Multi-row FETCH in V8 provided a significant performance boost, but what about multi-row INSERT? It can significantly boost inserting performance and reduce CPU usage. Converting existing programs involves some serious structural changes to both SQL and program logic. Learn from those who have gone before into the world of multi-row INSERT. In this session you will begin to understand the power and pitfalls of multi-row INSERT.

Objective 1: Explain the concept of how multi-row INSERT works
Objective 2: Describe some of the reasons and implications of using multi-row INSERT
Objective 3: Specify the changes required to implement multi-row insert in your COBOL programs
Objective 4: Identify potential pitfalls with; unit of work, error handling, array processing, back out processing
Objective 5: Describe the way this has been successfully implemented in one business
Points to Ponder
1. Our success story
2. Performance; reasons to explore multi-row SQL
3. Multi-row Insert concepts and syntax
4. Potential Pitfalls
5. Multi-row Insert; Our Reality
With every new version of DB2, we receive new features or enhancements which potentially provide performance gains. Some we gain automatically such as optimizer enhancements. Others we gain through DB2 object enhancements such as new tablespaces or compression. Still others are through application changes within new SQL which has to be implemented.

Multi-row Fetch and Insert can provide significant performance gains, but we first must take time and money to invest in these performance gains.
Our Success Story with Multi-Row fetch.

We know moving on to DB2 V8 would take some extra machine resources, but we also know how we could regain them by implementing multi-row fetch into some of our heavier batch applications. Here is our success story.
There are two reasons to take advantage of the multi-row fetch capability:

1. To reduce the number of statements distributed between your program address space and DB2.
2. To reduce the number of statements distributed between DB2 and the DDF address space.

The fewer trips made across the application address spaces and DB2, the lower the overall usage of the machine cycles.
IBM’s Multi-row Fetch performance chart. This chart graphics out CPU time for single-row and multi-row fetch based on 100,000 rows fetched with different rowset sizes. This shows where the performance gains bottom out.

We decided to utilize multi-row fetch max rowset size of 100.
There are two reasons to take advantage of the multi-row insert capability:

1. To reduce the number of statements distributed between your program address space and DB2.
2. To reduce the number of statements distributed between DB2 and the DDF address space.

The fewer trips made across the application address spaces and DB2, the lower the overall usage of the machine cycles.
Up to 30% faster INSERT performance

1. Performance improvement largely due to savings of API costs

2. Savings flattens out quickly, for example, savings (as a percentage) was equal
   - For 100 Rows, 500 Rows, 1000 Rows…
   - Reasonable “n” for MRI is about 200 – no additional savings above that, and downside (rollback) increases.

3. Distributed MRI performance up to 70% Elapsed Time and 50% Server CPU time reductions seen

4. Performance variable based on
   - Number of rows inserted
   - Number of columns inserted
   - Number of indexes on table
   - Class 2 accounting (on or off) – savings larger is Class 2 is on

Note:
Don’t use MRI to INSERT 1 row due to overhead to set up for MRI. Similar improvement with UPDATE and DELETE WHERE CURRENT OF when updating/deleting the entire rowset. This is in addition to the savings provided by MRF.
Discussion Points for Multi-Row Insert

1. Rowset; what is it.
2. A look at Host Variable Arrays
3. The new Insert syntax
4. Get Diagnostics and error handling
1. A group of rows returned by a single FETCH statement or inserted by a single multi-row INSERT statement
2. Controlled by the application – FETCH… FOR n ROWS or INSERT… FOR n ROWS
3. Minimum 1, maximum 32,767
4. Performance implications based on RowSet Size – see benchmarks (covered earlier)
5. Each group is operated on as a set.
Example for declaring a host variable array within Cobol.

Host variable arrays are used to pass a data array between DB2 multi-row SQL calls and your application. Remember that each column/element within the table must be an array itself. We can not use an array of structures. Variable character strings are not an exception to this rule because it is considered a single column within the table structure and DB2 works on each column independently within the DCLGEN.
New Insert Syntax

DB2 V8 Insert Syntax
Additions to the SQL Insert statement

1. Allow to add host-variable array into the values clause

2. New clause “For X Rows” – This clause is used when inserting HV-Arays and determines the number of entries to be inserted. The value for X can be an integer constant or an integer host-variable.

3. Atomic or Not Atomic - This clause is used when inserting HV-Arrays and determines how DB2 will process the statement.
   1. Atomic tells DB2 to process all or nothing. If one row errors within the HV-Array, then the entire statement errors and nothing is processed. The insert statement then returns one error message.
   2. Not Atomic tells DB2 to process each statement individually. If any of the HV-Array rows should fail on the insert the statement continues its execution and those rows whom failed are returned with the Get Diagnostic statement. The SQL Get Diagnostics statement determines which row (or rows) were not successfully inserted. The SQLCode will indicate if all failed, all were successful or at least one failed.

Note:
When using Atomic (all or nothing) insert with large rowsets could impact your application performance. Inserting 50 or 100 is fine but trying to insert 10,000 rows and you hit an error on row 9000, everything will have to be rolled back and don’t forget about the locks that will have to be held on the table.
Atomic : (All or Nothing)

Not Atomic : (Independent Inserts)
Get Diagnostics replaces or extends the diagnostic information available in SQLCA.

Supports SQL error message tokens longer than 70 bytes (SQLCA limitation)

Cannot be dynamically prepared – static only

Returns information in 3 different areas:
- Statement information - *Information about the SQL statement*
- Condition information - *Information about conditions or connections*
- Combined information - *Textual representation of all information*
GET DIAGNOSTICS is the standard way of checking the result of a multi-row operation. In the case of multi-row fetch, however, it is not necessary. The SQLCODE and SQLERRD3 fields of the SQLCA can be used to get the return code and number of rows fetched. Given the high cost of the GET DIAGNOSTICS using the SQLCA should be the way to go.

There is no pressing need to call GET DIAGNOSTICS for any successful (i.e. SQLCODE = 0 or +100) retrieval. GET DIAGNOSTICS is very expensive (in one situation I tested it was 3x the cost of the insert statement I had it monitoring). You only want to call it when there is something you need that isn't provided in the SQLCA. If you are fetching and get a SQLCODE +100 and all you want to know is how many rows were in the last fetch, use SQLERRD(3).
Some values from Get Diagnostics we currently use in our Error Handling.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER</td>
<td>DB2_RETURNED_SQLCODE – Returns the SQLCODE for the specified diagnostic.</td>
</tr>
<tr>
<td></td>
<td>RETURNED_SQLSTATE – Returns the SQLCODE for the specified diagnostic.</td>
</tr>
<tr>
<td>ROW_COUNT</td>
<td>DB2_REASON_CODE – Returns the reason code for the specified diagnostic.</td>
</tr>
<tr>
<td></td>
<td>DB2_ROW_NUMBER – Returns the row number where the condition occurred.</td>
</tr>
<tr>
<td></td>
<td>MESSAGE_TEXT – Returns the message text associated with the SQLCODE.</td>
</tr>
</tbody>
</table>
Error Handling

Get Diagnostics Example

** DO NOT STORE FIRST ERROR CONDITION (MAIN SQLCODE) WITHIN THE DIAGNOSTIC ARRAY. ONLY STORE THE ACTUAL SQL ERRORS IF WS960S-COND-NUMBER NOT EQUAL 1

** WE START AT THE SECOND ERROR CONDITION BUT MUST STORE SECOND ERROR WITHIN THE FIRST POSITION OF THE DIAGNOSTIC ARRAY

** ARRAY

EXEC SQL
GET DIAGNOSTICS CONDITION :cond
:sqlcode = DB2_Returned_SQLCode
:sqlstate = Returned_SQLState
END-EXEC

END-IF
Pitfall Examples:

Using multi-row insert on a table that contains a timestamp column I receive a SQLCODE -180. Even though I'm positive that the value I'm trying to insert is correct. The error message is: "THE DATE, TIME, OR TIMESTAMP VALUE *N IS INVALID". Why?

You may have inserted data pass you array boundary or you’re trying to insert data that hasn’t been populated into your array. Check “FOR n Rows” and determine if you’re outside what you’ve populated in the HV-Array.

Using multi-row insert on a table that contains a timestamp column which is also part of the unique index, I receive a -803. I’m using CURRENT TIMESTAMP for the value so why would this be a duplicate entry?

The CURRENT TIMESTAMP function is only retrieve once on the insert, therefore the timestamp will be the same for all rows inserted. To eliminate this you can wrap the scalar function GENERATE_UNIQUE around the CURRENT TIMESTAMP value and DB2 will provide a unique timestamp on insert.

I’ve converted an existing batch application which, handle millions of inserts for each execution, into a multi-row insert so we can take advantage of the performance enhancements. Everything is functioning properly within the application, but now I cause deadlocks between myself and other applications using the same table. This wasn’t occurring prior to the conversion. My commit logic did not change, so why would this start happening?

This could be a result of your rowset size. The higher your rowset the more locks you will hold.
Our Multi-Row Fetch and Insert methodology.
Our Plan

1. Don’t change any code that isn’t necessary. Limit code changes.
2. Must determine how we can control a unit of work properly
3. These changes should fit within our existing framework/ architecture
4. Let's keep the changes simple.
Multi-Row Insert Design

• Our Design
  • Make necessary SQL syntax changes
  • Handle input of the SQL
  • Add pre-SQL code
    • To execute or Not to execute the SQL
  • Add post-SQL code
    • Handle Errors
Pre-SQL Code:
Initialization and decision - If we’ve reached the end of the array then EXECUTE SQL Else manage HVA

Post-SQL Code:
Error checking and array management.

Assign Variables:
Move current host-variable array values into our existing host-variable array. By moving to the old existing HV value, none of the old existing code has to change.
Assign Variables
    Move host variable values into the host variable arrays.
    Manage subscripts

Pre-SQL Code:
    Decision: If HVA full or committing a Unit of Work then EXECUTE SQL Else manage HVA

Post-SQL Code:
    Error handling using Get Diagnostic and building of our custom Diagnostic Array
    Use custom Diagnostic Array for what we wish to handle; ie -803 Dup Row.
Using a common diagnostic array simplifies the diagnostic statement and allows for a common standardize error handling storage.
Post insert Error Handling – our custom post insert routine.
Unit Of Work Design

- **Unit of Work**
  - **Commits**
    - When do we commit?
    - How to we flush the HV-array?
    - Is our UOW a true UOW?
  - **Rollbacks**
    - Must we clear the HV-array?
    - Are we rolling back?

Commit Questions:

1. When do we commit?
   
   *The application should dictate when. We just need to be able to handle the true UOW on any commit.*

2. How to know when we should flush the HV-array into the table. Even when its not full?
   
   *We added an exit to our common commit logic. So we must handle pre-commit logic in any of our multi-row insert applications. The pre-commit area is fully customizable per application.*

3. Are we assured our unit of work is a true unit of work?
   
   *As long as we know and receive control when a commit is processed.*

Rollback Questions:

1. Must we clear the HV-array on a rollback?
   
   *Only if we plan on handling the rollback. The HV-Array data is not removed on a rollback.*

2. Are we rolling back? How do we know?

   *We added an exit to our common rollback logic. So we can handle pre-rollback logic in any of our multi-row insert applications. The pre-rollback area is fully customizable per application.*
Handling program writes within DB2 and with arrays in memory can become very complicated and tricky. Handling row by row and write for write directly with DB2 only is simple and is easily understood. But once we incorporate multi-row inserting and management of host-variable arrays along with singleton updates or deletes, things can begin to become very complex and hard to comprehend. When working with multi-row insert we must completely understand how and when we are to handle a commit or rollback on our unit of work.

We manage our unit of work on a commit by establishing a “Pre-Commit Exit”. This give the programmer an extension to our common commit routine to customize before the commit is executed. In the pre-commit routine a programmer can call his insert routine and set a flag which will force the array data to be inserted in the table before the actual commit is performed.

We provide the same similar exit for “Pre-Rollback Exit” to add customized code before the rollback is executed. This allow programmer to clear the host variable arrays and reset any subscripting.
What we learned.

- Performance of the multi-row Insert
- Converting to multi-row insert with little code changes
- Error Handling
  - Get Diagnostics
  - Building Diagnostic Array
- Unit of Work
  - Commit
  - Rollback
Thank you for attending.