DB2 for z/OS Performance

Road show edition
April / May 2008

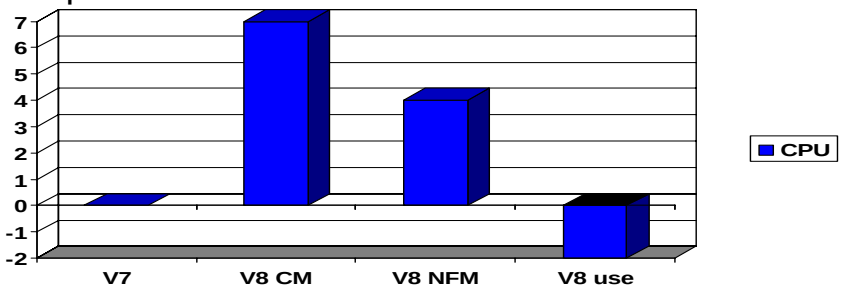
- **Abstract: DB2 for z/OS Performance:**

- **This session covers application performance topics for DB2 for z/OS V8 & 9 including:**
  - **Query performance enhancements such as materialized query table and non-index column distribution statistics**
  - **SQL performance enhancement such as more indexable predicates and multi-row Fetch, Update, Delete, Insert**
  - **Index enhancement such as variable length index keys**
  - **Other application performance enhancement such as trigger and lock avoidance**
- **This presentation provides information on DB2 for z/OS V8 & 9 performance. Please note that some product changes may result in changes.**

Acknowledgment and Disclaimer

- **Measurement data included in this presentation are obtained by the members of the DB2 performance department at the IBM Silicon Valley Laboratory. Akira Shibamiya is the primary source.**
- **The materials in this presentation are subject to**
  - **enhancements at some future date,**
  - **a new release of DB2, or**
  - **a Programming Temporary Fix**
- **The information contained in this presentation has not been submitted to any formal IBM review and is distributed on an "As Is" basis without any warranty either expressed or implied. The use of this information is a customer responsibility.**

V8 best practice performance plan example scenario

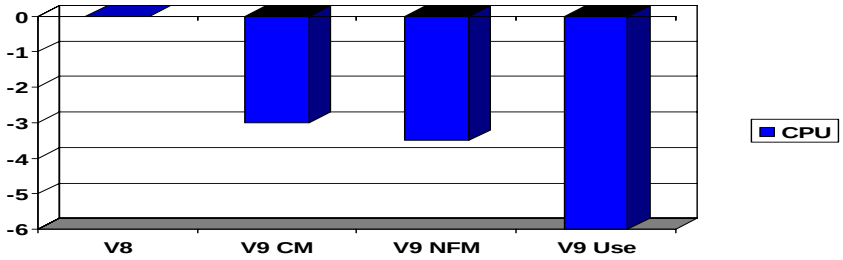


Data sharing Better statistics
REBIND
PGFIX(YES)
zIIP
zparms

DB design adjustments
Cluster, index
application changes
multirow fetch & insert
SQL adjustments

Your situation will vary. Less CPU is better.

DB2 9 z10, z9, z890 & z990 performance plan example scenario



Utilities	DB design adjustments
Histogram statistics	Index improvements
REBIND	application changes
DSNZPARMS	native SQL procedures
	SQL adjustments

Your situation will vary. Less CPU is better.
z800 and z900 expect +5% to +10% CPU

DB2 V8 for z/OS Performance Overview

- Performance / Scalability Enhancements
  - Improved partitioning scale and flexibility
  - Many index improvements
- Query / Access Path Performance Enhancements
- Multirow fetch and insert
- Synergy with new hardware: zIIP, MIDAW, DS8000, …

V8 Queries and data warehouses

- Optimization Improvements
  - Improved techniques
  - Enhanced data
  - Visual Explain
- Enhanced index options
- Materialized Query Tables
- New Partitioning options
- QMF improvements
- SQL enhancements

DB2 9 for z/OS Performance Overview

- Significant CPU time reduction in most utilities
- Synergy with new hardware: zIIP, MIDAW, DS8000, …
- Performance / Scalability Enhancements
  - Especially Insert, Update & Delete
- Query / Access Path Performance Enhancements
- Other Performance Enhancements
  - Native SQL procedure, index compression
  - LOBs, Varchar
- Improved virtual storage usage below 2GB DBM1

DB2 9 Query Enhancements

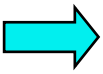
- SQL enhancements: INTERSECT, EXCEPT, cultural sort, caseless comparisons, FETCH FIRST in fullselect, OLAP specifications: RANK, ROW_NUMBER, …
- pureXML integration and text improvements
- Index improvements
  - Index on expression Larger index pages
  - Index compression Improved page split
- Improved Optimization statistics: Histogram
- Optimization techniques & REOPT(AUTO)
  - Cross query block optimization
  - Generalize sparse index & in-memory data cache method
  - Dynamic Index ANDing for Star Schema
- Analysis: instrumentation & Optimization Service Center

DB2 9 Scalability

- Insert performance APPEND INDEX LOG
  - INDEX on expression, 8K, 16K, 32K, split
  - Randomized index key, larger preformat
  - Log Latch contention & spin relief, archiving
  - Not logged table space
- Partitioned table with segmented space
- Memory improvements 64 bit address space

V8 Performance Highlights

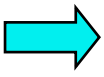
- 10 to 1000 times improvement possible from
  - Materialized Query Table
  - Stage 1 and indexable predicate for unlike data types
  - Distribution statistics on non-indexed columns
  - Other access path selection enhancements



Underlined features require rebind

Performance Highlight - continued

- 2 to 5 times improvement possible from
  - Star Join with work file index and in-memory work file
  - Partition Load/Reorg/Rebuild with DPSI
  - DBM1 virtual storage constraint relief
- Up to 2 times (more in distributed environment) improvement possible from
  - Multi-row Fetch, cursor Update, cursor Delete, Insert



Underlined features require rebind

Materialized Query Table

- Pre-selected and/or pre-computed results from large table(s) saved in much smaller MQT for fast subsequent access
  - Example: Avg Income, Height, NetAssetValue, ... of 300 million US residents grouped by 50 states
  - 10 to 1000 times faster possible for some queries
- Automatic query rewrite for dynamic SQL to take advantage of relevant MQT
  - Summary table can be used directly by both static and dynamic SQL

Materialized Query Table - continued

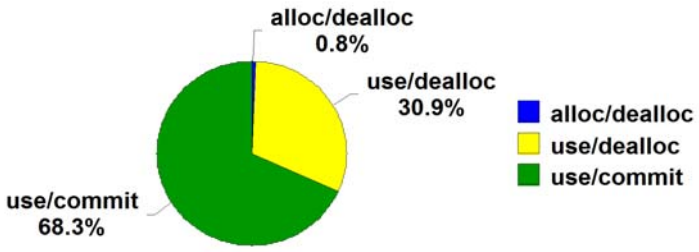
- Performance considerations for maximum use
  - For large MQT,
    - Use segmented tablespace because of almost instantaneous mass delete in REFRESH TABLE
    - Runstats after REFRESH for good access path selection
      - ☞ especially useful in join involving MQT
  - Zparm SPRMMQT for threshold to prevent unnecessary additional bind overhead for short-running SQL

Distribution stats on single and multiple columns

- Top N highest, and/or lowest, frequency of values and cardinality

Bind option

Acquire / release example




```
SELECT FROM A, SYSIBM.SYSPLAN B WHERE  B.ACQUIRE='A'
 AND B.RELEASE='D' ...
```

Better join sequence from more precise filter factor estimation of combined predicates

Distribution statistics ...

- DSTATS (Distribution stats for DB2 for z/OS)
  - A down-loadable tool available prior to V8
  - http://www.ibm.com/support/docview.wss?uid=swg24001598
- Fixes the most typical access path selection problems encountered today
  - Optimizer unable to come up with the best access path because of a lack of distribution stats on non-indexed columns which are referenced in predicates
    - ➔Can cause performance degradation due to access path change in a new release or after access-path-related maintenance

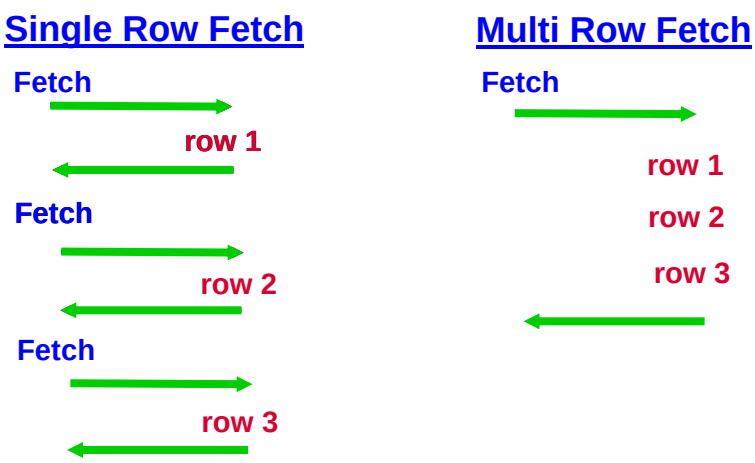
More Indexable Predicates

- For column comp-op value with unlike type or length
  - 4 byte char column = 8 byte host variable
  - Integer column = decimal host variable
  - Stage 2 and non indexable in V7
  - Stage 1 and indexable in V8
    - So index on char or integer column here can be used in V8 but not in V7
  - Also useful where a programming language does not support SQL data types. For example,
    - No decimal type by C/C++, no fixed-length char by Java

NOTES

- Stage 1 and indexable predicate in
  - V6: Column comp-op non column expression such as SELECT FROM A WHERE a1=x+y
    - also char/varchar of different size in equi-join such as SELECT FROM A,B WHERE 10byte char a1=20byte varchar b1
  - V7: Column comp-op column expression in join such as SELECT FROM A,B WHERE a1=b1+x, if table B joined to A
- But generally only if left side column has equal or bigger size and precision
- V8 removed this restriction for both local and join predicates

Multi-row Fetch



Multi-row Fetch - continued

- FETCH NEXT ROWSET FROM cursor FOR N ROWS INTO hva1, hva2, hva3
- Up to 50% CPU time reduction by avoiding API (Application Programming Interface) overhead for each row fetch (100 rows)
  - % improvement lower if more columns and/or fewer rows fetched per call
    - Higher improvement if accounting class 2 on, CICS without OTE, many rows, few columns
  - See later foils for distributed

Multi-row Insert

- INSERT INTO TABLE FOR N ROWS VALUES(:hva1,:hva2,...)
- Up to 40% CPU time reduction by avoiding API overhead for each row insert
  - % improvement lower if more indexes, more columns, and/or fewer rows inserted per call
- Similar improvement for multi-row cursor Update and Delete

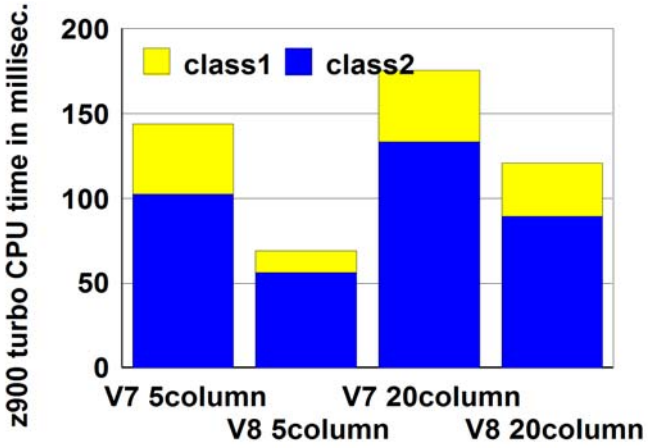
Multi-row in distributed environment

- Fetch, insert, update & delete
- Dramatic reduction in network traffic and response time possible
  - by avoiding message send/receive for each row in
    - Fetch when not [read-only or (CURRENTDATA NO and ambiguous cursor)]
    - Update and/or Delete with cursor
    - Insert
  - Up to 8 times elapsed time reduction observed (up to 4 times CPU time reduction)

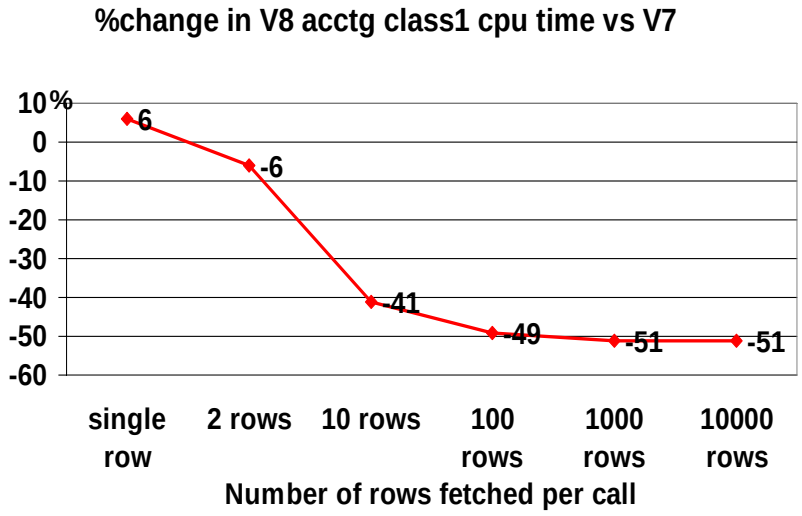
Distributed multi-row ...

- If Fetch with read-only or [CURRENTDATA NO and ambiguous cursor], multi-row Fetch is automatically enabled, resulting in
  - CPU time saving of up to 50%
  - But no significant difference in message traffic compared to V7 with Block Fetch
    - Note that multi-row Fetch is unblocked; i.e. if 10 Fetch calls are issued for 10 rows each, 10 blocks are sent, compared to 1 block if multi-row Fetch is not explicitly used.
    - V7 PQ49458 8/2003
      - OPTIMIZE FOR for access path and network blocking
      - FETCH FIRST for access path but not network blocking when no OPTIMIZE FOR clause

DSNTIAUL fetching 10000 rows with 5 and 20 columns



20 column 100000 row Fetch CPU Time

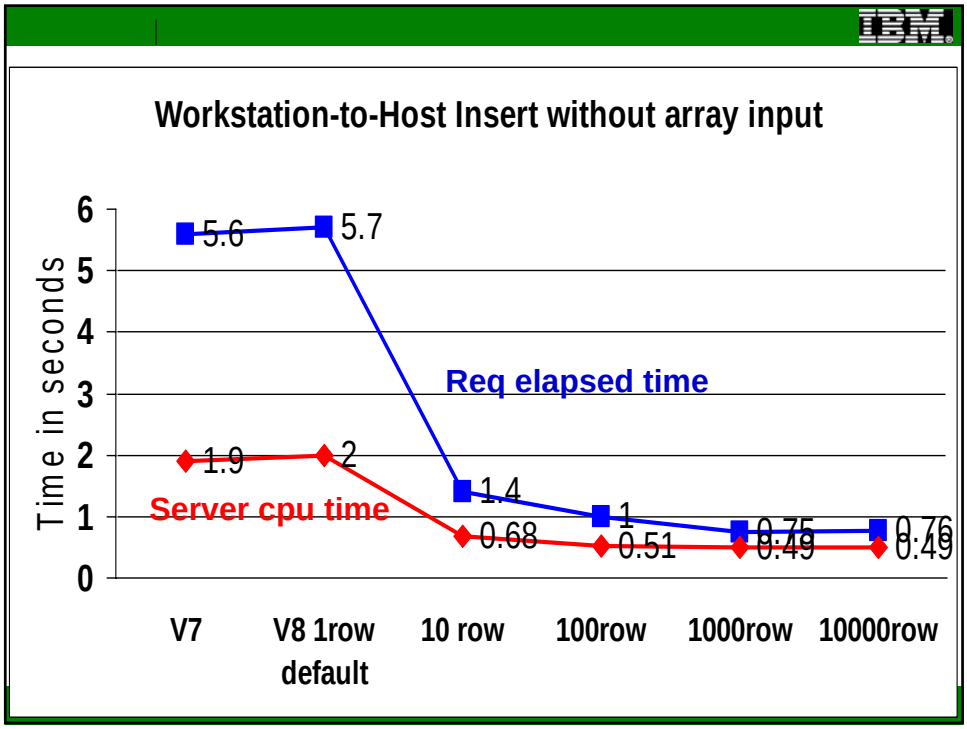


NOTES

- **The graph clearly shows that the percentage improvement goes up as more rows are fetched per Fetch call.**
  - **With 1 row fetch, V8 cpu is 6% higher than V7.**
  - **However, with 2 row fetch, V8 becomes faster by 6%.**
  - **Beyond 100 rows, about 50% improvement continues.**
  - **Similarly for elapsed time and class 2 cpu time.**
- **The measurement shown is for a very simple fetch via tablespace scan fetching 20 columns**
  - **Less %improvement for more complex Fetch involving join, sort, index access, more than 20 column fetch**
  - **More %improvement for less than 20 column fetch**

Multi-row Insert Workstation-to-Host

- **DB2 for z/OS V8 acting as a DRDA application server, accessed from a DB2 Connect Client running on Linux/Unix/Windows as a DRDA application requestor**
- **10000 20-column rows inserted**
- **10row/Insert call**
  - **-76% elapsed time and -63% cpu time compared to V7**
  - **-30% elapsed time and -38% cpu time compared to V7 array input**
- **100row/Insert call**
  - **-82% elapsed time and -63% cpu time compared to V7**
  - **-33%elapsed time and -49% cpu time compared to V7 array input**



Automatic use of multi-row Fetch

- DRDA as discussed previously
- DSNTEP4 = DSNTEP2 with automatic multi-row fetch
  - Up to 35% CPU reduction in fetching 10000 rows with 5 and 20 columns
- DSNTIAUL (sample Unload utility)
  - Up to 50% CPU reduction in fetching 10000 rows with 5 and 20 columns
- QMF with APAR

Elapsed Time Analysis

NOTES

- **Accounting report (not trace) by connection type most useful for initial analysis**
  - **Omegamon DB2 Performance Expert ACCOUNTING REPORT LAYOUT(LONG) ORDER(CONNTYPE) EXCLUDE(PACKAGE(*)) to group by thread connection type such as TSO, CICS, DB2CALL, RRS, IMS, DRDA, etc. for the period of interest.**
  - **Also STATISTICS REPORT LAYOUT(LONG) for the corresponding period extremely desirable**

Accounting Class 1 and 2

AVERAGE	CLASS1	CLASS2
ELAPSED TIME	233ms	19ms
CPU TIME	2.95ms	2.71ms
WAIT TIME		14.76ms
NOT ACCOUNTED TIME		1.31ms

- **For most cases**
  - **Class 1 for application + DB2 time**
  - **Class 2 for DB2 time only**
- **CICS without TS 2.2 or later threadsafe option**
  - **Class 1 CPU for task switch + DB2 time**
  - **Class 2 for DB2 time only**

High NOT ACCOUNTED time – 2 most likely causes

- CPU wait under high cpu utilization, especially with lower dispatching priority

  E.g. goal mode with low priority for DB2 address space compared to DDF enclave, CICS, WebSphere address space, or DDF enclave with SYSOTHER (discretionary)

- Excessive detailed online tracing with vendor tools

Other causes are much less frequent and widely varied

Some events not being captured by DB2, but more events are being captured in newer versions

Details on the web:

http://www.ibm.com/support/docview.wss?rs=64&context=SSEPEK&uid=swg21045823

NOTES

- Other causes are much less frequent and widely varied
  - **Some events not being captured by DB2, but more events are being captured in newer versions**
  - **Online support document: http://www.ibm.com/support/docview.wss?rs=64 &context=SSEPEK&uid=swg21045823**

Accounting Class 3

SUSPENSIONS	TOTAL TIME	#EVENTS
LOCK/LATCH	0.11ms	0.3
SYNC DATABASE I/O	8.73ms	8.86
SYNC LOG WRITE I/O	1.64ms	0.49
OTHER READ I/O	2.64ms	0.76
OTHER WRITE I/O	0.004ms	0.00
SERVICE TASK	1.60ms	0.47
.....		
TOTAL CLASS 3 WAIT	14.76ms	10.88

- **Class 3 acctg strongly recommended:** Negligible overhead except when high internal DB2 latch contention, eg over 10000/sec

NOTES

- Lock/Latch wait = Lock wait + IRLM latch wait + internal DB2 latch wait
  - In the rare case of over 10000 per second, disabling class 3 may significantly bring down class 1 and 2 cpu time.
- Sync I/O wait = wait for read or write i/o by this application agent
  - Avg time = 8.73ms/8.86 = 0.985ms
- Other read I/O wait = wait for read i/o by another application agent or prefetch engine
- Other write I/O wait = wait for write i/o by another application agent or write engine, may include some time waiting for log write-ahead

I/O wait time tuning

- Buffer pool tuning - discussed in Buffer Pool section
- I/O configuration tuning
  - Make sure of sufficient I/O resources
  - Faster device, such as ESS 800 or DS8000 as needed
  - Parallel Access Volume (PAV) beneficial if I/O contention with high IOSQ time in RMF
  - I/O striping

NOTES: agenda

- Minimizing #SQL calls, columns, host variables, predicates evaluated, SQL statements, rows searched
- OPT for N ROWS
- Existence check
- Dynamic SQL, JDBC/SQLJ
- Bind option acquire and release
- Thread reuse
- DB2 trace
- Distributed / stored procedure
- Catalog statistics check
- Compression, Encryption, Row-level Security

Minimize SQL Calls to Reduce API Overhead

- Filter out unnecessary rows by adding predicates rather than by application program checking
- Use of DB2 column functions rather than application program code
- Example: find how many employees make more than $10,000/month
  1. Select, fetching all 100000 employee rows
  2. Select Where Salary>10000, fetching 1000 rows
  3. Select Count Where ..., fetching 1 row
  - 100 times CPU time reduction possible from API elimination
  - Watch out for VSAM programmers, IO modules (stage 3 predicates)

Minimize #SQL Calls - continued

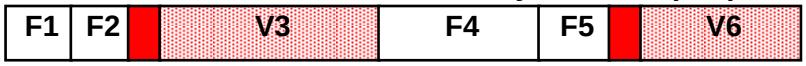
- Singleton SELECT is more efficient than OPEN, FETCH, CLOSE
- Fetch First N Rows Only in V7, in subquery V9
  - Limits the number of rows fetched to avoid fetching unwanted rows
  - Singleton Select (or SELECT INTO) can be used with Fetch First 1 Row even if multiple rows qualify
    - Avoids -811 SQLCODE
    - V8 supports ORDER BY for more meaningful query
  - Bigger improvement possible for CICS attach
- UPDATE without cursor is more efficient than OPEN, FETCH, cursor UPDATE, CLOSE
  - Up to 30% (possibly more if CICS) CPU time saving possible from singleton Select or Update compared to cursor operation

Minimize #SQL Calls - continued

- Reducing #SQL calls improves
  - API pathlength
  - Processor MIPS for row processing
    - Up to 2 to 3 times processor MIPS improvement possible from high-speed processor cache hit by repeated execution of a small set of modules/instructions and reduction in data moves
- V8 multi-row operation can significantly reduce the number of SQL calls issued
  - Up to 50% cpu reduction for simple (short-running) local Fetches, more for distributed

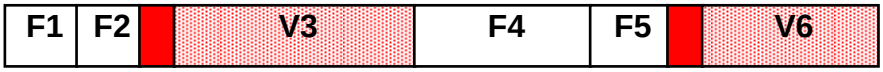
Minimize #Columns and Host Variables Referenced in SQL Calls

- Increasing order of cost
  - Local EBCDIC least -> ASCII or UNICODE or DRDA -> Single byte conversion -> Double byte conversion
  - Integer/char least and date/time/timestamp most expensive
- Try to avoid unnecessary columns
  - Doubled CPU time possible with 100 additional columns/host variables
- Put Varchar to end of row when many columns (>20)



V9 Varchar Performance Improvement

- **Remember the tuning recommendation for rows with many columns with any varchar present?**



  - **V9 DB2 internally executes this recommendation and more**

- **2 times or more improvement observed when many rows with many varchars are scanned and/or fetched using many predicates**
- **<5% improvement for a typical online transaction**
- **No difference if no varchar**
- **Reorg with rebuild compression dictionary if varchar columns when migrating to V9**

Minimize #Predicates Evaluated

- **Place most filtering predicates first in AND. (for predicates of the same type)**

WHERE HOME_STATE='MONTANA'	FF= 1%
AND HAIR='BROWN'	FF=10%
AND SEX='MALE'	FF=50%

  - **Weighted average of 1.01 predicates evaluated**
  - **If sequence of predicates is reversed, then the weighted average is 1.55, or 50% more predicate evaluation, which can lead to up to 20% cpu increase.**
- **Conversely, place most filtering predicates last in OR and IN-list without ACCESSTYPE=N.**

  eg STATE IN ('NEW YORK','FLORIDA','MONTANA')

Minimize #SQL Statements in a Program Where Possible

```
DO ....
    SELECT or INSERT or DELETE or UPDATE
END
                instead of
SELECT, INSERT, DELETE, or UPDATE
SELECT, INSERT, DELETE, or UPDATE
SELECT, INSERT, DELETE, or UPDATE
```

- Reduces EDM pool and thread storage
- Reduces allocate/deallocate cost at SQL execution and commit or deallocation
- Better exploitation of sequential detection and index lookaside
  - Potentially fewer Getpages, Lock requests, and faster I/O

Minimize # rows searched

- Try to get the maximum matching index columns for the best index filtering
- Insure predicate comparison for the same data type and length
  - Example: "where indexed-column=host-variable“
  - Especially prior to V8
    - V8 made most typical unlike data type comparisons stage 1 or sargable and indexable

Dynamic SQL

- Reduce dynamic bind frequency via
  - Dynamic statement caching with CACHEDYNAMIC YES
  - REOPT(ONCE) in V8 REOPT(AUTO) in V9
  - Improved monitoring in V8 Visual Explain
  - Next step in V9 Optimization Support Center
- Incremental bind in accounting
  - Static plan/package with VALIDATE(RUN) and bind time failure
  - Static SQL with REOPT(ALWAYS), or referencing Declared Temp Table, or private protocol in requestor

JDBC/SQLJ

- Use CACHEDYN YES for JDBC, or better yet use SQLJ or best choice is JLinQ (after DB2 9)
- Select/Update/Insert required columns only
  - More important in JDBC/SQLJ environment
- Store numeric as smallint or int to minimize conversion and column processing cost
  - Relative cost: Integer (lowest) -> Float -> Char -> Decimal -> Date/Time -> Timestamp (highest)
- Match Java and DB2 data type
  - V8 enhancement for non-matching data type

Existence Check

- SELECT FROM table WHERE EXISTS (SELECT FROM SYSIBM.SYSTABLES WHERE TYPE='A') …..
  - **In V7, all qualifying rows in this EXISTS subquery are retrieved and stored in a work file.**
    - **Select from SYSIBM.SYSTABLES where Type='A' Fetch First 1 Row Only followed by Select from outer table can be much faster.**
  - **In V8, this subquery execution is terminated as soon as a first qualifying row is found.**

Thread Reuse

- **Thread reuse for 5 to 20% cpu time reduction for light transactions**

NORMAL TERMINATION	AVERAGE	TOTAL
NEW USER	1.00	174752
DEALLOCATION	0	0
RESIGNON	0	0
INACTIVE	0	0

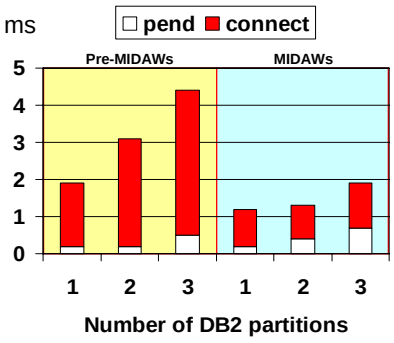
- **All except DEALLOCATION indicate successful thread reuse.**

Distributed/Stored Procedure

- Stored procedure to avoid DRDA overhead for each SQL call
  - Example: 10 Select, Insert, Update, and/or Delete calls in stored procedure
    - Results in 600us instead of 2100us overhead (10 SQL calls * 210us per SQL call) on z900 (2064-1) processor
    - Also faster response time because of as low as 1 rather than 10 message send/receive

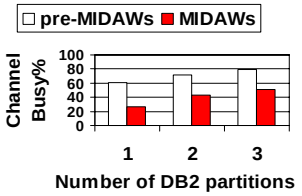
Figure 6-2 Channel speed comparisons with and without MIDAW

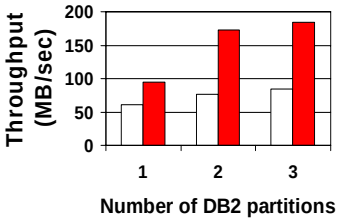
Parallel DB2 Table Scan, EF 4K (single channel)



Configuration:
MIDAW : z/OS 1.7
Pre-MIDAW: z/OS 1.4

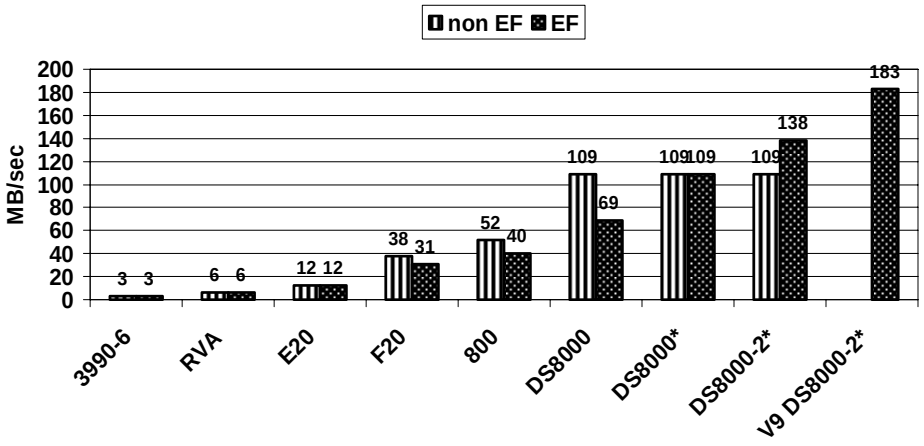
DB2 for z/OS Version 8
4000 byte row size
System z9 EC
FICON Express2
2 Gbit/sec link
DS8000 control unit





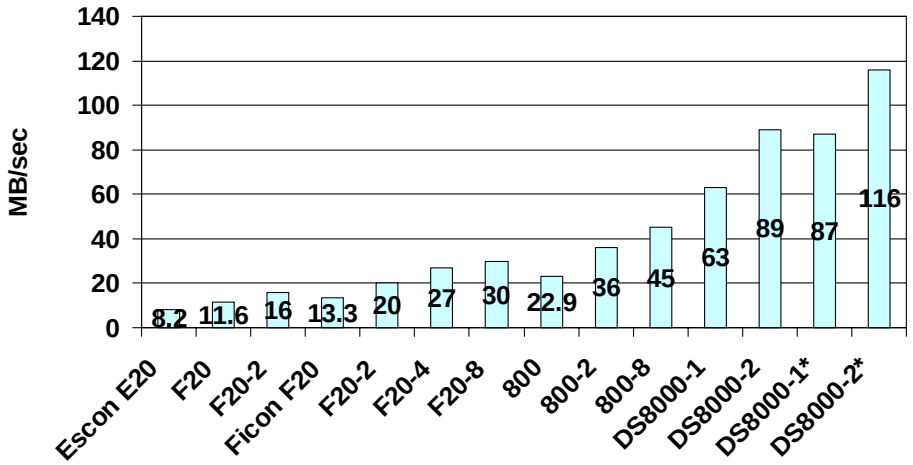
- This document contains performance information
- Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the numbers stated here.

Disk performance for sequential read



Maximum observed rate of active log write

- **First 3 use Escon channel, the rest is Ficon.**
- **-N indicates N i/o stripes; * MIDAW**

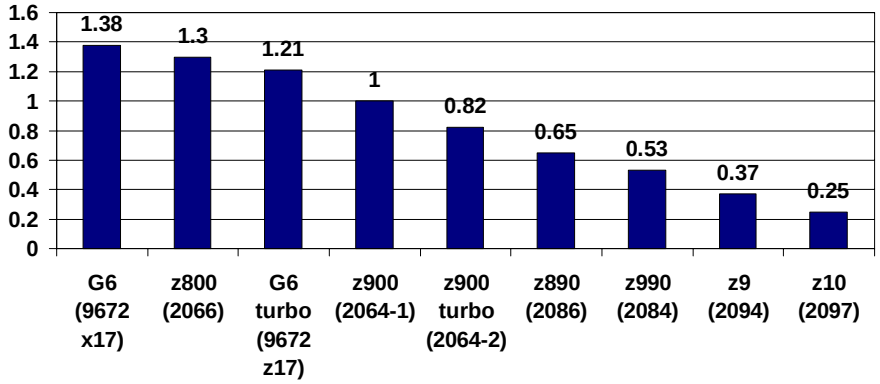


IO performance rules of thumb

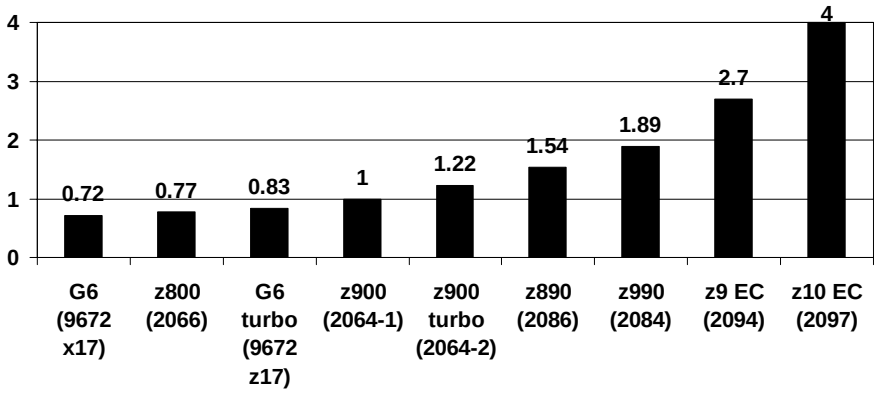
	4 K read	32 x 4K read
Old Rule of thumb	20 ms	64 ms
Current DS8300	1 - 2 ms	1.2 ms

Faster: 10 – 20 X 50 X

CPU Time Multiplier for some processor models



Recent single-processor relative CPU speeds



Index page split reduction

- **Bigger index page**
  - **4K, 8K, 16K, or 32K page**
    - **Up to 8 times less index split**
  - **Good for heavy inserts to reduce index splits**
    - **Especially recommended if high latch class 6 contention in data sharing**
      - ***Two forced log writes per split in data sharing***
    - **Or high latch class 254 contention in non data sharing shown in IFCID 57**

Index page split reduction - continued

- **Asymmetric index page split depending on an insert pattern**
  - **Instead of 50-50 split**
  - **Up to 50% reduction in index split**
  - **-20% class 2 cpu, -31% elapsed time, -50% log write i/o and async CF requests in one data sharing measurement**
    - **2 log write i/o's per split in data sharing**
  - **-10% cpu, -18% elapsed time, -20% index Getpage and BufferUpdate in one non data sharing measurement**

Access Path Enhancement

- **Cross query block optimization**
  - **Optimization across, rather than within, query blocks**
  - **More predicate transitive closure across query blocks**
- **Histogram statistics over a range of column values**
  - **Useful in range as well as equal predicates with high cardinality, eg Salary**
  - **Equal-depth (each interval with roughly same number of rows)**

Index Compression

Difference between data and index compression

	Data	Index
Level	**Row**	**Page (1)**
Comp on disk	**Yes**	**Yes**
Comp in Buffer Pool	**Yes**	**No**
Comp in Log	**Yes**	**No**
Comp Dictionary	**Yes**	**No (2)**
Average Comp Ratio	**10% to 90%**	**25 to 75% (3)**