSER;
locators.loc_bufsize = 4096;
locators.loc_buffer = (char )blobbuff;
locators.loc_write = writeit;
          (-or, for inserts- locators.loc_read = readit; )
locators.loc_open = openit;
locators.loc_close = closeit;
```

Best Practices for Informix Developers

Application Performance Tips

Smart Large Objects

BLOB and CLOB type column data are handled by a set of library functions:

- ifx_lo_create, ifx_lo_open
  - Create and open or open existing SLOB
- ifx_lo_from_buffer, ifx_lo_write, ifx_lo_writewithseek
  - Write data into a SLOB
- ifx_lo_to_buffer, ifx_lo_read, ifx_lo_readwithseek
  - Read data out of a SLOB
- ifx_lo_close
  - Close an open SLOB

Best Practices for Informix Developers

What else
do you want
to hear about?

Best Practices for Informix Developers

SPL Pearls

- Extensive procedural code written in SPL tends to be slower than equivalent host language application code.
- UDRs written in “C” or Java tend to perform better than SPL routines. However, SPL routines that primarily implement an SQL query perform very well.
- Avoid deeply nested SPL routines.
- Dynamic SQL in SPL routines perform well.

Best Practices for Informix Developers

SPL Pearls

Stored procedures that process extensive quantities of data, especially if they provide advanced data filtering, aggregation or other services which reduce the quantity of data that needs to pass back and forth between the RDBMS and the client application, can provide large performance improvements versus performing the same functionality in a client application just by reducing the communication traffic.

Best Practices for Informix Developers

General Pearls

Disconnect the client application from the data representation.

- Schema changes can be hidden from client code more easily.
- Reduces or eliminates need to coordinate code and schema changes.

Very important for developing mobile apps to be able to support multiple versions of the app and database schema!

Best Practices for Informix Developers

General Pearls

Carefully written fully dynamic SQL is independent of the data representation. It can:

- determine number and types of data elements required as input and/or returned as output.
- dynamically bind data to host data storage.

Poorly written dynamic SQL can be more closely adhered to data representation than embedded SQL.

Best Practices for Informix Developers

General Pearls

Interdependency Level	Description
I (low)	SQL and related business logic encapsulated in stored procedures
II	SQL and related business logic encapsulated in middleware
III	SQL only encapsulated in middleware
IV	SQL embedded in application source
V (high)	Dynamically generated or user provided SQL passed to the RDBMS or middleware server

Questions

?

Best Practices for Informix Developers

Art S. Kagel

art.kagel@gmail.com
art@askdbmgt.com

Web site: www.askdbmgt.com
Or: www.ibminformix.guru