

Multi-Row Fetch

Proven Benefits and Usage Considerations

By Susan Lawson and Dan Luksetich, YL&A

This brief article looks at usage of multi-row fetch, and where the true benefits can be gained by using this feature. It will also look at some interesting usage considerations when programming to use multi-row fetch.

The multi-row fetch capability was introduced to DB2 z/OS in Version 8. Since that time we have found great uses for it and have done testing to see at what point the maximum performance benefits are achieved, and if there are any considerations given the type of workload.

Multi-row fetching gives us the opportunity to return multiple rows (up to 32,767) on a single API call with a potential CPU performance improvement somewhere around 50%. It works for static or dynamic SQL, and scrollable or non-scrollable cursors. There is also support for positioned UPDATES and DELETES. The sample programs DSNTEP4 (which is DSNTEP2 with multi-row fetch) and DSNTIAUL also can exploit multi-row fetch.

By now we all know what they are, so let's talk about some real life experiences with them. We will look at the various reasons why to use them and any caveats of doing so.

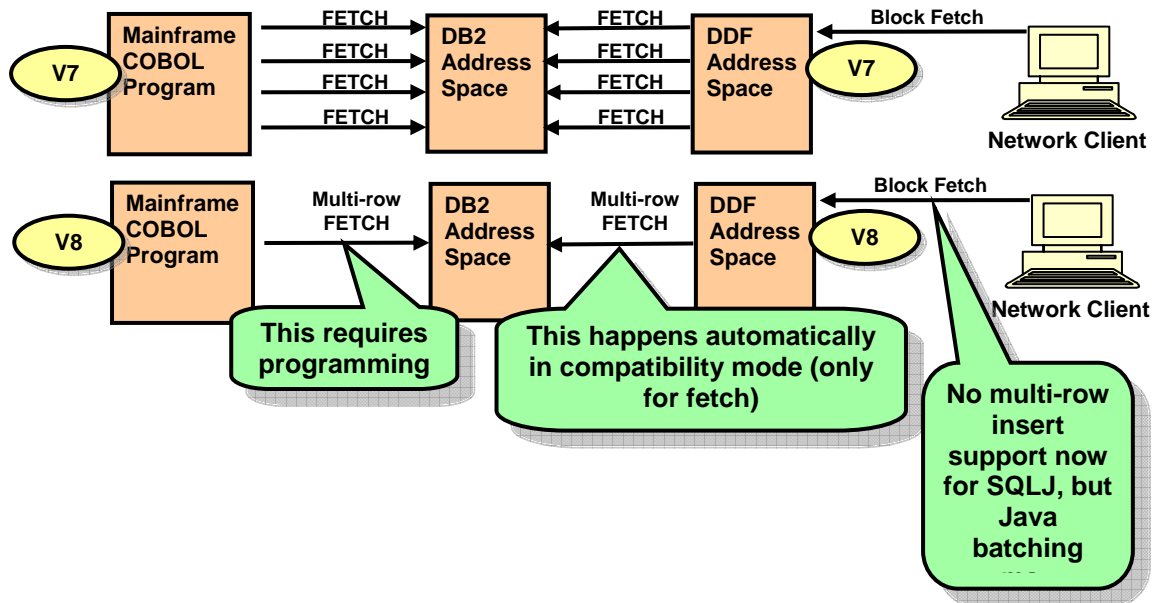
Advantages

First, there are two ways to take advantage of the multi-row fetch capability:

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

Why does this matter? Well every time you leave one address space to go to another it cost you 'DB2 Dollars' or 'MIP Money'. The less trips we have the lower the overall cost of our application. If you look at figure 1, the trips can be reduced between the address space for the mainframe COBOL application and the DB2 address space. This occurs when we code the new multi-row fetch statement using rowset positioning in our program. The second way to take advantage of multi-row fetch is in our distributed applications that are using block fetching. Once in compatibility mode in V8 the blocks used for block fetching are built using the multi-row capability without any code change on our part. This results in great savings for our distributed SQLJ applications. In one situation the observed benefit of this feature was when a remote SQLJ application migrated from V7 to V8 it did not have a CPU increase.

Figure 1. Multi-row fetch for local and distributed applications

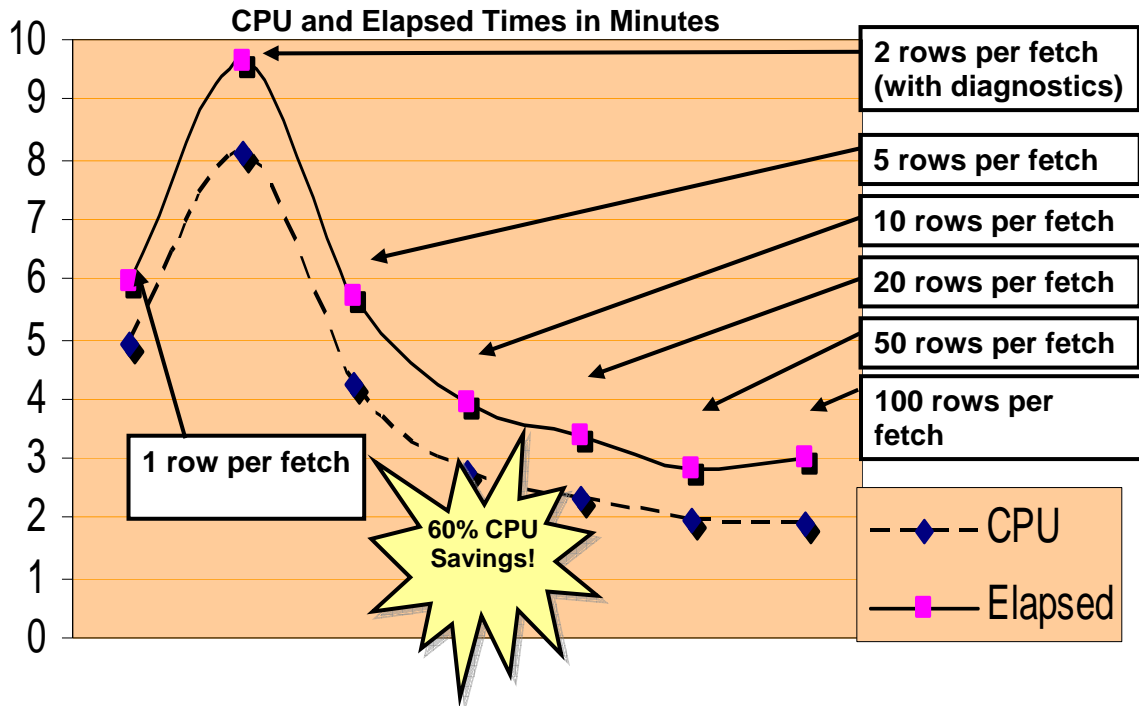


Performance Testing

Ok, so now we know why we should take advantage of this, so how do we know we'll use the feature efficiently? By testing.

Below are the test results in Figure 2 (courtesy of the rigorous testing of Dan Luksetich of YL&A) of using multi-row fetch with a local COBOL program performing a sequential read of approximately 39 million rows with 5 columns being returned. As you can see, the benefits increase with the more rows that are fetched with a multi-row fetch operation, however the point of diminishing returns is around 100 rows. Right around 100 rows the performance benefit seen was a 60% reduction in CPU.

Figure 2. Local COBOL Program w/Sequential Read of 39 Million Rows



Notice in the chart in Figure 2 the impact of using the GET DIAGNOSTICS statement. 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.

The aforementioned test was done with a sequential reader, so curiosity may ask about random reads. Is there a benefit? Remember, not all new features will benefit every type of workload. Testing must be performed to see the true benefits.

The next test was done with a local COBOL program with 50,000 random read requests at about 20 rows each. This particular application is very dependent upon I/O response in a random application, and therefore you would not expect any elapsed time improvement, but hope to see some CPU improvement.

The test showed that a one row fetch from the 50,000 cursors took about 592 seconds elapsed, 22.5 seconds CPU. When the application was changed to do a twenty row fetch it took about 626 seconds elapsed but achieved a new CPU time of 16.7 seconds. So there was a CPU savings of about 25%. Not bad, but again not the same as the sequential reader.

Implementation and Usage Considerations

To code for implementation of multi-row fetch is fairly straight forward. With a few changes to existing programs they can be quickly implemented. Below are some very

simple syntax examples showing the new ROWSET keyword for the DECLARE and the FETCH statements.

```
DECLARE CUR1 CURSOR  
WITH ROWSET POSITIONING  
FOR SELECT COL1, COL2 FROM TABLE1;
```

```
OPEN CUR1;
```

```
FETCH NEXT ROWSET FROM CUR1  
FOR :hv ROWS INTO :values1, :values2;
```

There are a few caveats to remember when coding these cursors. Remember that once you start working with a rowset you should remain working with rowsets, otherwise things can get confusing. In other words do not mix multi-fetches with single row fetches in the same cursor. Take the below example to illustrate the point.

This example shows the results of doing a mix of multi-row (rowset) fetching, single rowset fetching, and single fetch.

This first fetch is for a rowset of 10.

```
FETCH COUNT IS 0000000010  
IDS FETCHED:  
CUSID IS 00000000262 ACCT IS 00000002537  
CUSID IS 00000000262 ACCT IS 00000002538  
CUSID IS 00000000281 ACCT IS 00000002672  
CUSID IS 00000000281 ACCT IS 00000002673  
CUSID IS 00000000372 ACCT IS 00000003503  
CUSID IS 00000000372 ACCT IS 00000003504  
CUSID IS 00000000372 ACCT IS 00000003505  
CUSID IS 00000000372 ACCT IS 00000003506  
CUSID IS 00000000372 ACCT IS 00000003507  
CUSID IS 00000000372 ACCT IS 00000003508
```

This next fetch will do a rowset fetch, but for only one row. It fetched a row from the next available rowset.

```
FETCH NEXT ROWSET FOR 1 ROW  
CUSID IS 00000000372 ACCT IS 00000003509
```

This next fetch will fetch another rowset of 10. Everything still looks good...

```

FETCH COUNT IS 0000000010
IDS FETCHED:
CUSID IS 00000000372  ACCT IS 00000003510
CUSID IS 00000000372  ACCT IS 00000003511
CUSID IS 00000000378  ACCT IS 00000003576
CUSID IS 00000000388  ACCT IS 00000003709
CUSID IS 00000000623  ACCT IS 00000006085
CUSID IS 00000000623  ACCT IS 00000006086
CUSID IS 00000000623  ACCT IS 00000006087
CUSID IS 00000000623  ACCT IS 00000006088
CUSID IS 00000000623  ACCT IS 00000006088
CUSID IS 00000000623  ACCT IS 00000006089
CUSID IS 00000000746  ACCT IS 00000007421

```

However, now the application decides to do a traditional single row fetch (instead of a fetching a rowset of 1 as shown above). Notice what happens. The fetch grabbed the second row of the previous rowset due to the fact that it was positioned on the first row of the previously fetched rowset. Is that what the application really wanted or did they want the next available row???

```

FETCH 1 ROW NORMAL CURSOR
CUSID IS 00000000372  ACCT IS 00000003511 ←

```

Even more interesting is what happens when another rowset fetch of 10 is performed. Since the cursor was positioned on ACCT 3511 in the previous rowset it now fetches the next rowset from that position.

```

FETCH COUNT IS 0000000010
IDS FETCHED:
CUSID IS 00000000378  ACCT IS 00000003576 ←
CUSID IS 00000000388  ACCT IS 00000003709
CUSID IS 00000000623  ACCT IS 00000006085
CUSID IS 00000000623  ACCT IS 00000006086
CUSID IS 00000000623  ACCT IS 00000006087
CUSID IS 00000000623  ACCT IS 00000006088
CUSID IS 00000000623  ACCT IS 00000006089
CUSID IS 00000000746  ACCT IS 00000007421
CUSID IS 00000000746  ACCT IS 00000007422
CUSID IS 00000000746  ACCT IS 00000007423

```

So now think about some of the implications when mixing fetch types. And what if you are using dynamic scrollable cursors and moving back and forth using a mixture of multi-row and single row fetches. Where are you? It is certainly not recommended to mix multi-row fetching with single row fetching in the same cursor.

Positioned updates and delete are possible with multi-row fetching. However, again remember that you are working with a rowset. If you do a positioned delete without specifying the row number that you want to delete, you will delete the rowset.

One other thing to remember when using multi-row cursors is to watch out for locks. This is because every row being processed by a multi-row fetch must be locked. The cursor is

position on all rows in current rowset (the rows fetched by a multi-row fetch operation). These locks will depend on isolation level and whether or not a result table is materialized.

Conclusion

Multi-row fetching is definitely a performance benefit in many situations. Testing should be done to realize where the maximum benefits can be achieved because depending on the type of workload results can vary. Having multi-row fetching occur of distributed applications using block fetch is wonderful and may result in immediate benefits when moving from V7 to V8. But with all features there are proper usage techniques to employ. Be sure to read and practice implementation procedures and follow the recommended usage guidelines.

All testing done for this article was performing by Dan Luksetich of YL&A. If you want to hear more about performance and usage of multi-row fetching catch Dan and Susan on the February 2007 YL&A podcast at www.db2expert.com.