

## Session C02

2-6 October, 2006

Hilton Vienna

Vienna, Austria

# DB2 Performance Samples: Ten Tools for Faster Systems

IDUG® 2006

Europe

Steve Rees  
*IBM Toronto Laboratory*

Monday, October 2, 2006 • 1:30 p.m. – 2:30 p.m.

Platform: Unix, Linux and Windows

GoFurther



# Agenda

- Motivation, goals & framework
- The tools
  - Configuration analysis
  - Snapshot analysis
  - Plan analysis
  - Event monitor analysis
- Wrap up



# Goals for the presentation

- Present & explain ten sample performance diagnostic tools
  - What each tool does, and why that's good
  - Where the tool gets its data
  - The principles & techniques involved in turning raw data into crisply identified issues
  - How to use the tool
  - Sample output
  - The fine print (prerequisites, assumptions, dependencies, side-effects, limitations, etc...)
  - How it could be extended to be even better
- Some familiarity with DB2 performance diagnostics is useful here

# What do these tools do?

- Build on DB2's performance diagnostic interfaces
- Encapsulate techniques to extract, process, analyze & present performance data
  - ☑ Capture best practices from the lab & the field
  - ☑ Easy to use
  - ☑ Portable
  - ☑ Repeatable
  - ☑ Extendible
- Simplify the process of identifying problems
- Source for all tools are available at IDUG Insider  
<http://www.idug.org/user/userlogin.asp>

## A few caveats

- These are unsupported, as-is samples
  - ...to show you what monitoring data DB2 can produce
  - ...to show you how this data can be a great benefit to your system
  - ...to show you how to use DB2 monitoring interfaces
  - ...to show you how to use various DB2 technologies
- You are welcome to use these, to study them, to modify them, etc. (subject to the usual legal terms in license.txt)
- Comprehensive, robust, fully-supported, integrated performance monitoring tools are available from both IBM and from 3<sup>rd</sup> parties. These work extremely well with DB2, and implement many of the features shown in these samples.

# Agenda

- Motivation, goals & framework
- The tools
  - **Configuration analysis**
    - Snapshot analysis
    - Plan analysis
    - Event monitor analysis
- Wrap up



## Purpose of db2perf\_sanity:

- Sanity checks configuration parameters for basic causes of performance problems
- Warnings issued if
  1. Transaction log buffer < 128 pages
  2. Transaction log located under the db directory
  3. Too few page cleaners and prefetchers defined
  4. BUFFPAGE defaulted and all bufferpools are not explicitly sized
  5. Mincommit > 1
  6. Num\_poolagents < current number of connections

## Implementation:

C program using CLI & DB2 APIs

## How it works:

- Configuration data gathered from APIs, table functions and catalogs

For each parameter we're interested in

1. Extract parameter value from API / table function
2. Compare with recommend value
3. Print warning / success messages for each test



## How to use it:

We borrow error handling code, etc., from the published samples

### Preparation

1. Copy utility files & build script from `$DB2PATH/samples/cli` to the current directory

```
cp $DB2PATH/samples/cli/utilcli.* .  
cp $DB2PATH/samples/cli/bldapp .
```

- Build the program

```
bldapp db2perf_sanity
```

### Use

1. Run the program

```
db2perf_sanity <dbname>
```

2. Update configuration based on results, if necessary

For Windows, use

- COPY
  - %DB2PATH%
  - bldapp.bat
- etc.

## Sample output

Output from running the tool against the sample database

Running sanity tests on configuration for database SAMPLE

LOGBUFSZ:

Warning:

The log buffer size (logbufsz) is currently 8.

Recommendation:

The generally recommended size is 128 or greater.

Log path:

Warning:

The transaction log is currently located in '/home/srees/srees/NODE0000/SQL00003/SQLLOGDIR/', which seems to be under the database path '/home/srees'.

Recommendation:

In general, the transaction log should be located on its own device(s) if possible.

NUM\_IOCLEANERS:

Passed

NUM\_IOSERVERS:

Passed

BUFFPAGE & NPAGES:

Warning:

BUFFPAGE seems to be left at the default value of 1000, but the following bufferpools still have NPAGES set to either -1 or to 1000, so they still have the default size

IBMDEFAULTBP

Recommendation:

Use ALTER BUFFERPOOL to set NPAGES to the desired value for all bufferpools.

MINCOMMIT:

Passed

NUM\_POOLAGENTS:

Passed

## Notes:

- Based on rules-of-thumb
  - Not all messages applicable in all cases
- Path analysis is 'quick & dirty' - may not be 100% accurate
- Uses C because of need to get to configuration APIs
- Tool depends on DB2 v8.2 FP9 for table functions

# Agenda

- Motivation, goals & framework
- The tools
  - Configuration analysis
  - **Snapshot analysis**
  - Plan analysis
  - Event monitor analysis
- Wrap up



## Purpose of db2perf\_utils:

- Provides a variety of 'helper functions' that make life a little easier for us
  1. Translation from codes used in table functions to human-readable form
    - Statement Operations
    - Statement types
    - Lock types
  2. 'Quiet' SQL **drop** function :-)
    - Suppresses 'not found' errors.
    - Useful in CLP scripts when doing proactive cleanup (i.e. dropping objects that aren't there yet ...)

## Implementation:

SQL/PL UDFs

## How to use it:

### Preparation

1. Connect to the desired database

```
db2 connect to <dbname>
```

2. Create the stored procedures

```
db2 -td@ -f db2perf_utils.db2
```

GoFurther

## How to use it, cont'd:

### Use

#### 1. Translation UDFs

- Simply call them in SQL to translate fields returned from snapshot table functions & event monitors

```
db2 "select db2perf_<UDF>2str(<element value>) from ..."
```

Table function(s)	Element(s)	Translation UDF
SNAPSHOT_LOCK SNAPSHOT_LOCKWAIT	Lock object type	db2perf_ikobj2str
	Lock mode Lock mode requested	db2perf_ikmode2str
	Lock status	db2perf_ikstat2str
SNAPSHOT_TABLE	Table type	db2perf_tabtyp2str
SNAPSHOT_APPL_INFO	Application status	db2perf_apstat2str
SNAPSHOT_TBS_CFG	Tablespace type	db2perf_tbstyp2str
	Tablespace contents	db2perf_tbscon2str

## How to use it, cont'd:

Use

### 2. 'Quiet drop' function

```
db2 "call db2perf_quiet_drop( <suffix of DROP statement> )"
```

for example

```
db2 "call db2perf_quiet_drop('procedure db2perf_crmsg')"
```

GoFurther



## How it works:

```
CREATE FUNCTION db2perf_tbstyp2str(tablespace_type bigint)
:
BEGIN ATOMIC
  DECLARE retstr CHAR(3);
  :
  SET retstr = CASE tablespace_type
                WHEN 0 THEN 'DMS'
                WHEN 1 THEN 'SMS'
                ELSE NULL
              END;
  RETURN retstr;
END@
```

Values & strings  
extracted from  
sqlmon.h

```
CREATE PROCEDURE db2perf_quiet_drop( IN statement VARCHAR(1000) )
LANGUAGE SQL
BEGIN
  DECLARE SQLSTATE CHAR(5);
  DECLARE NotThere CONDITION FOR SQLSTATE '42704';

  DECLARE EXIT HANDLER FOR NotThere
    SET SQLSTATE = '      ';

  SET statement = 'DROP ' || statement;
  EXECUTE IMMEDIATE statement;
END@
```

Catch & overwrite  
'not found'  
SQLSTATE

## Sample output

```
db2 "select substr(tablespace_name,1,20) as 'Name',
      tablespace_type, db2perf_tbstyp2str(tablespace_type),
      tbs_contents_type, db2perf_tbscon2str(tbs_contents_type)
from table(snapshot_tbs_cfg(cast(null as varchar(256)),-1)) as t"
```

Name	TABLESPACE_TYPE 3	TBS_CONTENTS_TYPE 5
SYSCATSPACE	1 SMS	0 Any
SYSTOOLSPACE	1 SMS	0 Any
USERSPACE1	1 SMS	0 Any
TBS_ALL	0 DMS	0 Any
TEMPSPACE	1 SMS	2 System temporary

5 record(s) selected.

Untranslated values  
provided by table function

Translated values  
provided by the UDF

```
$ db2 "drop table blork"
SQL0204N  "SREES.BLORK" is an undefined name.  SQLSTATE=42704
$ db2 "call db2perf_quiet_drop('table blork')"
```

Return Status = 0

```
$ db2 "create table blork ..."
```

Table does not exist  
but quiet\_drop  
suppresses the error

## Purpose of db2perf\_bufferpool:

- Identifies & quantifies bufferpool-related performance issues
- Reports at overall level and by bufferpool
- Issues warnings if
  1. Bufferpool data or index hit ratios are below threshold
  2. Data or index prefetch ratio below threshold
  3. Page clean ratio below threshold
  4. Number of dirty steals above threshold
  5. Number of files closed above threshold

## Implementation:

SQL/PL stored procedures

## How to use it:

### Preparation

1. Connect to the desired database

```
db2 connect to <dbname>
```

2. Create stored procedures

```
db2 -td@ -f db2perf_utils.db2
```

```
db2 -td@ -f db2perf_bp.db2
```

3. Turn on bufferpool monitoring by default with  
**DFT\_MON\_BUFPOOL** dbm config switch

### Use

1. Connect to the desired database

```
db2 connect to <dbname>
```

2. Call stored procedure & examine results

```
db2 "call db2perf_bufferpool()"
```

3. Update configuration based on results, if necessary

## How it works:

- Snapshot data gathered from **snapshot\_database** and **snapshot\_bufferpool** table functions
- Warnings with severity levels 1-5 inserted into **db2perf\_msg** table

Metric	Formula	Activity Threshold	Severity		
			5 (worst)	3	1
Data Hit Ratio	$(LR - PR) / LR$	> 1000 LReads	< 60%	< 75%	< 90%
Index Hit Ratio	$(LR - PR) / LR$	> 1000 LReads	< 75%	< 85%	< 95%
Cleaning	Async Writes / Total Writes	> 1000 page writes	< 40%	< 65%	< 90%
Prefetch	Async data reads / Data phys reads	> 1000 async data reads	< 50%	< 70%	< 90%
Dirty page steals	Dirty Steals / 10,000 Tx	> 100 transactions	> 100	> 30	> 1
File Closes	Files Closed / 10,000 Tx	> 100 transactions	> 1000	> 100	> 10

## Sample output

```
$ db2 "call db2perf_bufferpool()"
```

```
Result set 1
```

```
-----
```

TS	SEVERITY	METRIC	VALUE	COMMENTS
2006...	3	Dirty Page Steals / 10k Tx	27	
2006...	3	Overall BP page clean ratio	41.6	
2006...	1	IBMDEFAULTBP data hit ratio	82.1	
2006...	1	Overall BP data hit ratio	82.1	
2006...	0	Overall BP index hit ratio	98.8	
2006...	0	Files closed / 10k Tx	0	
2006...	0	IBMDEFAULTBP idx hit ratio	98.8	
2006...		Overall BP data prefetch ratio	00.0	No data prefetching activity
2006...		Overall BP index prefetch ratio	00.0	No index prefetching activity
2006...		IBMDEFAULTBP data pftch ratio	00.0	No data prefetching activity
2006...		IBMDEFAULTBP index pftch ratio	00.0	No index prefetching activity

Lots of dirty page steals - look at bufferpool size if possible, and/or page cleaning parameters

```
11 record(s) selected.
```

## Notes:

- Thresholds & severity levels are easily tuned to support different environments
- Depends on DB2 v8.2.2 for table functions
- Creates SQL/PL stored procedure and **db2perf\_msg** message table in default schema
  - Returns a result set with the most recent rows added to the message table
  - Leaves the message table in place after execution
  - Messages remain in message table by default
    - Use **ORDER BY ts DESC** on **SELECT** to see most recent messages first

## Purpose of db2perf\_dynsql:

- Identifies & quantifies dynamic SQL-related performance issues
- Calls out groups of 'top 10' statements by:
  1. Total elapsed time
  2. Total CPU usage
  3. Most physical reads
  4. Most rows read
  5. Sorts
  6. Sort overflows
- Identifies statements that might benefit from parameter markers instead of literals

## Implementation:

CLP scripts + UDF in C



## How to use it:

### Preparation

1. Create the **db2perf\_quiet\_drop** utility stored procedure

```
db2 -td@ -f db2perf_utils.db2
```

2. Build & define the C user-defined function

```
cp $DB2PATH/samples/c/bldrtn . # use bldrtn script from
                                # DB2 samples
bldrtn db2perf_udf              # compile & copy UDF under
                                # sqllib
db2 connect to <dbname>
db2 -tvf db2perf_setupudf.db2   # CREATE FUNCTION for UDF
```

### Use

1. Connect to the desired database

```
db2 connect to <dbname>
```

2. Run the CLP script, sending the output to a file

```
db2 -tf db2perf_dynsql.db2 -r db2perf_dynsql.out
```

## How it works:

1. Grabs snapshot data from `snapshot_dynsql` table function into a scratch table `db2perf_dynsql`
2. Adds columns to `db2perf_dynsql` to store rank within each metric
3. Queries snapshot table with ORDER BY & FETCH FIRST to find top 10 statements in each metric

```
SELECT
    substr(char(row_num),1,2) as "#","Executions","Rows read", "% of Total",
    "r/r / 100","Statement"
FROM
    OLD TABLE
    ( UPDATE
        ( SELECT
            CAST(num_executions as INTEGER) as "Executions",
            CAST(rows_read as INTEGER) as "Rows read",
            CAST(pct_of_total_rows_read as SMALLINT) as "% of Total",
            100 * CAST(round(CAST(rows_read as FLOAT) /
                (num_executions+1),0) as INTEGER) as "r/r / 100",
            top10_rows_read,
            row_number() over (ORDER BY (rows_read) DESC) as row_num,
            substr( stmt_text,1,80 ) as "Statement"
        FROM db2perf_dynsql
        WHERE rows_read > 0
        ORDER BY "Rows read" DESC
        FETCH FIRST 10 ROWS ONLY )
    SET top10_rows_read = char(row_num) );
```

Finds top 10  
and for each one, uses  
SELECT from UPDATE  
&  
UPDATE through SELECT  
to record the rank within  
that list back in our  
snapshot table

## How it works, cont'd:

4. Pulls out all the 'Top 10' statements from our work table – chances are that some of them are Top 10 for more than one metric. Look at those first ...
5. For all statements that don't contain a parameter marker ('?'), calls the UDF **db2perf\_RmLiterals** to replace numeric and character literals with parameter markers. Counts how many duplicates this makes – i.e., how many statements with literals could be replaced with a single statement using parameters markers instead.

```
UPDATE db2perf_dynsql
  SET compressed_statement = db2perf_RmLiterals( translate( CAST( stmt_text as varchar(3000) ) ) )
 WHERE length(stmt_text) < 3000
    AND CAST( stmt_text as varchar(3000) ) NOT LIKE '%?%';

SELECT
  count(*) as "Count" , substr(compressed_statement,1,120) as "Statement without literals"
FROM db2perf_dynsql
WHERE substr( translate( ltrim( compressed_statement ) ),1,6 ) IN ( 'SELECT', 'INSERT' )
   AND num_executions = 1
GROUP BY substr( compressed_statement,1,120 )
HAVING count(*) > 10;
```

Only report cases where 10 or more statements could potentially be replaced by 1

## Sample output

### Top 10 dynamic SQL statements by execution time

#	Executions	Exec Time	% of Total	sec / 100	Statement
1	11712	478.286	20	4.083	Select D_NEXT_O_ID, D_TAX from DIST ...
2	117041	328.792	13	0.280	Insert into ORDER_LINE values (?, ? ...
:					
9	10800	36.042	1	0.333	Select MIN(NO_O_ID) from NEW_ORDER ...
10	11311	35.859	1	0.317	Select C_LAST, C_CREDIT, C_DISCOUNT ...

Heavy  
hitter!  
#1 in all 3  
of CPU use,  
physical  
reads &  
rows read

### Combined ranking of top dynamic SQL statements

Rank elapsed	Rank CPU	Rank phys rd	Rank R/R	Rank sorts	Rank sort ovf	Statement
10	10	4		6		Select C_LAST, C_CREDIT, ...
		9				Select O_OL_CNT, O_ID, ...
4	1	1		1		Select S_QUANTITY, S_DIST_01, ...
:						
6	5	2				Select Count(Distinct S_I_ID) ...
7	2			2		Update STOCK set S_QUANTITY = ?, ...

### List of dynamic SQL statements which differ only by literal values (Good candidates for parameter markers)

Count	Statement without literals
693	SELECT C_ID, C_FIRST FROM CUSTOMER WHERE (C_W_ID = ? AND C_D_ID = ? AND C_LAST = ?) ...

We can possibly replace 693 SQL  
PREPAREs with just one  
statement that uses parameter  
markers

## Notes:

- New 'Top 10' summaries are easily added
- 'Top 10's are easily changed to 'Top 20's, etc.
- UDF in C to remove literals far more natural than doing it in SQL!
- SELECT from UPDATE & UPDATE through SELECT made storing & merging the various Top 10 rankings very simple
- Script drops work table `db2perf_dynsql` at end
- Depends on DB2 v8.2 FP9 for table functions

## Purpose of db2perf\_locktree:

- Provides a (crude) graphical 'tree' view of in-flight lock wait relationships between DB2 connections, based on the `snapshot_lockwait` table function
- Helps visualize the locking dependencies between applications

## Implementation:

Recursive SQL/PL stored procedure

## How to use it:

### Preparation

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Create the `db2perf_quiet_drop` utility stored procedure  
`db2 -td@ -f db2perf_utils.db2`
3. Create the `db2perf_locktree` stored procedure  
`db2 -td@ -f db2perf_locktree.db2`

### Use

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Call the stored procedure to capture the state of lock wait relationships at that moment  
`$ db2 "call db2perf_locktree()"`
3. Examine the lock relationships in the result set returned from `db2perf_locktree`

## How it works:

1. Grabs lock wait snapshot data from `snapshot_lockwait` table function into a scratch table `db2perf_lockwait`
2. Finds lock waits at the 'root' – where the lock holder is not waiting on another lock. Starts with these as the roots of our trees
3. For each instance of lock wait
  - a) Draw a line to it from its 'parent' lock wait, if one exists (ie, if the owner of the lock we want is waiting on someone else...)
  - b) Writes details about this lock to our 'report' table `db2perf_locktree`
    - holder / waiter application ids
    - lock type
    - lock wait time, etc.
  - b) Recursively calls `db2perf_locktree` for each of the applications waiting on the 'parent'
4. Opens a cursor to return a result set with the lock tree

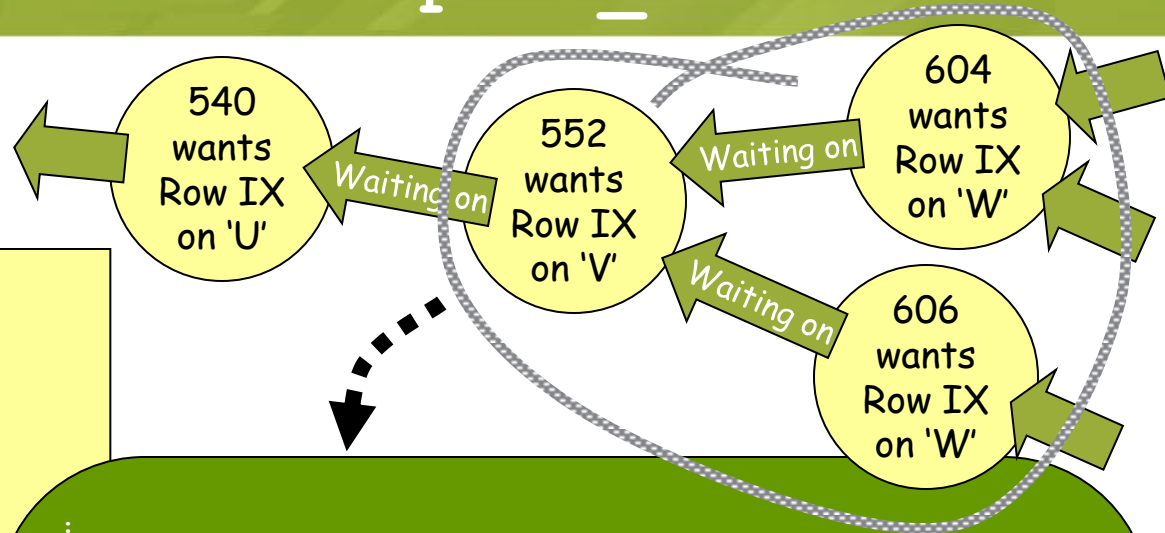


## Sample output

```

Waiter appl handle: 547 (getlock)
| Holder appl handle: 584 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 231451
| Lock escalation: N
| Table name: SREES.T
+-----
Waiter appl handle: 540 (getlock)
| Holder appl handle: 547 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 229446
| Lock escalation: N
| Table name: SREES.U
+-----
Waiter appl handle: 552 (getlock)
| Holder appl handle: 540 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 215371
| Lock escalation: N
| Table name: SREES.V
+-----
Waiter appl handle: 604 (getlock)
| Holder appl handle: 552 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 44941
| Lock escalation: N
| Table name: SREES.W
+-----
Waiter appl handle: 606 (getlock)
| Holder appl handle: 552 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 46951
| Lock escalation: N
| Table name: SREES.W
+-----
Waiter appl handle: 556 (getlock)
| Holder appl handle: 552 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 157304
| Lock escalation: N
| Table name: SREES.W
+-----
Waiter appl handle: 562 (getlock)
| Holder appl handle: 552 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 209336
| Lock escalation: N
| Table name: SREES.W
+-----
Waiter appl handle: 566 (getlock)
| Holder appl handle: 562 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 143217
| Lock escalation: N
| Table name: SREES.X
+-----
Waiter appl handle: 598 (getlock)
| Holder appl handle: 566 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 141204
| Lock escalation: N
| Table name: SREES.Y
+-----
Waiter appl handle: 602 (getlock)
| Holder appl handle: 598 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 109058
| Lock escalation: N
| Table name: SREES.Z

```



```

Lock escalation: N
Table name: SREES.U

```

```

-----Waiter appl handle: 552 (getlock)
| Holder appl handle: 540 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 215371
| Lock escalation: N
| Table name: SREES.V

```

```

-----Waiter appl handle: 604 (getlock)
| Holder appl handle: 552 (getlock)
| Lock object type: Row
| Lock mode requested: Intention Exclusive Lock
| Lock wait time (ms): 44941
| Lock escalation: N
| Table name: SREES.W

```

```

-----Waiter appl handle: 606 (getlock)
| Holder appl handle: 552 (getlock)

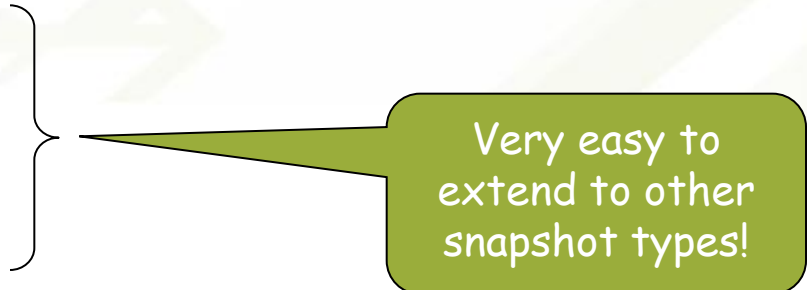
```

Application name

## Notes:

- Recursive SQL/PL calls are an excellent choice here
  - Maximum SQL/PL nesting depth currently caps length of lockwait chains we can display at 16
- Like lock snapshot, captures instantaneous picture when it's run, not cumulative
- Creates tables in the current schema
  - scratch tables `db2perf_lockwait`, `db2perf_appl_info`
  - report table `db2perf_locktree`
- Note - use **'order by line'** if selecting from `db2perf_locktree`
- Report is overwritten by each run
- Content of scratch tables are deleted at end of run

## Purpose of db2perf\_snapdiff:

- Collects snapshot data into DB2 tables
  - Compares data from 'before' & 'after' intervals
    - One 'interval' = the change between two snapshots
  - Produces a report in table `db2perf_snapdiff_report`
  - Supports:
    - Normalization of results to overall system activity
    - Thresholds (i.e., only report differences over X%)
  - Currently handles the following snapshot types
    1. Database Manager
    2. Database
    3. Tablespace
    4. Tables
    5. Bufferpool
- 

## Implementation:

SQL/PL stored procedures

## How to use it:

### Preparation

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Create stored procedures and snapshot storage tables  
`db2 -td@ -f db2perf_utils.db2`  
`db2 -td@ -f db2perf_snapdiff.db2`
3. Turn on monitoring by default with DFT\_MON\_XXX database manager configuration switches

## How to use it - the basics:

### Use

1. Connect to the desired database

`db2 connect to <dbname>`

2. Call stored procedure – the easy way, with one parameter

`db2 "call db2perf_snapdiff(<operation>)"`

Prints usage  
syntax if no  
operation is  
passed in

- 'start'** - collect 'start of interval' snapshot from table function and store it in one of our tables
- 'stop'** - collect 'end of interval' snapshot and store it in snapshot storage table
- 'diff'** - compare two rows in the snapshot tables and report what's different
- 'list'** - show what snapshot interval data has been collected
- 'delete'** - delete all snapshot interval data from storage tables

## How to use it - the basics:

Typical sequence of operations:

1. Get the first interval of data

```
db2 "call db2perf_snapdiff('start')"  
sleep(30)  
db2 "call db2perf_snapdiff('stop')"
```

2. Sometime later, get another interval of data

```
db2 "call db2perf_snapdiff('start')"  
sleep(30)  
db2 "call db2perf_snapdiff('stop')"
```

3. List the intervals we've got

```
db2 "call db2perf_snapdiff('list')"
```

4. Compare the latest 2 intervals

```
db2 "call db2perf_snapdiff('diff')"
```

## How to use it - more advanced...

```
db2 "call db2perf_snapdiff(  
    <operation>                , <snap_table_name> ,  
    <'before' interval #>      , <'after' interval #> ,  
    <normalize to Tx>          , <threshold_pct>    )" 
```

### <snap\_table\_name>

- chooses one snapshot storage table to act on (defaults to all)

### <'before' interval>, <'after' interval>

- number of 'before' & 'after' snapshot intervals to compare  
(default to the two most recent intervals)

### <normalize>

- 'Y', 'T', '1' means to normalize all data by the number  
of transactions executed during the snapshot period (defaults to 'Y')

### <threshold\_pct>

- 'clip level' below which we don't report differences  
(defaults to 5%)

```
db2 "call db2perf_snapdiff('diff' ,  
    NULL, 11, 10, 'Y', 5)"
```

NULL or '' here  
means compare all  
snapshot tables

Compare data from intervals 10 & 11

Normalize  
results to  
number of  
transactions,  
and don't  
show any  
differences  
smaller than  
5%



## How it works - collecting data:

- Our tables contain rows saved from snapshot table functions when 'start' & 'stop' are called

The 'TOC' table maps interval numbers to start/stop timestamps

db2perf\_snapdiff\_toc

INTERVAL	START	STOP	DBM	DB	TBS	TB	BP
1	2006-02-27-00.18.04.259134	2006-02-27-00.18.15.695562	Y	Y	Y	Y	Y
2	2006-03-02-22.02.19.480432	2006-03-02-22.02.50.158643	Y	Y	Y	Y	Y

We create our first interval by getting two snapshots:

call db2perf\_snapdiff('start')  
... wait a while ...  
call db2perf\_snapdiff('stop')

db2perf\_snapdbm

SNAPSHOT_TIMESTAMP	SORT_HEAP_ALLOCATED	...
2006-02-27-00.18.04.259134	1000	...
2006-02-27-00.18.15.695562	1000	...
2006-03-02-22.02.19.480432	1000	...
2006-03-02-22.02.50.158643	1000	...

db2perf\_snapdb

SNAPSHOT_TIMESTAMP	ROWS_READ	...
2006-02-27-00.18.04.259134	504265	...
2006-02-27-00.18.15.695562	836199	...
2006-03-02-22.02.19.480432	4253835	...
2006-03-02-22.02.50.158643	4627251	...

:  
db2perf\_snaptb

SNAPSHOT_TIMESTAMP	ROWS_WRITTEN	...	TABLE_NAME	...
2006-02-27-00.18.04.259134	12460	...	DISTRICT	...
2006-02-27-00.18.04.259134	124600	...	STOCK	...
2006-02-27-00.18.15.695562	20574	...	DISTRICT	...
2006-02-27-00.18.15.695562	205900	...	STOCK	...
2006-03-02-22.02.19.480432	104881	...	DISTRICT	...
2006-03-02-22.02.19.480432	1048906	...	STOCK	...
2006-03-02-22.02.50.158643	114106	...	DISTRICT	...
2006-03-02-22.02.50.158643	1140702	...	STOCK	...

The sometime later, when we want to compare DB2 activity with the first interval, we create our second interval by getting two more snapshots:

call db2perf\_snapdiff('start')  
... wait a while ...  
call db2perf\_snapdiff('stop')



We're finding 4 times: start & stop for each of interval 1 & 2

## How it works - comparing:

1. Finds the start/stop times for the intervals to be compared from the TOC
2. For each snapshot table to be compared
  - a. Finds the pairs of rows from this table with:
    - Timestamps matching the 'before' & 'after' interval times
    - Matching 'key column' values (if applicable)  
For example, rows with the same table name, the same bufferpool name, etc.
  - b) For each numeric column in the rows
    - i. Finds the normalized activity in intervals 1 & 2

$\frac{(\text{Interval 1 'stop'}) - (\text{Interval 1 'start'})}{(\text{\# transactions in Interval 1} / 1000)}$	$\frac{(\text{Interval 2 'stop'}) - (\text{Interval 2 'start'})}{(\text{\# transactions in Interval 2} / 1000)}$
--	--
    - ii. Calculates the difference between the normalized values for interval 1 & 2 for this column
    - iii. If the change between intervals is greater than the threshold  
Write the column name, interval values & difference to the report

“Confusing”, you say? Ok, in pictures ...

db2 call db2perf\_snapdiff('diff')

db2perf\_snapdiff\_toc

INTERVAL	START	STOP	DBM	DB	TBS	TB	BP
1	2006-02-27-00.18.04.259134	2006-02-27-00.18.15.695562	Y	Y	Y	Y	Y
2	2006-03-02-22.02.19.480432	2006-03-02-22.02.50.158643	Y	Y	Y	Y	Y

All these changes happen to be below the default threshold of 5%

db2perf\_snapdbm

SNAPSHOT_TIMESTAMP	POST_THRESHOLD_SORTS
2006-02-27-00.18.04.259134	0
2006-02-27-00.18.15.695562	0
2006-03-02-22.02.19.480432	0
2006-03-02-22.02.50.158643	0

Interval 1 post thersh sorts / 1k Tx  

$$\frac{0 - 0}{(45,654 - 27,657) / 1000} = 0$$

0 - 0 =  
0 change  
between  
Intervals 1 & 2

Interval 2 post thresh sorts / 1k Tx  

$$\frac{0 - 0}{(253,236 - 232,945) / 1000} = 0$$

db2perf\_snapdb

SNAPSHOT_TIMESTAMP	ROWS_READ	COMMITTS
2006-02-27-00.18.04.259134	504265	27657
2006-02-27-00.18.15.695562	836199	45654
2006-03-02-22.02.19.480432	4253835	232945
2006-03-02-22.02.50.158643	4627251	253236

Interval 1 rows read / 1k Tx  

$$\frac{(836,199 - 504,265)}{(45,654 - 27,657) / 1000} = 18,443$$

18,403 - 18,443 =  
-40 rows read / 1k Tx  
decrease between  
Intervals 1 & 2  
(-0.2%)

Interval 2 rows read / 1k Tx  

$$\frac{(4,627,251 - 4,253,835)}{(253,236 - 232,945) / 1000} = 18,403$$

db2perf\_snaptb

SNAPSHOT_TIMESTAMP	ROWS_WRITTEN	TABLE_NAME
2006-02-27-00.18.04.259134	12460	DISTRICT
2006-02-27-00.18.04.259134	124600	STOCK
2006-02-27-00.18.15.695562	20574	DISTRICT
2006-02-27-00.18.15.695562	205900	STOCK
2006-03-02-22.02.19.480432	104881	DISTRICT
2006-03-02-22.02.19.480432	1048906	STOCK
2006-03-02-22.02.50.158643	114106	DISTRICT
2006-03-02-22.02.50.158643	1140702	STOCK

Int. 1 DISTRICT rows written / 1k Tx  

$$\frac{(20,574 - 12,460)}{(45,654 - 27,657) / 1000} = 451$$

**DISTRICT:**  
455 - 451 =  
4 rows written  
per 1k Tx  
increase between  
intervals 1 & 2  
(0.8%)

Int. 1 STOCK rows written / 1k Tx  

$$\frac{(205,900 - 124,600)}{(45,654 - 27,657) / 1000} = 4,517$$

**STOCK:**  
4,524 - 4,517 =  
7 rows written  
per 1k Tx  
increase between  
intervals 1 & 2  
(0.1%)

Int. 2 DISTRICT rows written / 1k Tx  

$$\frac{(114,106 - 104,881)}{(253,236 - 232,945) / 1000} = 455$$

Int. 2 STOCK rows written / 1k Tx  

$$\frac{(1,140,702 - 1,048,906)}{(253,236 - 232,945) / 1000} = 4,524$$

2006...259134  
2006...695562  
2006...480432  
2006...158643

## Sample output

Here we happen to be comparing intervals before & after an index was dropped ...

```
$ db2 "call db2perf_snapdiff('diff')"
```

Result set 1

-----  
MESSAGE

db2perf\_snapdiff called at 2006-03-04-22.47.50.316251

```
operation:..... diff
snap_table_name:.....
'before' interval:..... -1
'after' interval:..... -1
normalize to 1k Tx:..... Y
threshold %:..... 5
```

:

Database Snapshot (db2perf\_snapdb)

\*\*\* for > 100% difference  
\*\* for > 33,  
\* for > 10

... many fewer transactions  
... many, many more rows read,  
logical data reads, physical  
data reads  
... many fewer index reads  
etc., etc., etc.

	03/02/2006 22:42:07 to 03/02/2006 22:43:08 -----	03/02/2006 22:50:49 to 03/02/2006 22:51:35 -----	
Normalizing to 1K Tx per Interval	40.9	0.1	-99.5 %
*** ROWS_READ	18228.0	41206692.7	2260516.9 %
*** POOL_DATA_L_READS	27898.6	10196246.9	36447.4 %
*** POOL_DATA_P_READS	1652.4	15433.7	833.9 %
** POOL_DATA_WRITES	1665.8	0.0	-100.0 %
* POOL_INDEX_L_READS	108994.1	73801.2	-32.2 %
* POOL_INDEX_P_READS	96.9	114.4	18.0 %
** POOL_INDEX_WRITES	262.7	0.0	-100.0 %
** POOL_READ_TIME	3025.5	4753.0	57.0 %
** POOL_WRITE_TIME	14882.7	0.0	-100.0 %

## Sample output cont'd

```

:
Table Snapshot (db2perf_snaptb) -----
    Table DISTRICT
                                03/02/2006      03/02/2006
                                22:42:07      22:50:49
                                to
                                03/02/2006      03/02/2006
                                22:43:08      22:51:35
                                -----
    Normalizing to 1K Tx per Interval
*   ROWS_READ                    40.9          0.1      -99.5 %
                                1370.9         1180.7     -13.8 %

    Table HISTORY
                                03/02/2006      03/02/2006
                                22:42:07      22:50:49
                                to
                                03/02/2006      03/02/2006
                                22:43:08      22:51:35
                                -----
    Normalizing to 1K Tx per Interval
*   ROWS_WRITTEN                 40.9          0.1      -99.5 %
                                428.8         295.1     -31.1 %

:
Buffer Pool Snapshot (db2perf_snapbp) -----
    Bufferpool IBMDEFAULTBP
                                03/02/2006      03/02/2006
                                22:42:07      22:50:49
                                to
                                03/02/2006      03/02/2006
                                22:43:08      22:51:35
                                -----
    Normalizing to K Tx per Interval
*** POOL_DATA_L_READS           40.9          0.1      -99.5 %
                                27893.9       10196289.1  36453.8 %
*** POOL_DATA_P_READS           1651.9       15433.7    834.2 %
**  POOL_DATA_WRITES            1663.9          0.0    -100.0 %

:

```

## Notes:

- Easily extended to compare numeric values in other snapshot tables
  - ...or pretty well any table with a timestamp column!
  - Table definitions are derived 'on the fly', not built in
- Some snapshot fields don't make sense to compare, e.g. instantaneous values like 'lock list used', etc.
  - Snapdiff supports (hard-coded) 'ignore lists' to overlook columns we're not interested in
- Depends on DB2 v8.2.2 for snapshot table functions

# Agenda

- Motivation, goals & framework
- The tools
  - Configuration analysis
  - Snapshot analysis
  - **Plan analysis**
  - Event monitor analysis
- Wrap up



## Purpose of db2perf\_plandiff:

- Examines the contents of the explain tables to identify plan changes
- Saves time in combing through db2exfmt output, looking for non-trivial changes
- Useful for 'bulk comparing' plans across migrations, system changes, etc.

## Implementation:

### Nested SQL/PL stored procedures

Note – db2perf\_plandiff in v1.1 of the toolbox has been updated to support comparisons across two sets of explain tables. The syntax described here is still supported, but refer to db2perf\_plandiff\HowTo.txt for more options.

-- Steve Rees, June 2009

## How to use it:

### Preparation

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Create the `db2perf_quiet_drop` utility stored procedure  
`db2 -td@ -f db2perf_utils.db2`
3. Create the `db2perf_plandiff` stored procedures  
`db2 -td@ -f db2perf_plandiff.db2`
4. Ensure the explain tables exist and are populated

### Use

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Call the stored procedure to compare all plans with matching SQL & matching patterns of requester, schema, source name & section  
`db2 "call db2perf_plandiff(  
    <requester>, <schema>, <source_name>, <section>)"`



## How to use it, cont'd:

For example

```
db2 "call db2perf_plandiff('SREES', 'SREES', 'FOO%', 0)"
```

compares all pairs of plans in the explain tables where

- Original statement texts match
- Requester and schema are 'SREES'
- Source name (i.e. package name) starts with 'FOO'
- Section number is anything (0 is wildcard, as in db2exfmt)

and displays the results

3. If differences are reported, use the contents of **db2perf\_plandiff\_report** to determine which statements to examine with **db2exfmt**

## How it works:

1. Opens a cursor C1 on **EXPLAIN\_STATEMENT** to find rows matching the patterns provided
2. Opens another cursor C2 on **EXPLAIN\_STATEMENT** to find all other rows that
  - a) Match the patterns, and
  - b) Match the statement text found in C1
3. For each pair of matching statements
  - a) Generates a 'signature string' for each from the operators & operands in the explain tables for that statement

db2exfmt version

Access Plan:

-----

Total Cost: 25.7601

Query Degree: 1

Rows  
RETURN  
( 1)  
Cost  
I/O  
|  
1

■ GRPBY  
■ ( 2)  
■ 25.7593  
■ 1.96576

|  
900.254

■ IXSCAN  
■ ( 3)  
■ 69.031  
■ 5.268

|  
450127

INDEX: SREES

NU\_ORD\_IDX1

db2perf\_plandiff version

```
$ db2 "call db2perf_planstring('SREES','2006-02-12-15.46.03.296586','SREES','DELS',1,')'"
Value of output parameters
```

-----

Parameter Name : PLAN\_STRING

Parameter Value : RETURN(1)<-Op(2) GRPBY(2)<-Op(3) IXSCAN(3)<-Object(NU\_ORD\_IDX1)

## How it works, cont'd:

- b) Compares the signature strings of the two plans
- c) Writes the comparison result to `db2perf_plandiff` along with

- SQL statement text

- Schema name

- Package name

- Section number

- Timestamp

- Estimated cost in timerons

} for both statements  
being compared

- 4. Opens & return a result set cursor with the plan comparison results

## Sample output

- Looking for plan changes before & after a slowdown in our system

Plan Change?	Package Name	Section	Timestamp	Cost (timerons)	Statement Text
Yes	SREES.DELS	1	2006-...296586	25	
	SREES.DELS	1	2006-...132561	28254	SELECT MIN( no_o_id ) INTO :H00009 :H00010 FROM new_order WHERE no_w_id=:H00001 AND no_d_id = :H00008 WITH RR USE AND KEEP EX CLUSIVE LOCKS
Cost of SELECT & DELETE on NEW_ORDER has skyrocketed - check out plans for these statements in db2exfmt					
Yes	SREES.DELS	2	2006-...296586	38	
	SREES.DELS	2	2006-...132561	28251	DELETE FROM new_order WHERE no_w_id = :H00001 AND no_d_id = :H00008 AND no_o_id = :H00009
No	SREES.DELS	3	2006-...296586	51	
	SREES.DELS	3	2006-...132561	51	UPDATE orders SET o_carrier_id = :H00002 WHERE o_id = :H00009 AND o_w_id = :H00001 AND o_d_id = :H00008
No apparent change in plan for UPDATE on ORDERS					
Report run at 2006-02-12-16.46.20.984718 Compared 90 plan pairs (4 look different, 86 look unchanged). Unable to compare 0 plan(s) due to length or complexity.					

- Based on this data, it's worthwhile digging into db2exfmt output, especially for the DELETE & SELECT statements

## Notes:

- Assumes explain tables exist in current default schema and are already populated
- Compares statements up to 30,000 characters in length
  - Warns when statements are found that are too long / complex to be compared
- Currently reports all plan comparison results – match or non-match
- Tip - use more restrictive patterns to reduce scope & improve runtime
  - E.g. 'PROD%' instead of just '%'
  - Highly populated explain tables (especially with many versions of the same statements) could cause long runtimes for this tool

## Purpose of db2perf\_plans:

- Mines the explain tables for useful information about SQL execution plans
  - Most expensive statements
    - By total cost
    - By I/O cost
  - Unreferenced indexes

## Implementation:

SQL/PL stored procedure

## How to use it:

### Preparation

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Create the utility stored procedures  
`db2 -td@ -f db2perf_utils.db2`
3. Create the **db2perf\_plans** stored procedure  
`db2 -td@ -f db2perf_plans.db2`
4. Ensure the explain tables exist and are populated

### Use

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Call the stored procedure  
`db2 "call db2perf_plans()"`

## How it works:

1. Selects the 10 statements from **EXPLAIN\_STATEMENT** with the greatest values for **TOTAL\_COST**
  - Write cost & SQL statement to **db2perf\_plans\_report**
2. Selects the 10 **RETURN** operators from **EXPLAIN\_OPERATOR** with the greatest **IO\_COST**
  - Join these with **EXPLAIN\_STATEMENT** to get the SQL text
  - Write cost & SQL statement to **db2perf\_plans\_report**
3. For each table referenced in **EXPLAIN\_OBJECT**
  - Find all indexes on that table from **SYSCAT.INDEXES**
    - If an index is not found in **EXPLAIN\_OBJECT**, write a message to **db2perf\_plans\_report**

RETURN is the top operator in the plan; its IO cost represents the whole plan



## Sample output

```
$ db2 "call db2perf_plans()"
```

```
Result set 1
```

```
-----
```

### Top 10 most expensive statements - total cost

Rank	Cost	Source	Section
-----			
1	30035	SREES.NEWS	6
		SELECT i_price, i_name, i_data INTO :H00056 , :H00055 , :H00043 FROM item WHERE i_id = :H00049	
2	7644	SREES.STKS	2
		SELECT count(distinct S_I_ID) INTO :H00006 FROM ORDER_LINE, STOCK WHERE OL_W_ID = :H00001 AND OL_D_ID = :H00002 AND OL_O_ID < :	

```
:
```

### Top 10 most expensive statements - I/O cost

Rank	Cost	Source	Section
-----			
1	2322	SREES.NEWS	6
		SELECT i_price, i_name, i_data INTO :H00056 , :H00055 , :H00043 FROM item WHERE i_id = :H00049	

```
:
```

## Sample output, cont'd

:  
Comparative ranking by total cost & I/O cost

Total Rank	I/O Rank	Cost Rank	Statement
1	1		SELECT i_price, i_name, i_data INTO :H00056, :H00055, :H00043
2	2		SELECT count(distinct S_I_ID) INTO :H00006 FROM ORDER_LINE, STOCK
3	3		UPDATE ORDER_LINE SET ol_delivery_d = :H00012 WHERE ol_w_id = :H00001
4	7		DECLARE READ_ORDERLINE_CUR CURSOR FOR SELECT ol_i_id, ol_supply_w_id,
5	6		UPDATE stock SET s_quantity = :H00052, s_order_cnt = :H0
6	5		UPDATE customer SET c_balance = :H00015, c_delivery_cnt =
7	4		UPDATE orders SET o_carrier_id = :H00002 WHERE o_id = :H00009 AND
8	8		UPDATE customer SET c_data1 = :H00039, c_data2 = :H00040
9			SELECT SUM( ol_amount ) INTO :H00011 FROM order_line WHERE ol_w_id =
10			SELECT s_quantity, s_dist_01, s_dist_02, s_dist_03, s_dist_04, s_dis
	9		DELETE FROM new_order WHERE no_w_id = :H00001 AND no_d_id = :H00008

Tables with unreferenced indexes

-----

Table: SREES.HISTORY  
HIST\_1

Table: SREES.ITEM  
ITEM\_IDX1  
ITEM\_1

## Notes:

- `db2perf_plans` doesn't currently ignore duplicate SQL statements
  - It might be reasonable for it to go for the most recent version of the plan
- There is a huge amount of information about SQL plans in the explain tables that could be mined!
  - Types of joins / scans / etc. used
  - Missing statistics
- The fact that indexes are unreferenced *in these plans* doesn't mean that they can necessarily be dropped
  - Extra digging likely required

# Agenda

- Motivation, goals & framework
- The tools
  - Configuration analysis
  - Snapshot analysis
  - Plan analysis
  - **Event monitor analysis**
- Wrap up



## Purpose of db2perf\_procevmon:

- Translates statement event monitor output produced by **'WRITE TO FILE'** option and **db2evmon** into .DEL files to IMPORT / LOAD back into DB2
  - Powerful tools in DB2 to mine this data!
- Creates a table with the same columns / layout as produced by **'WRITE TO TABLE'** option of **CREATE EVENT MONITOR**
  - Queries built with **'WRITE TO TABLE'** event monitor data in mind will work with tables built by **db2perf\_procevmon**

## Implementation:

C program

## How to use it:

### Preparation

#### 1. Compile db2perf\_procevmon

UNIX:        `cc -o db2perf_procevmon \`  
              `db2perf_procevmon.c \`  
              `-I $DB2PATH/include -L $DB2PATH/lib -l db2`

Windows: `cl db2perf_procevmon.c`  
          `rem %INCLUDE%, etc., must point to DB2 path`

#### 2. Capture statement event monitor output

`db2 "create event monitor e for statements`  
     `write to file '/tmp'"`

`db2 set event monitor e state=1`

*execute your workload ...*

`db2 set event monitor e state=0`

`db2evmon -path /tmp > db2evmon.out`

## How to use it, cont'd:

### Use

#### 1. Run **db2perf\_procevmon**

```
db2perf_procevmon <output file from db2evmon>  
                  <DEL file for statements> [ <DEL file for subsections> ]
```

for example

```
db2perf_procevmon db2evmon.out stmt_evt.del
```

#### 2. Create the tables to hold the statement / subsection data

```
db2 connect to <dbname>  
db2 -tvf db2perf_procevmon.db2
```

#### 3. LOAD / IMPORT the event monitor data into DB2 from the DEL file(s).

```
db2 load from <DEL file for statements> of DEL replace into  
      db2perf_evmon  
db2 load from <DEL file for subsections> of DEL replace into  
      db2perf_evmon_subsect
```

## How it works:

1. Reads lines from the input file
2. When the beginning of a statement event is seen
  - a) Collect values from the following lines and save them in an internal structure
  - b) When a line is seen that is not expected, dump what we have to the .DEL output file, and resume looking for the next line
3. Similar processing happens when a subsection event is seen

It would have been simpler to write things out as we find them, but we need to change the order of some fields to match the WRITE TO TABLE format

```

:
23) Statement Event ...
  Appl Handle: 13
  Appl Id: *LOCAL.DB2.060226054531
  Appl Seq number: 0020

  Record is the result of a flush: FALSE
  -----
  Type      : Dynamic
  Operation: Open
  Section   : 214
  Creator    : NULLID
  Package    : SQLC2E06
  Consistency Token : AAAAACEU
  Package Version ID :
  Cursor     : CLP_CURSOR_4
  Cursor was blocking: TRUE
  Text       : SELECT PARM_MODE FROM ...
  -----
  Start Time: 02/26/2006 00:55:06.286922
  Stop Time:  02/26/2006 00:55:06.286962
  Exec Time:  0.000040 seconds
  Number of Agents created: 1
  User CPU: 0.000000 seconds
  System CPU: 0.000000 seconds
:

```



## Notes:

- **db2evmon** output generally changes a bit from release to release
  - Compatible with v7.x, v8.2

GoFurther

## Purpose of db2perf\_evmon:

- Mines statement event monitor data to identify 'heavy hitters'
  - Top 10 SQL statements (either static or dynamic)
    - Execution time
    - Physical reads
    - Rows read
    - Sorts
  - COMMIT / ROLLBACK frequencies / times

## Implementation:

SQL/PL stored procedure

## How to use it:

### Preparation

1. Create the utility stored procedures  
`db2 -td@ -f db2perf_utils.db2`
2. Create the `db2perf_evmon` stored procedure  
`db2 -td@ -f db2perf_evmon.db2`

### Use

1. Connect to the desired database  
`db2 connect to <dbname>`
2. Collect statement event monitor data in DB2
  - Either using 'WRITE TO TABLE' option, or 'WRITE TO FILE' followed by `db2perf_procevmon` and LOAD
3. Call `db2perf_evmon( <tablename> [ , <top N> ] )`  
`db2 "call db2perf_evmon('evmon_tbl',20)"`

Report the  
'Top 20'  
statements  
in  
evmon\_tbl

## How it works:

1. Builds dynamic SQL 'count (\*)' statements to summarize overall event monitor data
  - # of events
  - # of transactions
  - COMMIT time

GoFurther



## How it works, cont'd:

2. For each of our 'Top *N*' categories by statement (elapsed time, rows read, etc.)
  - a) Builds dynamic SELECT to aggregate the statistic we're after, across all events with matching SQL text/package/section.
    - Fetches only the first *N* rows (default to 10)
    - Translates statement type codes, operation codes, etc., to words  
e.g. statement type 2 = 'Static SQL'
  - b) For each aggregate row fetched  
If the statement is static, retrieves the SQL text from **SYSCAT.STATEMENTS**

## Sample output

```
$ db2 "call db2perf_evmon('e_stmt_static',5)"
```

Statistics on event monitor table e\_stmt\_static

Number of events:..... 189798

Number of connections:..... 21

Number of transactions:..... 5580

Number of rollbacks:..... 0

Start / stop timestamps:..... 2006-03-06-15.06.23.975777 to 2006-03-06-15.16.20.279026 (956.3 seconds)

### Top 5 statements by elapsed time

Elapsed	Package	Section	# Events	CPU Time	Type	Statement
15.7	PAYS	2	2717	0.000	STATIC	DECLARE CUST_CURSOR1 CURSOR FOR SELECT
5.7	SYSSN400	58	8856	0.000	DYNAMIC	Select S_QUANTITY, S_DIST_01, S_DIST_0

:

### Top 5 statements by total physical reads

Physical

Reads	Package	Section	# Events	Type	Statement
5028	NEWS	7	14442	STATIC	SELECT s_quantity, s_dist_01, s_dist_02,
954	SYSSN400	58	8856	DYNAMIC	Select S_QUANTITY, S_DIST_01, S_DIST_02, S_DIST_0,

:

### Top 5 statements by rows read / written

:

### Top 5 statements by sort time

:

## Notes:

- Aggregated CPU times are often zero on some platforms due to minimum 10ms resolution supplied by the operating system
- Opportunity to extend this to exploit time series relationships
  - Time spent in the client
  - Synchronization / ordering of SQL statements
  - 'Pauses' in execution

# Summary

## Ten sample tools to simplify performance work on DB2!

<b>db2perf_sanity</b>	Configuration sanity check
<b>db2perf_utils</b>	Translate numeric monitor elements to strings
<b>db2perf_bufferpool</b>	Bufferpool snapshot analysis
<b>db2perf_dynsql</b>	Dynamic SQL snapshot analysis
<b>db2perf_locktree</b>	“Graphical” lockwait display
<b>db2perf_snapdiff</b>	Collect / compare snapshots
<b>db2perf_plandiff</b>	Highlight differences in plans
<b>db2perf_plans</b>	Explain table analysis
<b>db2perf_procevmon</b>	Translate db2evmon output for import into DB2
<b>db2perf_evmon</b>	Statement event monitor analysis



# Summary

- Source-based & easily extendible
- Many best practices built in, e.g.
  - Basic configuration guidelines in db2perf\_sanity
  - Bufferpool hotspots in db2perf\_bp
  - Places to use parameter markers in db2perf\_dynsql
  - Finding unused indexes in db2perf\_plans
- Many tasks made easier, e.g.
  - Understanding lock wait dependencies in db2perf\_locktree
  - Finding plan differences in db2perf\_plandiff
  - Finding changes in snapshot data in db2perf\_snapdiff
  - Simulating 'static SQL snapshot' in db2perf\_evmon
- Many great technologies demonstrated, e.g.
  - SQL/PL programming, nested & recursive calls, result sets
  - Advanced SQL – SELECT from INSERT, etc.
  - Information in the explain tables

## I like these – where do I get support?

- There is none – these are unsupported, as-is samples
  - ...to show you what monitoring data DB2 can produce
  - ...to show you how this data can be a great benefit to your system
  - ...to show you how to use DB2 monitoring interfaces
  - ...to show you how to use various DB2 technologies, such as SQL/PL stored procedures, and INSERT over SELECT, etc.
- You are welcome to use these, to study them, to modify them, etc. (subject to the usual legal terms in license.txt)
- Comprehensive, robust, fully-supported, integrated performance monitoring tools are available from both IBM and from 3<sup>rd</sup> parties. These work extremely well with DB2, and implement many of the features shown in these samples.

Session C02

DB2 Performance Samples: Ten Tools for Faster Systems

**Steve Rees**

IBM Canada Laboratory

srees@ca.ibm.com

GoFurther