



Information
ON Demand 2010

Create Insight. Transform. Go Beyond.



DB2 for z/OS Optimising Insert Performance

John Campbell

Distinguished Engineer

IBM DB2 for z/OS Development



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
- Understand how to best apply and use new features

Agenda

- Insert Space Search
- Most Important Tablespace Attributes
- Typical Performance Bottlenecks and Tuning
- DB2 9 Performance Enhancements
- Summary

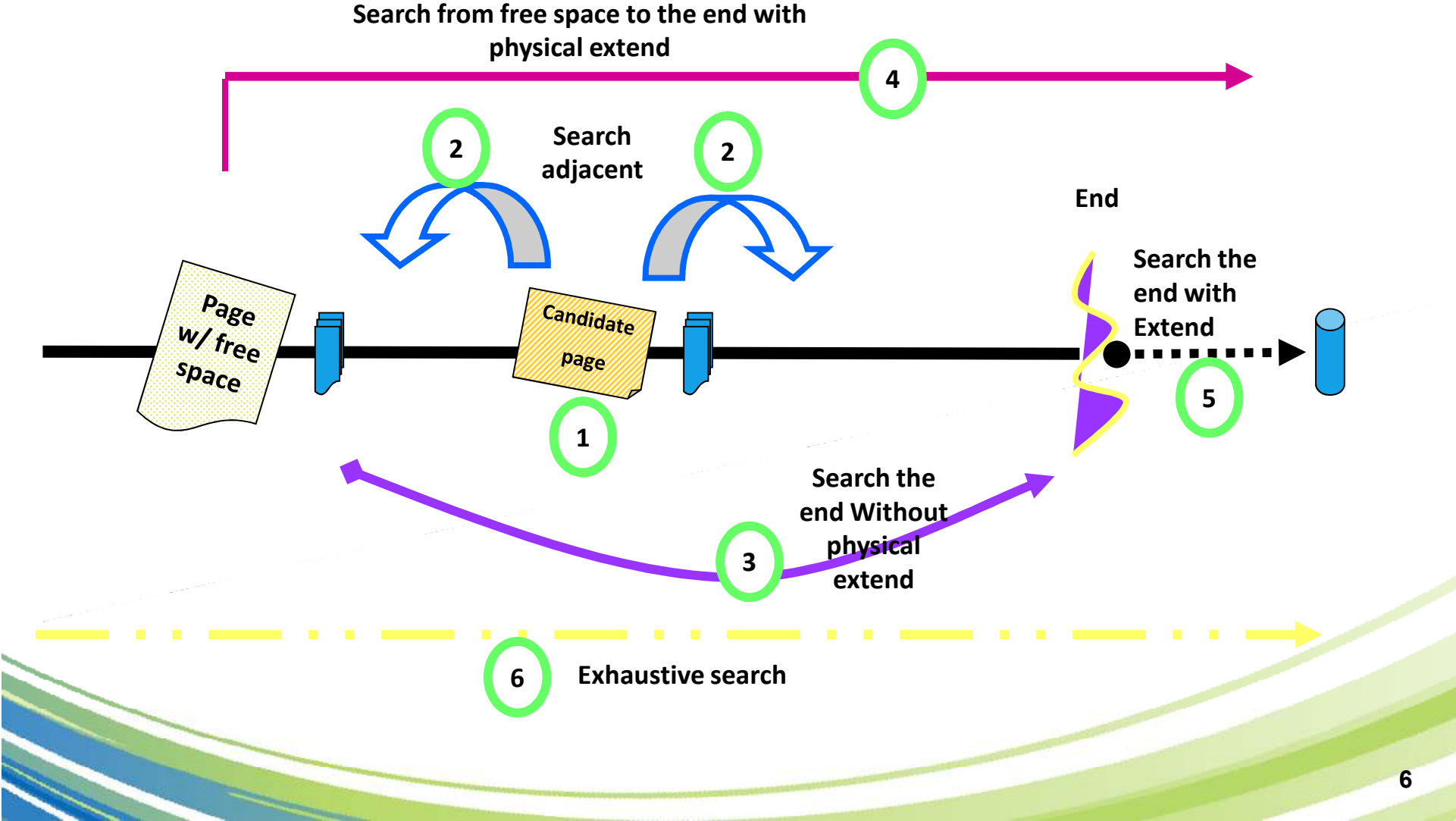
Key Physical Design Questions

- Design for maximum performance throughput or space reuse?
- Random key insert or sequential key insert?
- Store rows in clustering sequence or insert at the end?
- Input records sorted into clustering key sequence?
- What are indexing requirements and are they justified?

Choice: Performance or Space Reuse

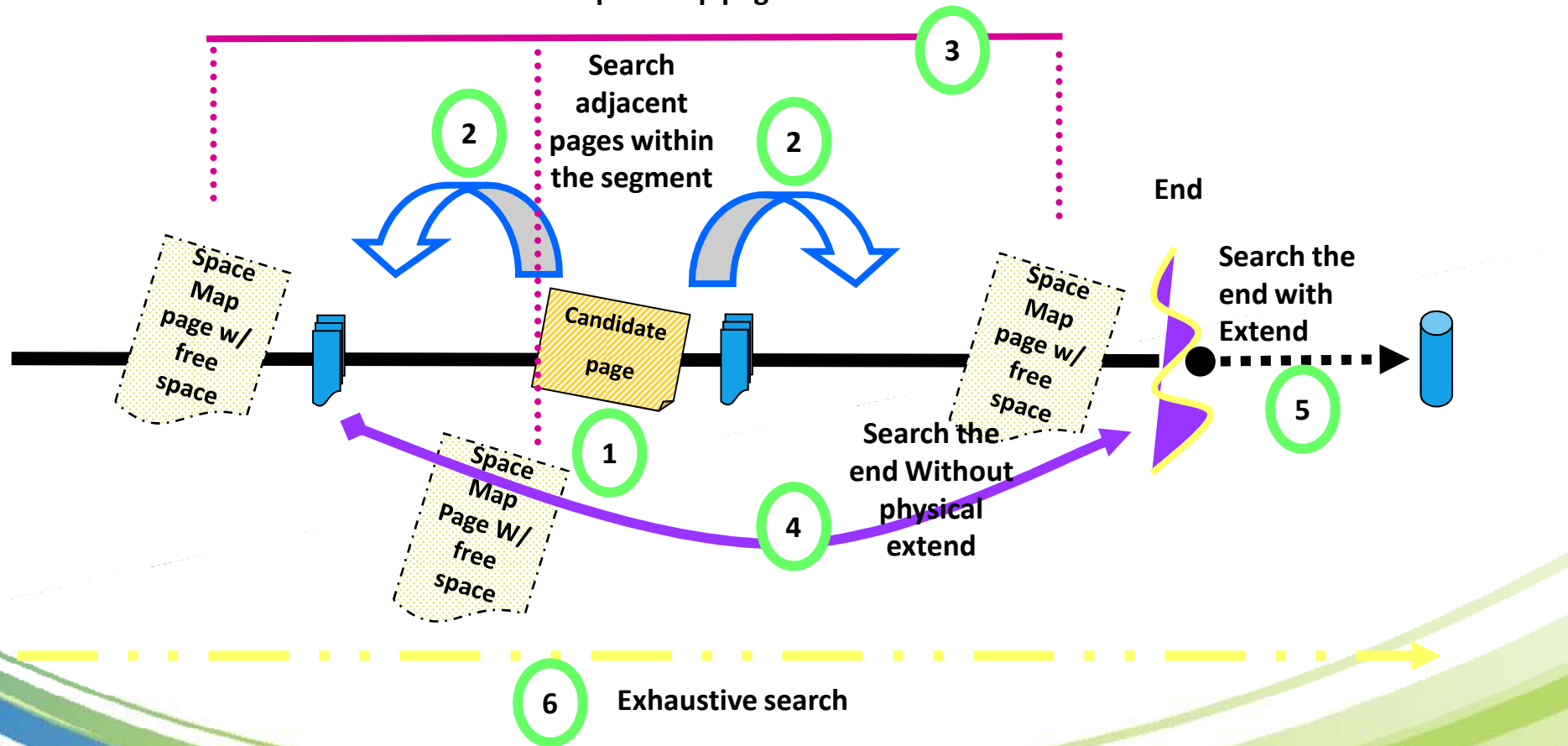
- High performance, than less space reuse
- Better space reuse, than less performance
- Classic partitioned tablespace
 - Usually better performance especially in data sharing environment
- Segmented or Universal tablespace
 - Usually better space management due to more space information in spacemap pages

Insert – Space Search Steps (Partitioned Tablespace)



Insert – Space Search Steps (Segmented Tablespace)

Search the spacemap page that contains lowest segment that has free space to the end of spacemap page



Most Important Tablespace Attributes

Segmented Tablespace

- Segmented tablespace provides for more efficient search in fixed length compressed and true variable length row insert
 - Spacemap page contains more information on available space so that only a data page with guaranteed available space is accessed
 - 2 bits per data page in non segmented tablespace ($2^{**}2=4$ different conditions)
 - 4 bits per data page in segmented tablespace ($2^{**}4=16$ different conditions)
 - But more spacemap page updates
 - Possible performance penalty with data sharing

Segmented Tablespace ...

- SEGSIZE
 - General recommendation is to use large SEGSIZE value consistent with size of pageset
 - Typical SEGSIZE value 32 or 64
 - Large SEGSIZE
 - Provides better opportunity to find space in page near by to candidate page and therefore maintain clustering
 - Better chance to avoid exhaustive space search
 - Small SEGSIZE
 - Can reduce spacemap page contention
 - But less chance of hitting 'False Lead Threshold' of 3 and looking for space at the end of pageset/partition
 - 'False Lead' is when spacemap page indicates there is a data page with room for the row, but on visit to the respective data page this is not the case
- Also applies to Universal Table Space (DB2 9)

Partitioned Tablespace

- Use page range partitioning by dividing tablespace into partitions by key range
- Spread insert workload across partitions
- Can reduce logical and physical contention to improve concurrency and reduce cost
- Separate index B-tree for each index partition of partitioned index (good for concurrency)
- Only one index B-tree for non-partitioned index (bad for concurrency)
- Over wide partitioning has potential to reduce number of index levels to reduce performance cost

Trade-off in Free Space Search

- Insert to the end of pageset/partition
 - To minimize the cost of insert by minimising
 - Read/Write I/Os, Getpages, Lock requests
- Search for available space near the optimal page
 - To store data rows in clustering index sequence
 - To store leaf pages in index key sequence
 - To minimize dataset size
- Search for available space anywhere within the allocated area
 - To minimise dataset size
 - Can involve exhaustive space search which is expensive
 - Use large PRIQTY/SECQTY and large SEGSIZE to minimize exhaustive space search

Distributed Free Space

- Use distributed free space – PCTFREE and/or FREEPAGE
 - For efficient sequential read of index
 - For efficient sequential read of data via clustering index
 - To minimize index split
- Carefully calculate settings
- Default distributed free space
 - 0 FREEPAGE
 - 5% PCTFREE within data page
 - 10% PCTFREE within index page

Distributed Free Space ...

- For best insert performance
 - Random key insert to index
 - Use non-zero index PCTFREE and/or FREEPAGE
 - To reduce index leaf page splits
 - For efficient sequential index read
 - Sequential key insert to index
 - Immediately after LOAD, REORG, or CREATE/RECOVER/REBUILD INDEX
 - Use PCTFREE 0 to reduce the number of index pages and possibly index levels by populating each leaf page 100%
 - Use PCTFREE=FREEPAGE=0 for data to reduce both sync read and async write I/Os for each row insert
 - Possible performance penalty for query in terms of sync single page I/O when reading multiple rows via clustering index

TRACKMOD NO

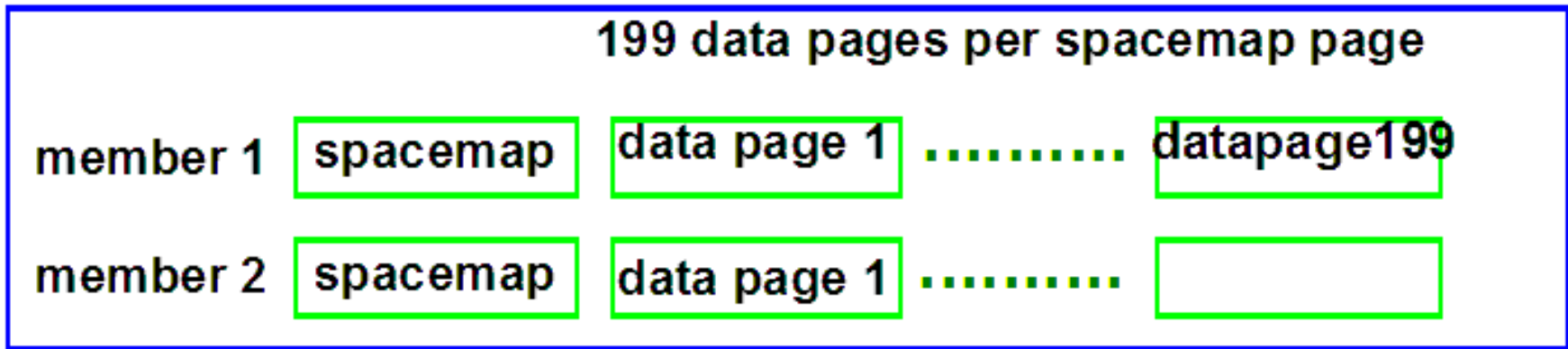
- Reduces spacemap contention in data sharing environment
- DB2 does not track changed pages in the spacemap pages
- It uses the LRSN value in each page to determine whether a page has been changed since last copy
- Trade-off as degraded performance for incremental image copy because of tablespace scan

MEMBER CLUSTER

- INSERT into “hot spots”
 - Resulting in excessive page p-lock and page latch contention on
 - Spacemap page
 - Data page if LOCKSIZE(ROW)
 - Can occur when
 - Concurrent insert in same area of the tablespace among different members in a data sharing environment
 - Tablespace has row level locking attribute
 - True sequential key or spot sequential key
- MEMBER CLUSTER
 - Member-private spacemap and corresponding data pages

MEMBER CLUSTER ...

- Rows inserted by Insert SQL are not clustered by clustering index
 - Instead, rows stored in available space in member-private area
- Option not available on segmented tablespace or UTS



MEMBER CLUSTER

- Can reduce spacemap contention
 - Used in the high concurrent insert in the data sharing environment
 - Can reduce spacemap sharing between data sharing members
 - In non-data sharing environment, smaller spacemap page can reduce the interest of spacemap page between threads
- Data clustering can be restored via REORG
- Assuming insert-only workload, may want to use LOCKSIZE ROW and larger data page size when using MEMBER CLUSTER
 - Better space use
 - Reduce working set of bufferpool pages
 - Reduce # Getpages
 - Reduce # Lock Requests
 - Reduce # CF requests

MEMBER CLUSTER, PCTFREE = 0 and FREEPAGE = 0

- Introduced in V7 APAR PQ87381
- Re-instated in V9 via APAR PK81470
- Only applies to classic partitioned tablespace / simple tablespace
- Insert to the end of tablespace without looking at the free space
- Will reduce longer chain of spacemap page search as tablespace gets bigger
- Good for tablespace with seldom delete or update that creates free space
- Exhaustive search through spacemap pages belong to the member before physical extend
 - Success depends on deletes and inserts being spread across members

MAXROWS n

- Optimisation to avoid wasteful space search on partitioned tablespace in fixed length compressed and true variable length row insert
- Must carefully estimate 'average' row size and how many 'average' size rows will fit comfortably in a single data page
- When MAXROWS n is reached the page is marked full
- But introduces on going maintenance challenges
 - Could waste space?
 - What happens if compression is removed?
 - What happens if switch from uncompressed to compressed?
 - What happens when new columns are added?

Typical Performance Bottlenecks and Tuning Observations

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 buffer pool availability
 - Index read I/O time = $N * \# \text{indexes} * \sim 1\text{-}2 \text{ ms}$
 - Sync data read I/O time = $\sim 1\text{-}2 \text{ ms}$ per page (0 if insert to the end)
 - Deferred async write I/O for each page
 - $\sim 1\text{-}2 \text{ ms}$ 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

Read and Write I/O for Index and Data ...

- 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 = buffer pool 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

Read and Write I/O for Index and Data ...

- With high deferred write thresholds, write I/Os for data or index entirely resident in buffer pool can be eliminated except at system checkpoint or STOP TABLESPACE/DATABASE time
- Use VDWQT=0% for data buffer pool 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

Active Log Write

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

$$\frac{\text{\#CIs created in Active Log} * 0.004\text{MB}}{\text{statistics interval in seconds}}$$

- 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

Lock/Latch and Service Task Waits

- Rule-of-Thumb on LOCKSIZE
 - Page lock (LOCKSIZE PAGE|ANY) as design default and especially if sequentially inserting many rows/page
- Page P-lock contention in data sharing environment
 - Index page update
 - Spacemap page update
 - Data page update when LOCKSIZE ROW

Lock/Latch and 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 tablespace with high PRIQTY

DB2 Latch Contention

- Latch Counters LC01-32 in OMPE Statistics Report Layout Long
- Rule-of-Thumb on Internal DB2 latch contention rate
 - Investigate if $> 10000/\text{sec}$
 - Ignore if $< 1000/\text{sec}$
- 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 $>1\text{K}-10\text{K}$ contentions/sec, disabling Accounting Class 3 trace helps to significantly reduced CPU time as well as elapsed time

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 9 Performance Enhancements

Reduced LRSN Spin and Log Latch Contention

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

Large Index Page Size

- Available in NFM
- With or without index compression
- Potential to reduce the number of index leaf page splits, which are painful especially for GBP-dependent index (data sharing)
 - Reduce index tree latch 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 buffer pool 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, for clustering index only
- In DB2 9, now possible for additional indexes where CLUSTERRATIO $\geq 80\%$
- Potential for big reduction in the number of index getpages with substantial reduction in CPU time

Asymmetric Leaf Page Split

- Available in 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
- 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!
 - e.g. Very bad choice for applications with range predicates
- Recommend making randomized indexes only when buffer pool 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

Table APPEND Option

- New APPEND option is provided for INSERT
 - CREATE/ALTER TABLE ... APPEND YES
- Always use with MEMBER CLUSTER in data sharing
- Will reduce longer chain of spacemap page search as tablespace keeps getting bigger
- But will drive need for more frequent tablespace reorganization
- Degraded query performance until the reorganization is performed
- Behaviour the same as 'pseudo append' with "MC00"
 - MEMBER CLUSTER and PCTFREE=FREEPAGE=0
 - Will switch between append and insert mode
 - Success depends on deletes and inserts being spread across DB2 members of data sharing group

Summary

Summary – Key Points

- Decide whether the data rows should be clustered/appended at the end
- Sort inserts into clustering key sequence
- Use classic 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
- Important new DB2 9 new feature functions such as large index page size