2010 IDUG North America

IDUG

DB2 for z/OS Optimizing Insert Performance – Part 2

John Campbell & Frances Villafuerte
IBM DB2 for z/OS Development
Email: CampbelJ@uk.ibm.com & FrancesV@us.ibm.com

Session Code: B07

05/12/2010 11:00 AM – 12:00 PM
Platform: DB2 for z/OS

Objectives

- Understand typical performance bottlenecks
- How to design and optimise for high performance
- How to tune for optimum performance
- Understand the new features of DB2 V9 and DB2 V10
- Understand how to best apply and use new features

Agenda

- Typical Performance Bottlenecks and Turning
  - Index page
  - Application point of view
  - Bufferpool
  - Index
- DB2 10 Performance Enhancements
  - UTS enhancements, Index IO parallelism
  - Unique index with INCLUDE, log latch contention reduction
  - “Middle sequential” optimization, in-line LOB
- Summary

Typical Performance Bottlenecks and Tuning Observations

Large Index Page Size

- Available in V9 NFM
- Potential to reduce the number of index leaf page splits, which are painful especially for GBP-dependent index (data sharing)
  - Reduce index tree lotch contention
  - Reduce index tree p-lock contention
- Potential to reduce the number of index levels
  - Reduce the number of getpages for index traversal
  - Reduce CPU resource consumption
- Possibility that large index page size may aggravate index bufferpool hit ratio for random access

Large Index Page Size Examples

Rows In Table	1,000,000,000								
Key Length	4	8	16	32	64	128	256	512	1024
Page Size									
4096									
Entries/Leaf	336	252	168	100	56	29	15	7	3
Leafs	2,976,191	3,968,254	5,952,381	10,000,000	17,857,143	34,482,759	66,666,667	142,857,143	333,333,334
Non-Leaf fanout	331	242	158	93	51	26	13	7	3
Index Levels	4	4	5	5	6	7	9	11	19
8192									
Entries/Leaf	677	508	338	203	112	59	30	15	7
Leafs	1,477,105	1,968,504	2,958,580	4,926,109	8,928,572	16,949,153	33,333,334	66,666,667	142,857,143
Non-Leaf fanout	666	488	318	187	103	54	27	14	7
Index Levels	4	4	4	4	5	6	7	8	11
16,384									
Entries/Leaf	1360	1020	680	408	226	120	61	31	15
Leafs	735,295	980,393	1,470,589	2,450,981	4,424,779	8,333,334	16,393,443	32,258,065	66,666,667
Non-Leaf fanout	1,336	980	639	376	207	108	55	28	14
Index Levels	3	4	4	4	4	5	6	7	8
32,768									
Entries/Leaf	2725	2044	1362	817	454	240	123	62	31
Leafs	366,973	489,237	734,215	1,223,991	2,202,644	4,166,667	8,130,082	16,129,033	32,258,065
Non-Leaf fanout	2,676	1,963	1,280	755	414	218	111	56	28
Index Levels	3	3	3	4	4	4	5	6	7

Increased Index Look Aside

- Prior to DB2 9, used for clustering index only
- In DB2 9, now possible to use for additional indexes where CLUSTERRATIO >= 80%
- Potential for big reduction in the number of index getpages with substantial reduction in CPU time

Asymmetric Leaf Page Split

- Available in V9 NFM and automatic
- Design point is to provide performance relief for classic sequential index key problem
- Asymmetric index page split will occur depending on an insert pattern when inserting in the middle of key range
  - Instead of previous 50-50 split prior to DB2 9
  - Up to 50% reduction in index split
- Asymmetric split information is tracked in the actual pages that are inserted into, so it is effective across multiple threads across DB2 members

Asymmetric Leaf Page Split …

- APAR PK62214 introduces changes to the tracking and detection logic, and it should work much better for data sharing
  - Before: DB2 9 only remembered the last insert position and a counter
  - Now: DB2 remembers an insert 'range' and tolerates entries being slightly out of order
    - It may still not be effective for large key sizes (hundreds of bytes), or if entries come in very bad order (i.e., they do not look sequential)
    - But for simple cases like 3, 2, 1, 6, 5, 4, 9, 8, 7, 12, 11, 10 ... DB2 will be able to determine that the inserted entries are ascending

Randomised Index Key

- Index contention can be a major problem and a limit for scalability
- This problem is more severe in data sharing because of index page P-lock contention
- A randomized index key can reduce lock contention
- CREATE/ALTER INDEX … column-name RANDOM, instead of ASC or DESC
- Careful trade-off required between lock contention relief and additional getpages, read/write I/Os, and increased number of lock requests
- This type of index can provide dramatic improvement or degradation!
- Recommend making randomized indexes only when bufferpool resident

Identifying Unreferenced Indexes

- Additional indexes require overhead for
  - Data maintenance
    - INSERT, UPDATE, DELETE
  - Utilities
    - REORG, RUNSTATS, LOAD etc
  - DASD storage
  - Query optimization time
    - Increases DB2 Optimizer's choices to consider
- But identifying unused indexes is a difficult task
  - Especially in a dynamic SQL environment

Identifying Unreferenced Indexes …

- RTS records the index last used date
  - SYSINDEXSPACESTATS.LASTUSED
    - Updated once in a 24 hour period
      - RTS service task updates at first externalization interval (set by STATSINT) after 12PM
    - If the index is used by DB2, update occurs
    - If the index was not used, no update
- "Used" as defined by DB2 means:
  - As an access path for query or fetch
  - For searched UPDATE / DELETE SQL statement
  - As a primary index for referential integrity
  - To support foreign key access

Read and Write I/O for Index and Data

- Random key insert to index
  - N sync read I/Os for each index
    - N depends on # index levels, # leaf pages, and bufferpool availability
    - Index read I/O time = N * #indexes * ~1-2 ms
  - Sync data read I/O time = ~1-2 ms per page (0 if insert to the end)
  - Deferred async write I/O for each page
    - ~1-2 ms for each row inserted
    - Depends on channel type, device type, I/O path utilisation, and distance between pages
  - Recommend keeping the number of indexes to a minimum
    - Challenge the need for low value indexes

Read and Write I/O for Index and Data …

- Sequential insert to the end of data set
  - For data row insert, and/or ever-ascending or descending index key insert
  - Can eliminate sync read I/O
  - Deferred async write I/O only for contiguous pages
    - ~0.4 ms per page filled with inserted rows
    - Time depends on channel type, device type and I/O path utilisation

Bufferpool – Deferred write thresholds

- With high deferred write thresholds, write I/Os for data or index entirely resident in bufferpool can be eliminated except at system checkpoint or STOP TABLESPACE/DATABASE time
- Use VDWQT=0% for data bufferpool with low hit ratio (1-5%) if single thread insert
  - Else VDWQT=150 + # concurrent threads (e.g., 100) if sequential insert to the end of pageset/partition
  - When 250 buffers are updated for this dataset, 128 LRU buffers are scheduled for write
- Use VDWQT=0% for sequential index insert
- Use default if not sure, also for random index insert

Bufferpool – Deferred write thresholds …

- Recommendations on deferred write thresholds
  - VDWQT = Vertical (dataset level) Deferred Write Threshold
    - Default: when 5% of buffers updated from one dataset, a deferred write is scheduled
  - DWQT = bufferpool level Deferred Write Threshold
    - Default: when 30% of buffers updated, a deferred write is scheduled
  - Want to configure for continuous ‘trickle’ write activity in between successive system checkpoints
    - VDWQT and DWQT will typically have to be set lower for very intensive insert workloads

Active Log Write

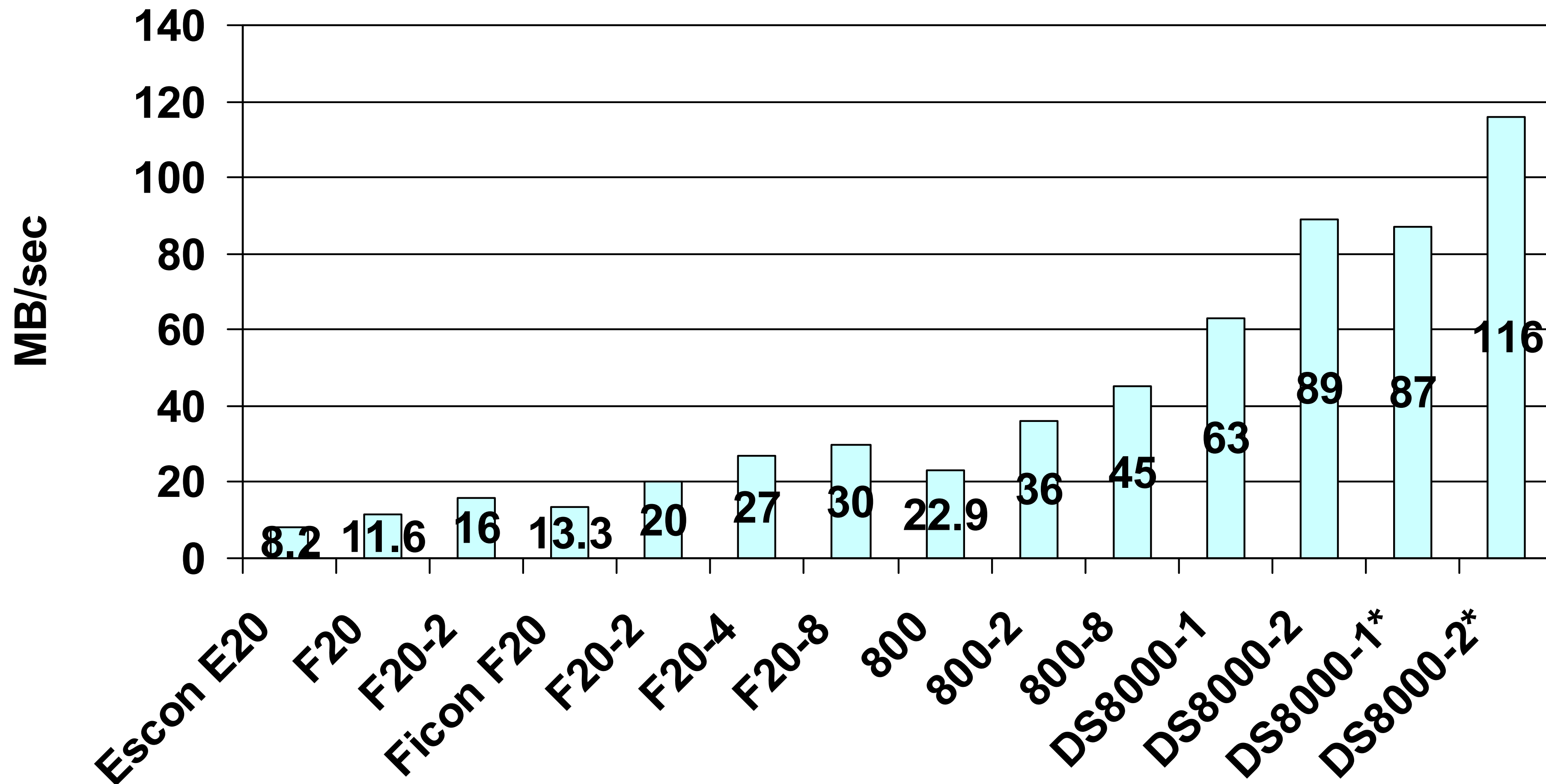
- Log data volume
  - From DB2 log statistics, minimum MB/sec of writing to active log dataset can be calculated as

    #CIs created in Active Log * 0.004MB
    ----
    statistics interval in seconds

  - Pay attention to log data volume if >10MB/sec
    - Consider use of DB2 data compression
    - Use faster device as needed
    - Consider use of DFSMS striping

Maximum Observed Rate of Active Log Write

- First 3 use Escon channel, the rest is Ficon
- -N indicates N I/O stripes; * MIDAW



Insert CPU Rough Rule of Thumb

To get the CPU time for other processor models, see http://www-03.ibm.com/systems/z/advantages/management/lspr/ on Internal Throughput of various IBM processors

	9672-Z17 CPU time
No index	40 to 80us
One index with no index read I/O	40 to 140us
One index with index read I/O	130 to 230us
Five indexes with index read I/O	500 to 800us

Insert CPU Rough Rule of Thumb …

- 9672-Z17 CPU time = 40 to 80us
  + 30 to 50us * number of indexes
  + 40us * number of I/Os

- Examples
  - If 1 index and no read I/O because of sequential index insert
    - 40 to 80us + 30 to 50us = 70 to 130us
    - CPU cost for write I/O can be ignored because of sequential write of contiguous pages
  - If 3 indexes and 1 random read I/O for each index
    - 40 to 80us + (30 to 50us)*3 + 40us*3*2 (read +write) = 370 to 470us

DB2 Latch Contention in Heavy Insert Application

- Latch Counters LC01-32 in DB2 PM/PE Statistics Report Layout Long
- Rule-of-Thumb on Internal DB2 latch contention rate
  - Investigate if > 10000/sec
  - Ignore if < 1000/sec
- Class 6 for latch for index tree P-lock due to index split - Data sharing only
  - Index split is painful in data sharing - results in 2 forced physical log writes
  - Index split time can be significantly reduced by using faster active log device
  - Index splits in random insert can be reduced by providing non-zero PCTFREE
- Class 19 for logical log write latch - Both non-data sharing and data sharing
  - Use LOAD LOG NO instead of SQL INSERT
  - Make sure Log Output Buffer fully backed up by real storage
  - Eliminate Unavailable Output Log Buffer condition
- If >1K-10K contentions/sec, disabling Accounting Class 3 trace helps to significantly reduced CPU time as well as elapsed time

Reduced LRSN Spin and Log Latch Contention

- Applies to data sharing
- Available in V9 and automatic
- Less DB2 spin in NFM for TOD clock to generate unique LRSN for log stream for a given DB2 member
- No longer holds on to log output buffer latch (LC19) while spinning (CM)
- Unique LRSN only required as it pertains to a single index or data page (NFM)
- Benefits
  - Potential to reduce LC19 Log latch contention
  - Potential to reduce CPU time especially when running on faster processor

Service Task Waits

- Service task waits most likely for preformatting
  - Shows up in Dataset Extend Wait in Accounting Class 3 Trace
  - Typically up to 1 second each time, but depends on allocation unit/size and device type
  - Anticipatory and asynchronous preformat in DB2 V7 significantly reduces wait time for preformat
  - Can be eliminated by LOAD/REORG with PREFORMAT option and high PRIQTY value
  - Do not use PREFORMAT on MEMBER CLUSTER table space with high PRIQTY because of 'long' first insert after page set open

Identity Column and Sequence Object

- DB2 to automatically generate a guaranteed-unique number for sequencing each row inserted into table
- Much better concurrency, throughput, and response time possible
  - Compared to application maintaining a sequence number in one row table, which forces a serialisation (one transaction at a time) from update to commit
  - Potential for 5 to 10 times higher insert/commit rate
- Option to cache (default of 20), saving DB2 Catalog update of maximum number for each insert
  - Eliminating GBP write and log write force for each insert in data sharing
- Recycling or wrapping of identity column and sequence value

GENERATE_UNIQUE()

- Built-in function with no arguments
- Returns a bit data character string 13 bytes long
- Provides a unique value which is not sequential
  - Unique compared to any other execution of the same function
- Allocation does not involve any CF access
- Based exclusively on STCK value
- DB2 member number and CPU number are embedded for uniqueness
- Example

```
CREATE TABLE EMP_UPDATE
(UNIQUE_ID CHAR(13) FOR BIT DATA,
EMPNO CHAR(6),
TEXT VARCHAR(1000)) ;

INSERT INTO EMP_UPDATE VALUES (GENERATE_UNIQUE(),
   ’000020’, ’Update entry...’) ;
```

Multi Row Insert (MRI)

- INSERT INTO TABLE for N Rows Values (:hva1,:hva2,…)
- Up to 40% CPU time reduction by avoiding SQL API overhead for each INSERT call
  - % improvement lower if more indexes, more columns, and/or fewer rows inserted per call
- ATOMIC (default) is better from performance viewpoint as create of multiple SAVEPOINT log records can be avoided
- Implication for use in data sharing environment (LRSN spin)
- Dramatic reduction in network traffic and response time possible in distributed environment
  - By avoiding message send/receive for each row
  - Up to 8 times faster response time and 4 times CPU time reduction

DB2 V10 Performance Enhancements

Enhancements to PBG

- Explicitly adding partition to PBG
  - Improve performance of adding partition during INSERT
  - Provide flexibility for DBA to monitor space growth
  - Dictionary page is not copied from previous partition during ALTER
    - Compression dictionary can be generated during INSERT
  - Beneficial for transfer data between systems using DSN1COPY
  - Example:
    - ALTER TABLE …. ADD PARTITION
    - CREATE TABLESPACE .... NUMPART n …
- Create more than one partition on PBG
  - Keyword NUMPART on CREATE TABLESPACE DDL also applied to PBG
  - Example
    - CREATE TABLESPACE TS1 IN DB1
NUMPARTS 3

Index I/O Parallelism for index updates

- Transactions that perform inserts into tables with many indexes defined can experience high response times due to index I/O wait time as the index updates are performed sequentially
- I/O parallelism for index updates is to improve the insert workload throughput performance by overlapping the synchronous I/O wait time for different indexes on the same table
  - Need 3 or more indexes defined on table to benefit
  - Pages from different indexes on the same table prefetched into bufferpool in parallel for insert
  - Still one processing task even though the I/O operations are overlapped

Unique Index with INCLUDE columns

- Index is used to enforce unique key constraint on table
- Additional indexes are required to achieve index only access or index only filtering during query processing for columns that are not part of the unique key constraint
- Additional non-key columns can now be defined on a unique index to reduce the total number of indexes
- Indexes that participate in referential integrity (RI) will support the additional columns, however these columns will not be used to enforce RI
- Improvements expected
  - Faster insert performance as less indexes need to be updated
  - Reduced DASD space usage
  - Potential to stabilize query access path as less similar indexes to chose from

Log Latch contention reduction

- The log latch, LC19, can be a major source of contention for workloads that have high insert activity
- DB2 V9 improved the situation for many data sharing systems
- DB2 10 provides improvement for both data sharing and non-data sharing subsystems
- The log latch is still used, but is used for a much shorter duration
- LC19 contention is still possible
- Log latch is still used for –SET LOG SUSPEND processing

Index Look Aside for Referential Integrity

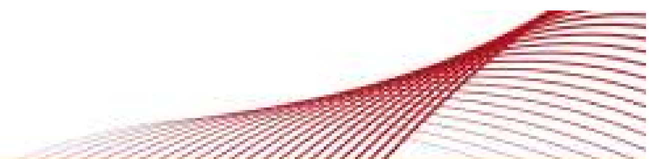
- Sequential detection and index look aside
- Avoid RI check for each insert of a child under the same parent

Sequential insert into middle of clustering index

- "Middle-sequential" ("Spot Sequential") inserts
- Optimization when index manager picks the candidate RID during sequential insert (next lowest key rid)
- Higher chance to find the space and avoiding a space search

In-line LOB function

- Only available in V10 NFM for table that is in UTS table space with RRF format
- In-line portion of the LOB resides in the base table instead of LOB table space
- Improve performance for small LOB or portion of LOB data that is frequently referenced
- Combine with Defer Define option could save DASD
  - LOB table space is not allocated until the first insert to the LOB table space
  - Possible DBD contention when table has many LOB, AUX columns during the define process in concurrency environment
- Use proper page size
  - Small page size could cause less record inserted into page and more space map page update is needed
- ALTER TABLE to alter the in-line length



Summary

Summary – Key Points

- Decide whether the data rows should be clustered/appended at the end
- Sort inserts into clustering key sequence
- Use partitioned tablespace and partitioned indexes
- Keep the number of indexes to a minimum and drop low value indexes
- Tune deferred write thresholds and distributed free space to drive ‘trickle write’
- Use large PRIQTY/SECQTY and large SEGSIZE to reduce frequency of exhaustive space search
- Use data compression to minimise log record size
- Use faster channel, faster device, DFSMS striping for active log write throughput
- Use MEMBER CLUSTER and TRACKMOD NO to reduce spacemap page contention and when using LOCKSIZE ROW to reduce data page contention
- Use Identity column, sequence object, GENERATE_UNIQUE() built-in function with caching to efficiently generate a unique key

Thank You !

John Campbell CampbelJ@uk.ibmcom
Frances Villafuerte Francesv@us.ibm.com