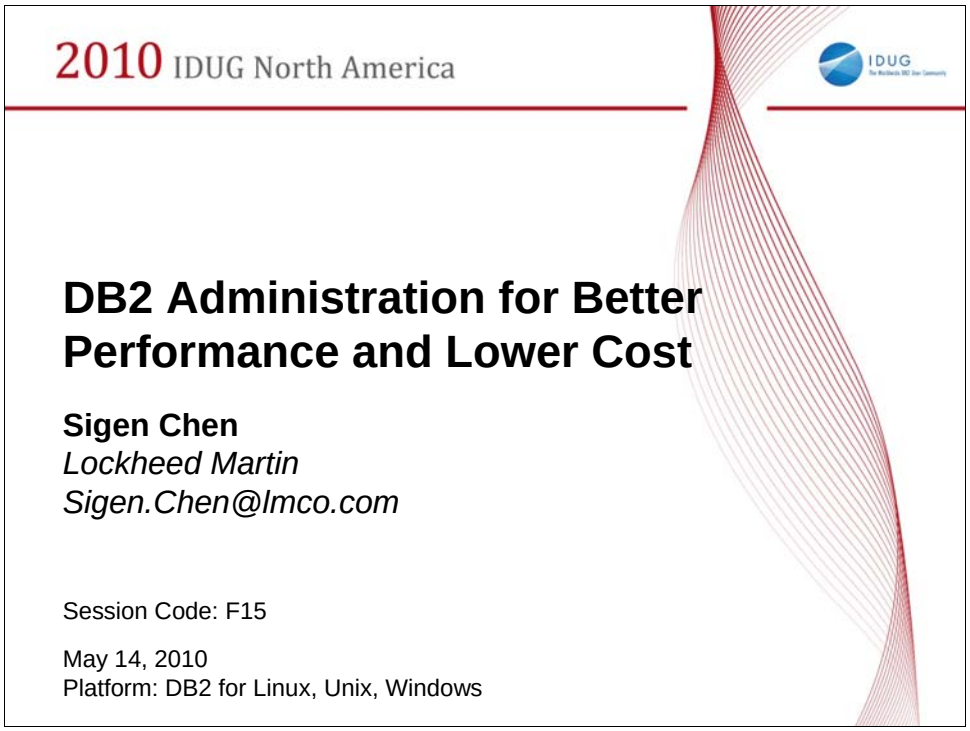


Presentation Abstract

This presentation will cover some practical aspects of improving DB2 application performance while keeping the cost low. The focus will be on exploring DB2 deep compression feature; choosing proper programming language and APIs for a given job; optimizing query and index; tuning key configuration parameters.

Better Performance and Lower cost means - given the system resources, no additional HW/SW resources were added, in fact we have saved Disk space by employing data compression, saved run time via compression, API, indexing, tuning.

Abstract

This presentation will cover some practice to improve DB2 application performance while keeping the cost low. The focus will be on exploring the DB2 Deep Compression feature, choosing proper language and APIs for a given job, optimizing queries and indices, tuning key configuration parameters. Sample code and user application benchmarking data (normalized) will be discussed.

Outline

- This presentation shows how we have achieved excellent DB2 application performance under low cost.
- DB2 Deep Compression feature and its impact on space saving, performance, and resource usage is tested.
- Choosing the right programming language and APIs for a given job is essential.
- Writing proper query and creating the correct index is the key.
- Performance improvement is from double to 40+ fold.
- Some benchmarking test data (normalized) will be used to illustrate the performance impact.

Better Performance and Lower cost means - given the system resources, no additional HW/SW resources were added, in fact we have saved Disk space by employing data compression, saved run time via compression, API, indexing, tuning.

Compression – saved space (>50%), eased memory contention, improved query performance, but drove CPU busier

Language and API may be an architecture issue, but DBAs can contribute to the decision making

Query and Index is always in the heart of the performance elimination process

Test, test, and test based on application specific workloads

Deep Compression - Approach

- 'Static Dictionary' – repeating patterns are replaced by symbols.
- The compression dictionary is physically stored within the data table
- Data exists in compressed format on disk and in memory (bufferpool), and is uncompressed when data is examined
- User data is compressed in the logs
- There is a CPU cost associated with compression, it works better in a non-CPU bound system

Since V9, DB2 LUW supports deep compression feature. Which made space saving, and in many cases performance are improved as well. We will share some performance data collected in our warehouse database system.

Notice that V9.7 started to support index compression and LOB data compression. Test data presented here only dealt with user table compression. Index and LOB compression will be tested later.

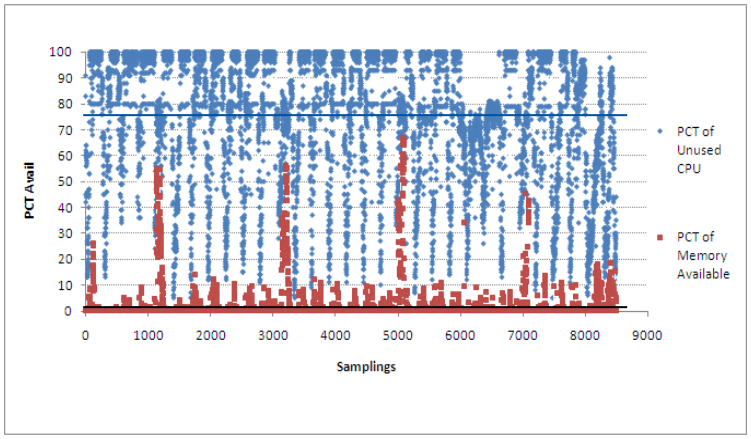
Deep Compression – Pros & Cons

- Reduced storage requirements - more data can be stored in a given disk space
- Performance:
  - Decreased disk I/O traffic
  - Improved bufferpool hit ratio (more data per bufferpool page), reduced memory contention in my warehouse DB case
  - Smaller log size (particularly helpful for HADR)
- Maintenance: Smaller backup images, shorter time to backup, reorg etc.
- Can increase CPU consumption (Do I have CPU to spare?)

Compression is a great feature, whether activate or not, depends on your system resources, particularly available CPU cycle.

Does DB server have CPU to spare – see graph on next slide (Fig. 1)

Fig. 1 CPU and Memory Usage of DB Server (4 CPU, 6 GB Memory, One Month Observation)



Samples were taken every 5 minutes daily, observation time period was a month before the compression.

In the case of our Warehouse database server, there is still available CPU, but not memory – system experienced I/O wait, memory overflow, paging etc..

AVG Available CPU 76%

AVG Available Memory 1.8%

From the historic resources usage data, evidently that this server has the memory constraint, but still has plenty CPU cycles.

Deep Compression – Will it help me?

- **INSPECT ROWCOMPESTIMATE TABLE NAME <tablename> SCHEMA <schema name> RESULTS table.inspect.out**
- **db2inspf table.inspect.out table.inspect.out.fmt**
- **cat table.inspect.out.fmt | grep "Percentage of pages saved"**

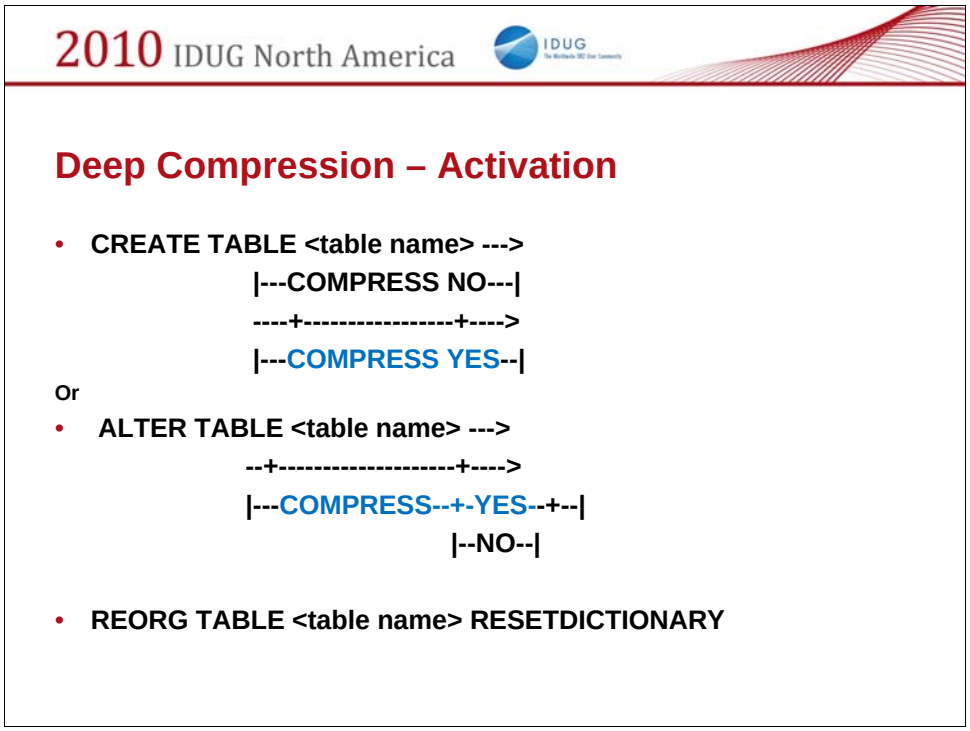
Percentage of pages saved from compression: 68

DATABASE: WAREHOUS
VERSION : SQL09054
2009-11-05-14.06.40.331892

Action: ROWCOMPESTIMATE TABLE
Schema name: DB2INST1
Table name: NMEMH
Tablespace ID: 2 Object ID: 5
Result file name: table.inspect.out

Table phase start (ID Signed: 5, Unsigned: 5; Tablespace ID: 2) : DB2INST1.NMEMH

Data phase start. Object: 5 Tablespace: 2
Row compression estimate results:
Percentage of pages saved from compression: 68
Percentage of bytes saved from compression: 68
Compression dictionary size: 39808 bytes.
Expansion dictionary size: 32768 bytes.
Data phase end.



DB2 V9.5

Optimal compression dictionaries and hence compression ratios, are achieved when the compression dictionary is built from an inclusive sample set. Table reorg (and inspect) builds a dictionary based on all the table data and thus produces the most optimal dictionary.

The effectiveness of using small data subsets to build a compression dictionary is well demonstrated. This is the motivation behind Automatic Dictionary Creation (no table reorg) as part of table growth.

Deep Compression – Verification 1

- **Admin view**

```
select substr(TABNAME,1,10) as tbname,
    COMPRESS_ATTR,DICT_BUILDER,
    COMPRESS_DICT_SIZE,ROWS_SAMPLED,
    BYTES_SAVED_PERCENT
 from SYSIBMADM.ADMINTABCOMPRESSINFO
 where tabname like 'NMEMH%'
```

TBNAME	COMPRESS_ATTR	DICT_BUILDER	COMPRESS_DICT_SIZE	ROWS_SAMPLED	SAVED_PERCENT
NMEMH	Y	REORG	39808	2000000	68
NMEMH2	Y	TABLE GROWTH	40576	1472	70

Slide 8 to 14 present data from Pure Tests of controlled environment, with two tables that have 2-million rows

Runstats to update statistics after reorg, so you would see the compress information

Deep Compression – Verification 2

- **db2diag.log**

FUNCTION: DB2 UDB, data management, sqldReorgDictionaryDriver, probe:160
MESSAGE : ADM5592I A compression dictionary was built and inserted into object "5" in tablespace "2" via "REORG" processing.

FUNCTION: DB2 UDB, data management, sqldOnlineInsertDictionary, probe:4009
MESSAGE : ADM5592I A compression dictionary was built and inserted into object "4" in tablespace "2" via "TABLE GROWTH" processing.

Slide 8 to 14 present data from Pure Tests of controlled environment, with two tables that have 2-million rows

Deep Compression – Verification 3

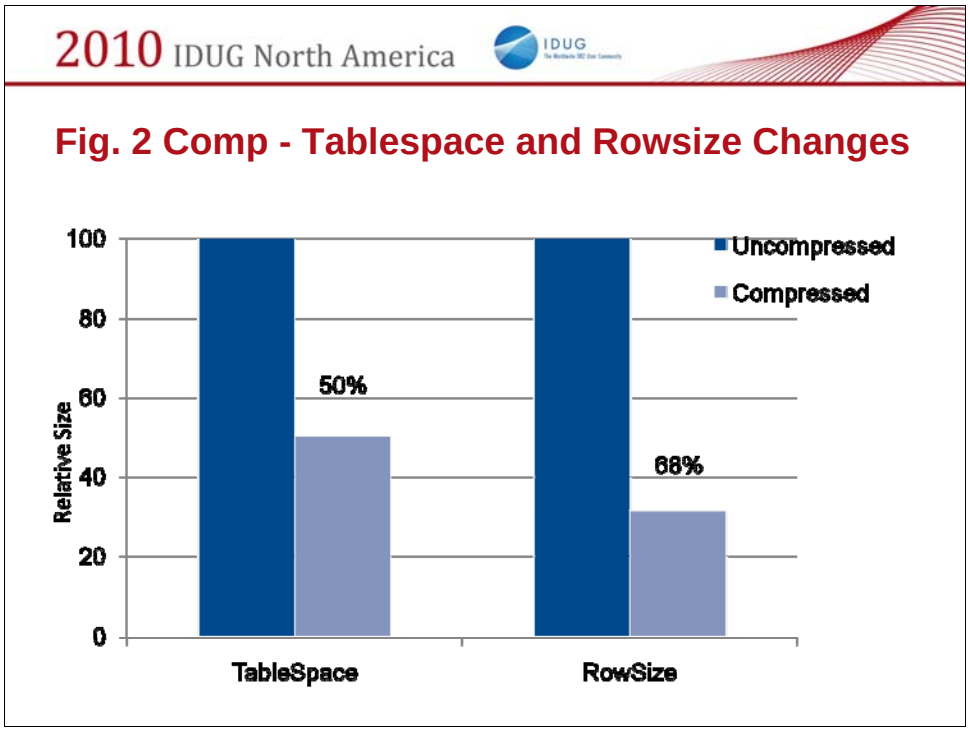
- **Tablespace Used**

$ List tablespaces show detail | grep –i “Used pages”

Used pages = 1,000,288 (before compression)

Used pages = 368,000 (after compressed)

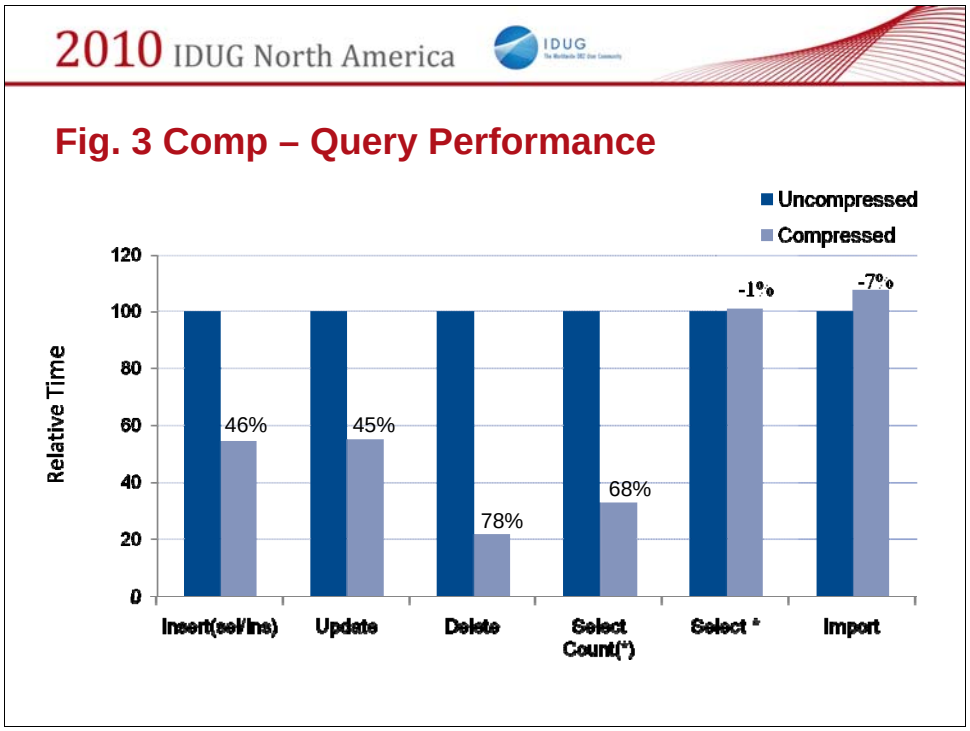
Slide 8 to 14 present data from Pure Tests of controlled environment, with 2 tables that have 2-million rows



Slide 8 to 14 present data from Pure Tests of controlled environment, with 2 tables that have 2-million rows

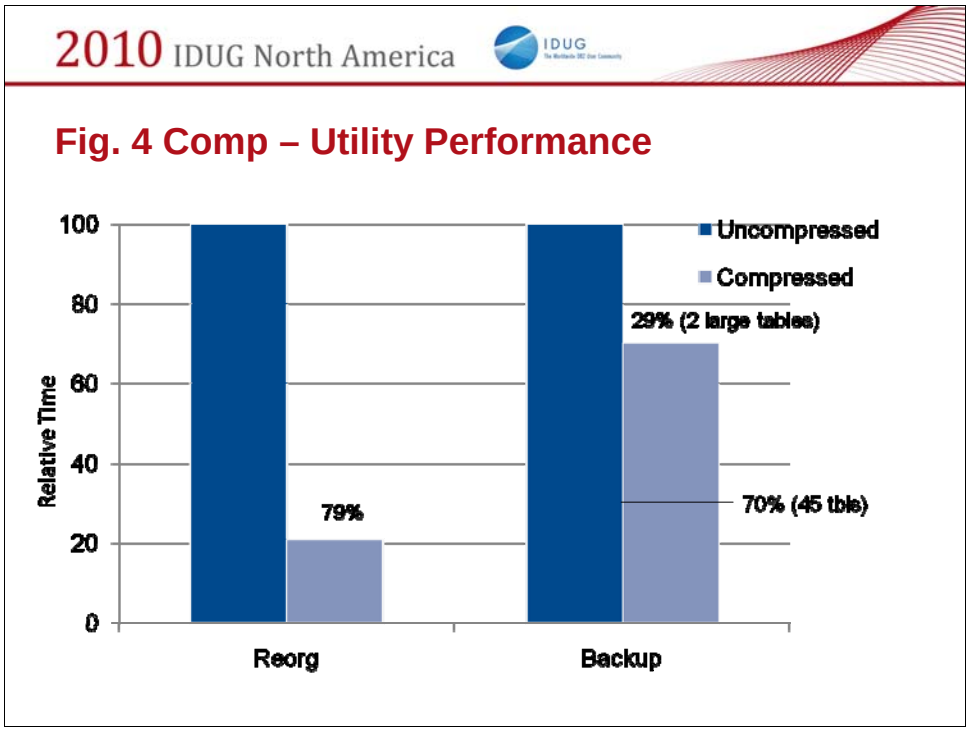
Tablespace reduction: 50%

Rowsize reduction: 68%



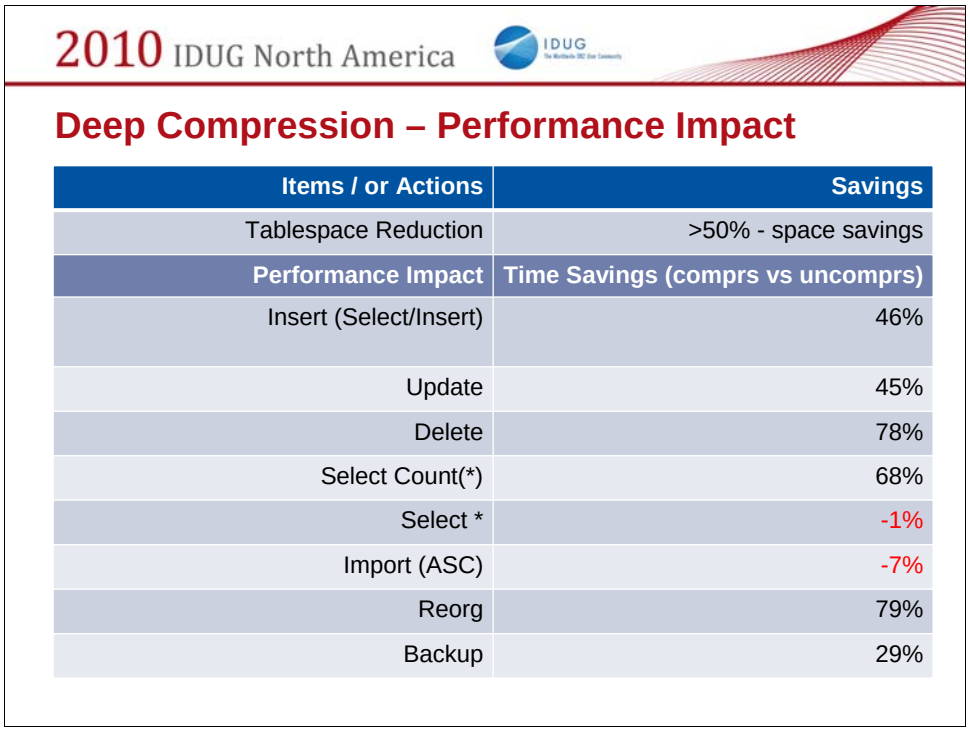
Slide 8 to 14 present data from Pure Tests of controlled environment, with two tables that have 2-million rows

SQL	% of Time Saving
Insert (select/insert):	46%
Update:	45%
Delete:	78%
Select count(*):	68%
Select *:	-1%
Import:	-7%



Slide 8 to 14 present data from Pure Tests of controlled environment, with 2 tables that have 2-million rows

Maint Job	% Time Savings:
---	---
Reorg:	79%
Backup:	29% (two big tables)
	70% (40 large tables)



Slide 11 to 14 present data from Pure Tests of controlled environment, with 2 tables that have 2-million rows

Performance Impact on A Warehouse DB Server

- 45 tables of real warehouse production system were compressed, each with more than 1,000,000 records
- Tablespace saving is 66.5% (>half TB)
- Composite workloads (mostly selects and select inserts of a reporting app) is about 38.7% faster on average
- Backup time changed from 8h 6m to 2h 41m (67%)
- Reorg time changed from 13h 58m to 8h 58m (35%)
- How did the compression affect CPU, Memory usage?

45 tables (records range from 1 M to 50M rows) were compressed

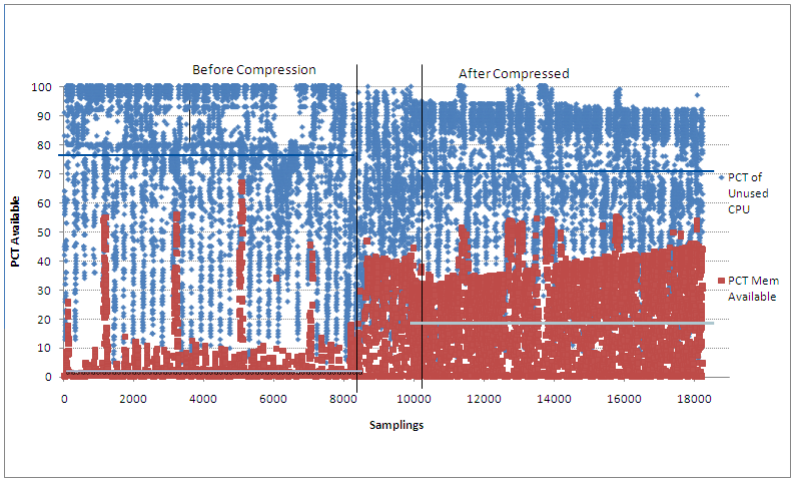
Tablespace saving is more profound (50% -> 66.5%)

Reporting workload (composite SQLs) is 38.7% faster

Backup time changed from 8 hours to 2.5 hours

2010 IDUG North America

Fig .5 Overall System Resources Impact (CPU and memory usage) by Activating Compression



Samples were taken every 5 minutes, one month before compression, 2 weeks compression, a month after compression.

When 40+ tables (records range from 1 M to 50M rows) of a warehouse database were compressed

	AVG Available CPU (%)	AVG Available Memory (%)	Conclusion
Before Comp	76.5	1.86	Had CPU, no memory
During the conversion	60.5	10.7	Made CPU busier, freed some memory
After Comp*	70.8	19.8	CPU is about 6% busier, ~18% more memory freed

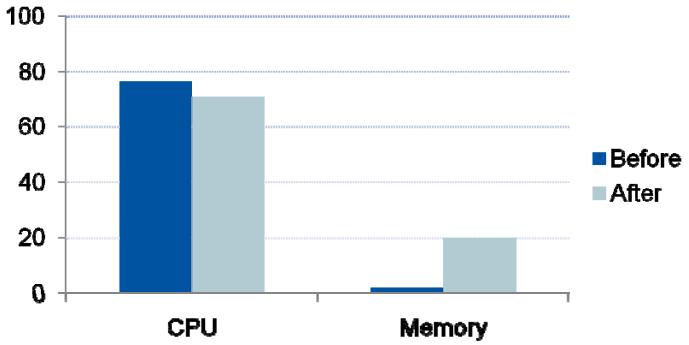
CPU and Memory Dynamic Changes

Compression	Available CPU (%)		Available Memory (%)	
	AVG	STDEV	AVG	STDEV
Before	76.50	25.14	1.81	6.18
During	60.53	23.83	10.72	14.85
After	70.75	23.10	19.83	17.07

Further explain the last graph

CPU usage is about 6% busier, about 18% memory is freed by compression. This better served our warehouse database server.

Average CPU and Memory Availability Change Before and After The Compression



Further explain the last graph

CPU usage is about 6% busier, about 18% memory is freed by compression. This better served our warehouse database server.

Outline

- This presentation shows how we have achieved excellent DB2 application performance under low cost.
- DB2 Deep Compression feature and its impact on space saving, performance, and resource usage is tested.
- ***Choosing the right programming language and APIs for a given job is essential.***
- Writing proper query and creating the correct index is the key.
- Performance improvement is from double to 40+ fold.
- Some benchmarking test data (normalized) will be used to illustrate the performance impact.

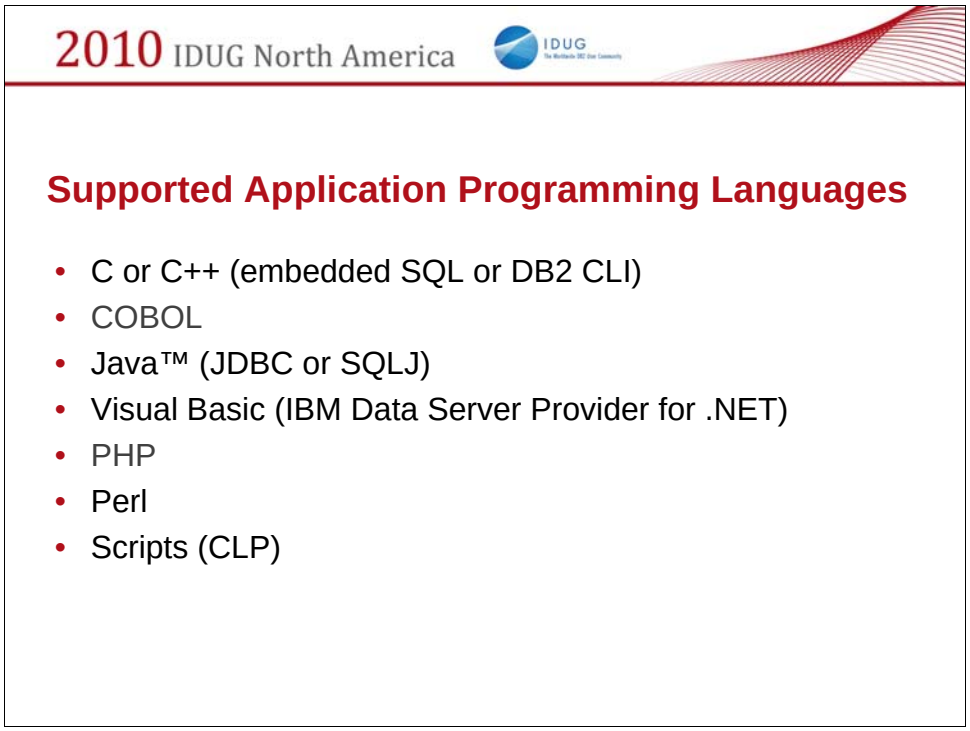
Better Performance and Lower cost means - given the system resources, no additional HW/SW resources were added, in fact we have saved Disk space by employing data compression, saved run time via compression, API, indexing, tuning.

Compression – saved space (>50%), eased memory contention, improved query performance, but drove CPU busier

Language and API may be an architecture issue, but DBAs can contribute to the decision making

Query and Index is always in the heart of the performance elimination process

Test, test, and test based on application specific workloads



DB2 Supported API:

C or C++ (embedded SQL or DB2 CLI)
COBOL
Java (JDBC or SQLJ)
Visual Basic (IBM Data Server Provider for .NET)
PHP
Perl

Choose the Right API for a Given Job

- Embedded (1)
- CLI (1.03)
- ADO/IBM Provider (1.31)
- ADO/MS Bridge (1.47)
- JDBC T2 (1.56)
- Perl (3.78)
- Script (CLP) (> 4.80)
  (Script meant to be interactive, small, quick, maint etc.)

Numbers in the braces are the relative time, smaller the better. (mostly select stmts in test cases)

I tested all, but COBOL and PHP, because I did not know those two languages

JDBC T2 and T4 comparison data depends on which java to use

When /usr/bin/java (generic) is used, T2 performed better than T4; but when ~/sqllib/java/jdb64/bin/java is used, T4 is better than T2 ?!

T2 vs T4

T2/OSJ	T4/OSJ	T2/IBMJ	T4/IBMJ
0.91	1.38	1.91	1.57

OSJ=/usr/bin/java

IBMJ=~/sqllib/java/jdk64/bin/java

number is execution time, smaller the better

$ /usr/bin/java -version

java version "1.5.0"

Java(TM) 2 Runtime Environment, Standard Edition (build pxz64devifx-20090327 (SR9-SSU))

IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux s390x-64 j9vmxz6423-20081129 (JIT enabled)

J9VM - 20081126_26240_BHdSMr

JIT - 20081112_1511ifx1_r8

GC - 200811_07)

JCL - 20090327

$ ~/sqllib/java/jdk64/bin/java -version

java version "1.5.0"

Java(TM) 2 Runtime Environment, Standard Edition (build pxz64dev-20070511(SR5))

IBM J9 VM (build 2.3, J2RE 1.5.0 IBM J9 2.3 Linux s390x-64 j9vmxz6423-20070426 (JIT enabled)

J9VM - 20070420_12448_BHdSMr

JIT - 20070419_1806_r8

GC - 200704_19)

JCL - 20070511

Fetch APIs

- SQLBindCol (SQLHSTMT StatementHandle, /* hstmt */
  SQLUSMALLINT ColumnNumber, /* icol */
  SQLSMALLINT TargetType, /* fCType */
  SQLPOINTER TargetValuePtr, /* rgbValue */
  SQLLEN BufferLength, /* dbValueMax */
  SQLLEN *StrLen_or_IndPtr); /* *pcbValue */

- SQLGetData (SQLHSTMT StatementHandle, /* hstmt */
  SQLUSMALLINT ColumnNumber, /* icol */
  SQLSMALLINT TargetType, /* fCType */
  SQLPOINTER TargetValuePtr, /* rgbValue */
  SQLLEN BufferLength, /* cbValueMax */
  SQLLEN *StrLen_or_IndPtr); /* pcbValue */

- SQLFetchScroll (SQLHSTMT StatementHandle,
  SQLSMALLINT FetchOrientation,
  SQLLEN FetchOffset);

Numbers in the braces are the relative time

Above test result generated by using proper data type in binding. Notice that proper data type is better than using SQL_C_CHAR

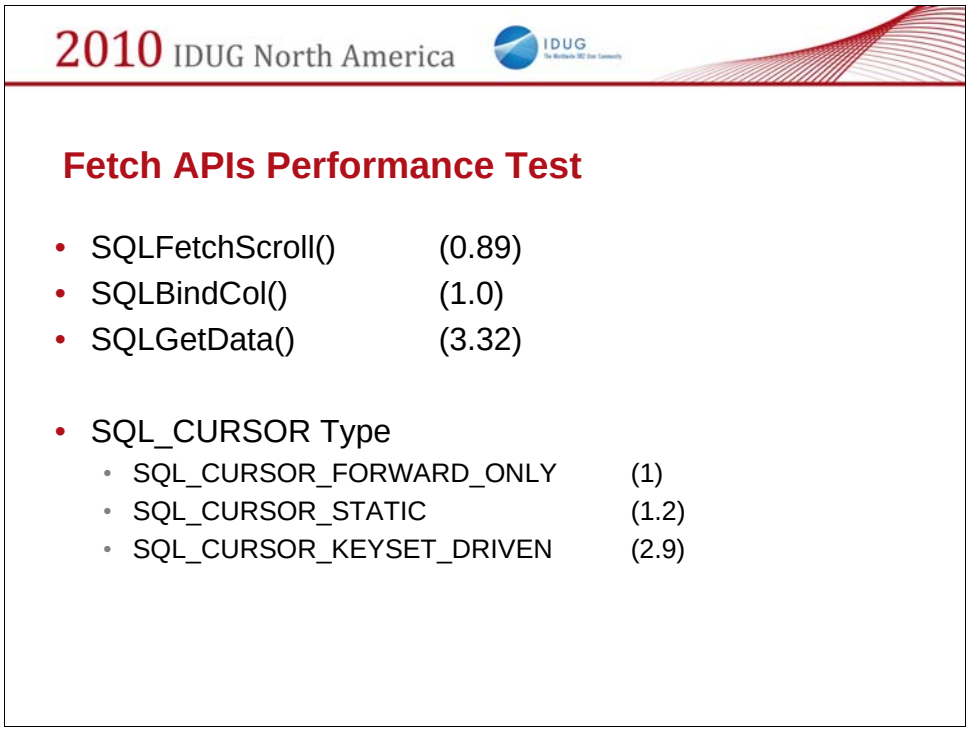
SQLBindCol() is used to associate (bind) columns in a result set to either:

Application variables or arrays of application variables (storage buffers), for all C data types. Data is transferred from the DBMS to the application when SQLFetch() or SQLFetchScroll() is called. Data conversion might occur as the data is transferred.

A LOB locator, for LOB columns. A LOB locator, not the data itself, is transferred from the DBMS to the application when SQLFetch() is called. Alternatively, LOB columns can be bound directly to a file using SQLBindFileToCol().

SQLGetData() retrieves data for a single column in the current row of the result set. This is an alternative to SQLBindCol(), which is used to transfer data directly into application variables or LOB locators on each SQLFetch() or SQLFetchScroll() call. An application can either bind LOBs with SQLBindCol() or use SQLGetData() to retrieve LOBs, but both methods cannot be used together.

SQLGetData() can also be used to retrieve large data values in pieces.



Numbers in the braces are the relative time

For fetching data, 10 cols x 200,000 rows in our test case, if the time for using typical SQLBindCol() is normalized to 1.00, the performance sequence from the fastest to the slowest is:

Above test result generated by using proper data type in binding. Notice that proper data type is better than using SQL_C_CHAR

Array Fetch

In the above test data, the fetch was sequential, i.e., retrieving rows starting with the first row, and ending with the last row. In that case, we know SQLFetchScroll() gives the best performance. What if an application to allow the user to scroll through a set of data both forwards and backwards? DB2 CLI has three types of the scrollable cursors –

(1) forward only (default) cursor - can only scrolls forward.

(2) static read-only cursor - is static, once it is created no rows will be added or removed, and no value in any rows will change

(3) keyset-driven cursor - has ability to detect changes to the underlying data, and the ability to use the cursor to make changes to the underlying data. Keyset-driven cursor will reflect the changed values in existing rows, and deleted rows; but it will not reflect added rows. Because the set of rows is determined once, when the cursor is opened. It does not re-issue the select statement to see if new rows have been added that should be included.

Insert APIs

- CLI USE_LOAD (0.36)
- Array_Insert (0.42)
- NotLoggedInitially (0.81)
- SQLBindParameter (1.0)

Numbers in the braces are the relative time

Above test result generated by using proper data type in binding. Notice that proper data type is better than using SQL_C_CHAR

Insert APIs

```
……
rc = SQLBindParameter(hstmt, 1,
                      SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
                      0, 0,
                      (SQLINTEGER *)col1, sizeof((SQLINTEGER *)col1 ),
                      &lvalue );
rc = SQLBindParameter(hstmt, 2,
                      SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                      100, 0,
                      (SQLCHAR *)col2, 100, NULL );

/* execute the statement, assume that n (100,000) rows to be inserted */
while(pass++<n){
    rc=SQLExecute( hstmt );
    *col1 = *col1+1;
}
```

Above test result generated by using proper data type in binding. Notice that proper data type is better than using SQL_C_CHAR

Typical Row Insert

```
……
rc = SQLBindParameter(hstmt, 1,
                      SQL_PARAM_INPUT, SQL_C_LONG, SQL_INTEGER,
                      0, 0,
                      (SQLINTEGER *)col1, sizeof((SQLINTEGER *)col1 ),
                      &lvalue );
rc = SQLBindParameter(hstmt, 2,
                      SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                      100, 0,
                      (SQLCHAR *)col2, 100, NULL );

/* execute the statement, assume that n (100,000) rows to be inserted */
while(pass++<n){
                rc=SQLExecute( hstmt );
                *col1 = *col1+1;
}
……
```

Example of Array Insert

```
SQLINTEGER col1[]= {1,2,3,4,5,6,7,8,9,10, ……100};
SQLCHAR col2[100][100]=
   {"A1","B2”,”C3”,”D4”,”E5”,”F6”,”G7”,”H8”,”I9”,”J10”,……”z100”};

rc=SQLSetStmtAttr(hstmt,
                  SQL_ATTR_PARAMSET_SIZE,
                  (SQLPOINTER)100, 0);
rc = SQLBindParameter(hstmt, 1,
                  SQL_PARAM_INPUT, SQL_C_LONG,
   SQL_INTEGER,
                  0, 0,  col1, 0, NULL);
rc = SQLBindParameter(hstmt, 2,
                  SQL_PARAM_INPUT, SQL_C_CHAR, SQL_CHAR,
                  100, 0,  col2, 100, NULL );
while(pass++<n)
   rc=SQLExecute( hstmt );
```

Array could be array of rows, or array of columns. Example showed at this slide is array of columns.

Query Feature Examples

- **Dynamic Query Using Param Marker (?) to Reuse the Stmt, save SQLPrepare() calls**
  - 100,000 records updates yielded 302x performance improvement compared to static value binding
  - Performance gain depends on the resultSet
- **Blocking results using 'optimize for N rows'**
  - Fetching 100,000 rows with1000 blocking size resulted 1.9x ~ 5.6x performance improvement depends on the API

Example of using parameter markers:

```
char * sqlstmt = (SQLCHAR *) "insert into table1 (col1,col2….) values (?,?....)";
SQLPrepare(hstmt,sqlstmt, SQL_NTS);
SQLBindParameter(hstmt,1, SQL_PARA_INPUT, SQL_C_LONG, SQL_INTEGER, 0, 0, (SQLINTEGER *) col1 sizeof((SQLINTEGER *)col1), &lvalue);
…..
While(true){
SQLExecute(hstmt);
//do something to change the condition
}
```

Example of using blocking:

```
Select col1, col2 from table1 where col1 in (x,y) OPTIMIZE for 1000 rows;
```

Ensure that you resultset may have more than N rows.

Outline

- This presentation shows how we have achieved excellent DB2 application performance under low cost.
- DB2 Deep Compression feature and its impact on space saving, performance, and resource usage is tested.
- Choosing the right programming language and APIs for a given job is essential.
- ***Writing proper query and creating the correct index is the key.***
- Performance improvement is from double to 40+ fold.
- Some benchmarking test data (normalized) will be used to illustrate the performance impact.

Better Performance and Lower cost means - given the system resources, no additional HW/SW resources were added, in fact we have saved Disk space by employing data compression, saved run time via compression, API, indexing, tuning.

Compression – saved space (>50%), eased memory contention, improved query performance, but drove CPU busier

Language and API may be an architecture issue, but DBAs can contribute to the decision making

Query and Index is always in the heart of the performance elimination process

Test, test, and test based on application specific workloads

Create Necessary Indices

- Snapshot and event monitoring data to capture bottleneck queries
- db2exfmt to check the access plan
- db2advis suggests indices / or create your own
- Always test the new indices using application specific test cases
- Unnecessary indices can hurt
- Performance improvement in our cases range from double to 40+ times

Cumulative Time Savings

Effect of APIs, Query Enhancement, Indexing, Tuning

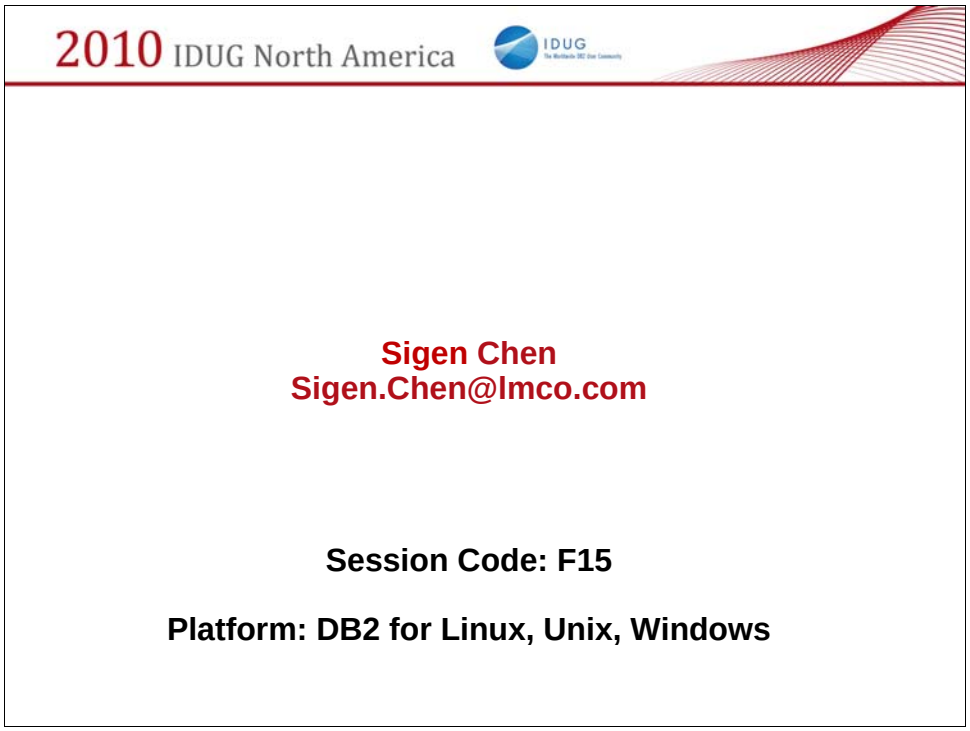
- ✓ Initial cost (yearly): 584 Units (of cost)
- ✓ Optimization cost: 40 Units
  - ✓ Including coding, indexing, proactive maintenance
- ✓ Final (Subsequent Yearly) cost: 1 Unit

Deep Compression is not part of the above effort

Caution:

Time saving may vary depends on the applications and system resources. But one may always find bottlenecks and conduct the costs/savings analysis by eliminating the bottlenecks with cost effective measures.

In our case, API improvement, Query enhancement, Indexing, Tuning did not require much additional resource. If there is free CPU cycle to spare, deep compression would not cost much either.



Sigen Chen is a DB2 UDB DBA, currently works for Lockheed Martin Information Technologies as a Senior Information System Analyst. Prior to join Lockheed Martine, he worked as a Software Developer at DB2 Performance Group, IBM Toronto Lab. Sigen was the speaker at IDUG 2008 North America and IDUG 2008 Europe.