Bullets

• Understand the database design techniques with the PBG tablespaces
• Learn about the latest zParms and the optimum settings for OLTP or DW BI systems
• Learn the latest coding considerations and techniques for Java and .NET applications
• Find out the latest recommendations for application performance coding
• Understand the latest consideration and recommendations for efficient setup and maintenance of your table spaces and indexes
Discoveries and Recommendations

- **Architecture**
- **Systems**
- **Database**
- **Applications**
- **SQL & Access Paths**
Process and Procedures – Good!

- Procedures need to be standardized
  - Different COBOL and JAVA processes
    - Business driven analysis for entire environment
    - EXPLAIN/SQL peer review and verification

- How are services reused, shared and improved?
  - COBOL versus JAVA
  - New for every project
    - How many services are shared between applications?
    - Standardized architecture and services
      - Stay away from AJAX, Spring, ibatis, hibernate, etc.....
Resources

- Developer/DA/DBA ratios need to be improved
- Service point analysis/Function point
- Management support structures

- Consolidated testing environment
  - Development should use consolidated environment
  - Improved QA/performance testing environment
  - Some statistics on business ‘click’ activities

Statistics on Business ‘click’ Activities

- No business capacity figures and history
  - Business increases XX%
    - CPU capacity needs to increase CC%
    - Disk space needs to increase DD%
    - Developers/Support staff needs to increase by PP%

- Number of ads, customers and orders/items
  - History of metrics of profits, loss and performance
  - Develop these business conversations
    - Relate them to ALL of your systems
SUSPENDS, DEADLOCKS & TIMEOUTS

- SUSPENDS - I wait while you process
- DEADLOCKS – I can’t process, you have it
- TIMEOUTS – I waited too long for that data

- System, database(s) & application contention
  - Slows all aspects of processing down—
    **everything**
    - Very CPU & I/O intensive
      - Expensive to manage, negotiate & resolve
**Environment Snapshot**

**SNAPSHOT FOR:** 04/04/10 05:39:17 **THRU:** 04/05/10 14:56:56 **DELTA:** Y

**DB2 COMMAND:**

**TIME:** 14:56:56

**CPU:**

- **BP:** 15.11
- **BPX1:** 17.05
- **BPX8:** 24.68
- **BPX80:** 19.63
- **BPX81:** 19.63

**DB HITS:**

- **BPX1:** 19.63
- **BPX8:** 19.63
- **BPX80:** 19.63
- **BPX81:** 19.63

**PT HIT %:**

- **BPX1:** 19.63
- **BPX8:** 19.63
- **BPX80:** 19.63
- **BPX81:** 19.63

**LOADING**

- **ARCHIVE READS:** 37
- **SQL TOTAL:** 74572463
- **DEADLOCKS:** 63
- **LOG WRITE WRITES:** 0
- **LOG TOTAL:** 2682
- **THROTTLE:** 390
- **CHECKPOINTS:** 1834
- **UTIL TOTAL:** 47840

**WISCELLANEOUS**

- **THREADS CREATED:** 6220000
- **TOTAL CPU:** 80.51:10.0
- **SYSTEM CMD:** 11258
- **BEND/ALLOC STATS:** 19513211
- **STORAGE LIMIT:** 25540
- **EXCEPTIONS:** 15500

**DATA DISPLAYS**

**CURSORS not being CLOSED Improved**

- **OPEN CURSOR, FETCH & CLOSE CURSOR**
  - This is the standard DB2 application transaction
  - Commits data & cleans up the all WAS & DB2 resources
    - ResultSet.close();
    - queryStmt.close();
    - Conn.close();
  - Then allow the connection to be released & reused
- **313m OPEN CURSOR**
- **1.9B FETCH**
- **NOW! 268m CLOSE CURSORS was .75m per hr**
- **55M delta, 5M/day, - only 208,333 per hr**
RID Pool Failures 4/15

- RID Pool Failures turns into a TS Scan
  - Huge # of locks, # of I/Os & amount of CPU
  - LONG long run-time for applications
  - 370 every day & was probably much worse during peak

```
DB2 V9: OSN CURRENT TIME: 14:36:58
COMMAND:   \n
INTERVAL FOR: 04/04/10 08:30:17 THRU 14:36:32

EXECUTION REPORT: Data Manager RID Statistics

TERMINATED - RIDS LIMIT : 4063
TERMINATED - STORAGE : 0 PATH NOT USED
TERMINATED - ON LIMIT : 0 NO STG FOR RIDS : 78
TERMINATED - CONCURRENT : 78 RETRIEVED EXCEEDED MAX : 5536
BLKS CURRENTLY ALLOCATED: 0
HWM CONCURRENT ALLOCATED: 2840
```

Cost of Down Time?

- Disaster Recovery
- Business Continuity
  - Down time cost
- Monitoring the transaction response time
  - Incentives for limiting #abends and
  - 99.97% uptime guarantee? How
Database Design Considerations - RI

Referential Integrity (RI) extensively used

- 119 RI levels is potential very costly
  - 51 levels is the highest seen in 15 years
    - Less than 10 levels is usually recommended
    - Would like more details on why 119 levels

38 levels potentially CASCADE delete
- Delete 1 row CASCADEs through 38 tables
- Could be 100K rows or more
  - Good or bad? Need more analysis
Table Compression - 4/16

- Compression of DB2 data
  - Saves CPU & I/O for all processing
  - Compression guidelines
    - Tables > 500,000 rows with row length >75
    - 10 gig – 12,846 cyls potentially can be compressed
    - More processing and usage analysis needed

- Negative Compression is happening
  - Wasting space compressing wrong tables

Partitioning Requirements

- Many million row tables that need partitioning
  - Clustering – inserting data on same page
  - Segmented tablespace is ineffective for millions

- **Table > 100,000 rows should be partitioned**

- **Separation of Operational and Historical**
Lock Escalation

• Application is holding too much data
  – Locking many rows *escalated* to locking entire table
    • Row level locking (RRL) is used on several tables
    • RRL might not be right choice for ALL situations

• Application flow needs analysis/modification
  – Why does it need to lock that much data?

• LOCKMAX keyword used on the definitions
  – **LOCKMAX should be removed**

Improved maintenance

• More Reorgs
• Separation of TS away from IX
• Separation of System away from Applications
  – All on the same disk volume
  – No data set performance priorities for data
Segmentation Strategies

- Separation of Operational vs. Historical
- ODS
  - Replication
- MQTs
- Versioning
  - UNION ALL Views
- IBM’s Smart Analytic Optimizer
- Temporal & bi-temporal tables - DB2 10

UNION ALL Views

- Providing SELECT Transparency

CREATE LOGICAL_TABLE VIEW AS

SELECT columns FROM LOGICAL_TABLE, other tables WHERE some amazing filters

DB2 Query Rewrite

SELECT FROM
MQT – 10 to 1000 times improvement!

- 5B rows per year–10 per 4k page= ½B pages
- MQT aggregates save large amounts of everything
  - Create aggregates for every possibility
    - "On Demand" information
    - Sales by department
    - Sales by zip code
    - Sales by time period – day/week/month/quarter/AP
  - All reporting and analysis areas
  - Trace usage to create/eliminate aggregates
- Total by month ½B I/Os versus 12 I/Os

Application

Discoveries and Recommendations
End to End Monitoring

Network Performance

- Vendor benchmarks are perfect, your environment isn’t
  - It will never go as fast every environment is different
  - System is more likely to be I/O or CPU performance constrained
  - Keep historical records of performance elements

- Synchronize monitoring – hardware, software, network & users
  - Understand all the elements of the performance puzzle
  - Only monitor what you can change or manage
    - Too much monitoring overhead will become a problem

- Report start/end duration of activities
  - Use accurate tools - A fool with a tool is still a fool!
  - Use automated testing whenever possible

Need to develop inventory list of reports available
For both the MF & WAS

1. Understand what reports/elements details are related across platforms
2. Understand how they are acquired
3. Test run gathering everything for a five minute time frame
4. Automate and generate the reports from a script or a click of a button
5. Gaps within the type of information available

I/O–CPU–Memory–Conn Id–SQL–Method Name
JVM – Deeper Dive Needed

- JVM Heap Size
  - How big should your Heap be?
- Garbage Collection Frequency
  - How often do you take it out?
- Garbage Collection Time
  - How long does it take?
- Average JVM Heap Size After Garbage Collection
  - Does it make any difference?
- Garbage Collection Frequency
  - Multiple GC at once
    - How long does GC take
    - How big performance impact is GC

- JVM garbage collector takes care of it
  - Clean up unreferenced memory items
- Testing Methodology
  - Concurrency test
    - How many users, application types and overlap duration
  - Long running test
    - Memory problem happen after application runs five hours
  - Repetitive test
    - Application memory issues
- JVM Settings
  - -XX:MinHeapFreeRatio=\%
  - -XX:MaxHeapFreeRatio=\%
  - -XX:NewSize=bytes
  - -XX:MaxNewSize=bytes
  - -XX:NewRatio=value
  - -XX:SurvivorRatio=number
  - -XX:TargetSurvivorRatio=\%
  - -XX:MaxPermSize=MB
  - -XX:-CleanPagesOnUncommit
- Program compilation options
  - -XX:-InlineUnreachedCalls
  - Defaults are usually okay

pureQuery Programming Advantages

- Static Bind removes java dynamic overhead
  - Reduces the Dynamic Statement Cache
  - Reduces EDM Pool overhead
  - Allows better memory tuning capabilities
- Faster overall execution by .001 per transaction
  - 20 million java dynamic transactions per day
  - 333 minutes of CPU saved per day
    - Chargeback $10-$38 per minute = $12,654.00/day
    - 250 business days > $3 million per year
    - Reduced CPU demand > 5.5 hours of CPU per day

Your mileage may vary
**SQL & Access Paths**

Discoveries and Recommendations

---

**TOP 10 SQL Problems**

- Delayed filtering on COLL
  
  AND C.STAT_CD = 'D1'
  
  AND C.STAT_REAS_CD = '500'

- COLL02 index has
  
  - STAT_CD.STAT_REAS_CD
  
  - Needs STAT_CD

- Sorts: ORDER BY LAST_UPDT_TS DESC
  
  - Recommendation:
    
    - Add Index IXDEMOI12
      
      - LAST_UPDT_TS DESC
### Analysis of Top # Execs

<table>
<thead>
<tr>
<th>STMT ID</th>
<th># EXECS</th>
<th>CPU TIME</th>
<th>Avg CPU Time</th>
<th>ELAPSED TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>000021999</td>
<td>93914</td>
<td>20.686235</td>
<td>0.000220</td>
<td>38.791262</td>
</tr>
<tr>
<td>000022000</td>
<td>93914</td>
<td>12.685280</td>
<td>0.000135</td>
<td>15.524048</td>
</tr>
<tr>
<td>000023680</td>
<td>78692</td>
<td>36.850395</td>
<td>0.000468</td>
<td>47.273804</td>
</tr>
<tr>
<td>000023682</td>
<td>78658</td>
<td>13.619133</td>
<td>0.000173</td>
<td>53.361565</td>
</tr>
<tr>
<td>000023681</td>
<td>78650</td>
<td>15.140486</td>
<td>0.000193</td>
<td>55.643611</td>
</tr>
<tr>
<td>000023679</td>
<td>78475</td>
<td>8.522043</td>
<td>0.000109</td>
<td>17.090447</td>
</tr>
<tr>
<td>000021428</td>
<td>53372</td>
<td>8.715232</td>
<td>0.000163</td>
<td>28.730075</td>
</tr>
<tr>
<td>000021429</td>
<td>53372</td>
<td>6.328804</td>
<td>0.000119</td>
<td>29.682331</td>
</tr>
<tr>
<td>000021427</td>
<td>53372</td>
<td>5.646092</td>
<td>0.000106</td>
<td>23.316107</td>
</tr>
</tbody>
</table>

Need to drill down

### TOP 10 SQL Problems

- **DB2 10 Index INCLUDE Option**
  - INCLUDE LAST_UPDT_TS DESC
- **Put OPTIMIZE FOR 1 ROW clause**
  - To turn off Merge Join
- **Sort cannot be satisfied with 3 column index**
  - due to range condition on second column
Questionable Access Paths

• List Pre-fetch, Multiple Index Access and Hybrid Join
• Are usually good when fetching all rows that qualify
• Caught via monitor exceeding limit
• RID list sizes varied across queries

Many Access Path Types

• Table Access
  – Tablespace scan – segmented or partitioned
• Index Access
  – One Fetch
  – Nested loop join
  – Merge Scan join
  – Star Join – Cartesian or Pair-wise
  – Outer join – Left/Right
  – Hybrid join – 2 Types: C or N
  – Multiple index access
  – Matching Index access
  – Non-Matching Index Access
  – List Pre-fetch
  – Index Lookaside

• Join tables based on criteria
• Qualify rows – filtering data
• Aggregate data - grouping
• Ordering data - Sorts

• How much CPU and I/O does your access path require?
Expensive Access Paths

- Expensive access paths need justification
- Great when all result rows are required
- High percent will *not* be justified

<table>
<thead>
<tr>
<th>Access Path</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX SCAN</td>
<td>1</td>
</tr>
<tr>
<td>LIST PREFETCH</td>
<td>29</td>
</tr>
<tr>
<td>LIST PREFETCH</td>
<td>1</td>
</tr>
<tr>
<td>LIST PREFETCH</td>
<td>4</td>
</tr>
<tr>
<td>MATERIALIZED VIEW</td>
<td>31</td>
</tr>
<tr>
<td>MATERIALIZED VIEW</td>
<td>6</td>
</tr>
<tr>
<td>MULTIPLE INDEX ACCESS</td>
<td>2</td>
</tr>
</tbody>
</table>

500,000 rows or more

Explain, Explain, Explain, Explain, Explain

- Know your SQL access paths
- Tablespace scans should be eliminated
  - 15% of the workload
- Use db2expIn on all used SQL
  - Know the costs of the statements
- Fix top, frequently used SQL
- Analyze, realize and justify performance tuning ROI
How many access paths are required?

- Application may use many access paths in a program
  - Use efficient SQL Joins over single SQL statements
  - Application logic should do one SQL statement versus two
  - SQL Joins of two tables are faster than application logic

- SQL uses many statistics to choose access path
  - The list of parameters is only getting longer
  - Verify your statistics are current and accurate

- DB2 functions are faster than application code
  - DB2 enhancements in latest version optimize opportunities

- MQTs and MDC are optimized for DW workloads
  - Save 750 million I/Os per DAY using MQTs

Application - SQL

- Minimal Cursors should be defined in a process
  - Retrieving many rows per SQL for processing logic
    - Eliminate RID Pool Failures & CLOSE cursors

- Focus on SQL access paths executed frequently
  - Push processing into the database engine through Joins
    - Verify using STAGE 1 SQL predicates are used
    - Additional analysis of most frequently executed

- Focus on DB2 and internal SQL functions
  - Dramatic improvement by using SQL functions
  - OLAP functions provide tremendous improvement
    - RANK, CUBE, etc.. Are much more efficient than an application
Application - SQL

- Join validation through select and where column usage
  - Verify the join criteria matches index column definitions
  - Qualify data with a WHERE clause not in the application code
    - Saves I/Os, buffer pool space and overall access

- Minimize SQL/data trips across network
  - Only bring columns you need within the SQL
  - USE CASE and WHEN logic within SQL

- Always check return codes
  - Get Diagnosis clause started with Version 8

- Use block Inserts Selects etc when possible
  - DB2 uses “set” mentality

SQL Summary

- Many complex SQL efficient paths
  - Each path has its performance aspects
    - Improve the SQL access to a more efficient SQL access
    - Avoid any and all Sorts when possible
    - Use as much WHERE clause criteria as possible
    - Only reference columns needed

- Use Joins instead of single SQL statements
  - Use SQL that references many rows instead of cursors
  - How many do you know?
  - Which access path is most efficient

- All are the best access path for the SQL written
  - Given the statistics, database design and indexes available
  - Need another index definition? INCLUDE column an option?
Improve SQL Efficiency

- Drive processing into the DB2 Engine!!!!
  - DB2 is faster than any application process!!

- Use Joins or sub-queries and phrases
  - OPTIMIZE FOR 1 ROW, WITH UR or FETCH FIRST

- CASE Statements or SELECT INTO with ORDER BY
  - Table Expressions OR Common Table Expressions (CTE)

- Materialized Query Tables – MQTs
  - DW Functions – RANK, ROW_NUMBER etc.

- Multi-Row Fetch

- Use Global Temp tables

Summary
Bad Application Processing

• RID Pool Failures
• Huge number of Abends/Rollbacks
• Deadlocks stop processing
• Locking waiting on data
• Destructive Reads & Writes
• Bad System settings
• Bad Data Sharing application processing

Improve Systems Settings

• System setting are not good for processing
  – Settings should be customized and monitored
• Production end to end monitoring and reporting needs to be developed
• Separate operational from historical data
  – Versioning
  – MQTs
• WAS Optimization
  – Monitoring, JVM, GC, Logging, # of Connections
Verify DB Design for performance

• Database is designed for performance
  – Normalization of objects is reasonable
  – Many objects support the processing
  – DB Objects clustered for processing
  – DB Objects are partitioned
  – Indexes support processing
  – Relational “Set” mentality can be processed

• Separate Operational & Historical data
  – Operations are streamlined and not bottlenecked
    • Operations are not waiting or locking

Applications and SQL

• Application design for performance
  – Delayed filtering of the data
    • Optimal Index design needed
    • Manually add Predicate Transitive Closure filter
  – RID POOL failures
    • Monitor for process usage and eliminate them
    • This is more important now than ever due to DB2 10
  – Justify Expensive Access Paths
  – Leverage the Cost Savings in DB2 10 Soon
    • Reference Dave’s DB2 10 Webcast and White Paper and new DB2 Education
Verify SQL process is extremely efficient

• SQL maximizes use of DB2 engine
  – Joins, CTE and MQTs are utilized when possible
  – Advanced SQL techniques are used/encouraged
    • SQL is EXPLAINed and understood

• Processing minimizes I/O and network traffic

• Application only get data that it uses!
  – Data minimized and gets only when absolutely needed
  – Application or utility is only run when it is necessary
    • Best performing utility is one that is never needed

Questions or Comments?

Dave Beulke
Dave@DaveBeulke.com
Dave Beulke & Associates
A division of Pragmatic Solutions, Inc.
3213 Duke Street
Suite 805
Alexandria, VA 22314
703 798-3283