



IBM Software Group

Sessions LU29 / LU30

DB2 Performance Toolbox: Ten Sample Tools for Faster Systems

Parts 1 & 2

Steve Rees

srees@ca.ibm.com

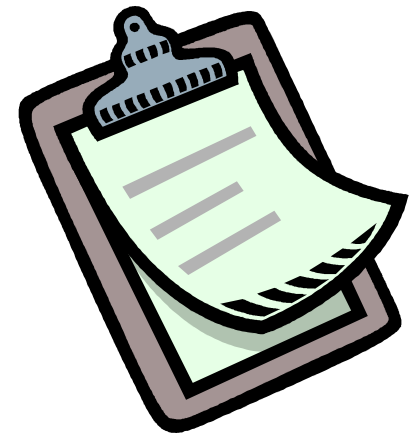
DB2 Information Management Software



ON DEMAND BUSINESS™

Agenda

- Motivation, goals & framework
 - The tools
 - ▶ Configuration analysis
 - ▶ Handy utilities
 - ▶ Snapshot analysis
 - ▶ Plan analysis
 - ▶ Event monitor analysis
 - Wrap up
- Part 1
- Part 2



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

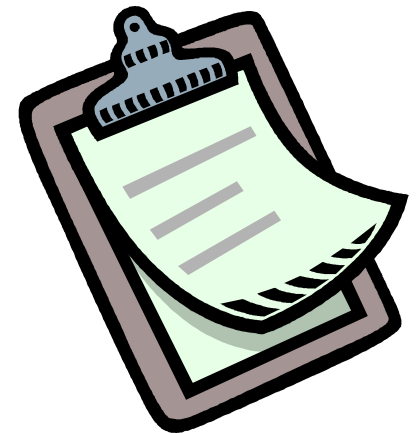


- Build on DB2's performance diagnostic interfaces
- Encapsulate techniques to extract, process, analyze & present performance data
 - ✓ Capture best practices from the lab & the field
 - ✓ Easy to use
 - ✓ Portable
 - ✓ Repeatable
 - ✓ Extendible
- Simplify the process of identifying problems
- Source for all tools will be available at

ftp://ftp.software.ibm.com/ps/products/db2/tools/

Agenda

- Motivation, goals & framework
 - The tools
 - ▶ **Configuration analysis**
 - ▶ Handy utilities
 - ▶ Snapshot analysis
 - ▶ Plan analysis
 - ▶ Event monitor analysis
 - Wrap up
- Part 1
- Part 2



#1: db2perf_sanity

db2perf_sanity

- ▶ Sanity checks configuration parameters for basic causes of performance problems, and makes recommendations
- ▶ 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



#1: db2perf_sanity

Why use it?

- ▶ Helps the DBA quickly zero-in on “no-brainer” problems
- ▶ Everything db2perf_sanity currently catches would be have been fixed by db2 autoconfigure anyway, but ...
 - Small databases built with the default parameters and tuned only sporadically can grow organically into systems people depend on
 - Problems may take a while to develop
 - Poor configuration settings may take even longer to be found
- ▶ Except in rare cases (e.g. when MINCOMMIT really does help) following the recommendations will be at worst neutral to system performance.



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



#1: db2perf_sanity

How to use it

Preparation

We borrow error handling code, etc., from the samples that come with DB2

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

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

2. Build the program

```
bldapp db2perf_sanity
```

For Windows, use

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

Use

1. Run the program

```
db2perf_sanity <dbname>
```

2. Update configuration based on results, if necessary



#1: db2perf_sanity

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/SQLOGDIR/', 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

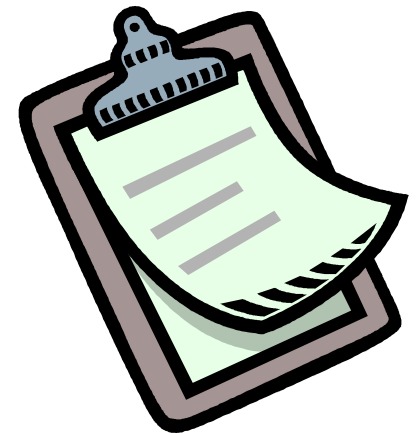
Some notes

- ▶ Based on rules-of-thumb
 - Generally very reliable, but not flawless
- ▶ Path analysis is 'quick & dirty' - may not be 100% accurate
 - More likely to miss a cases of unintentional disk sharing than to claim overlap when there is none
- ▶ Uses C because of need to get to configuration APIs
- ▶ Tool depends on DB2 v8.2.2 for table functions
- ▶ Future versions could ...
 - Be converted to SQL/PL once database configuration table functions are available
 - Check for other conflicts of LOGPATH, e.g. tablespace containers



Agenda

- Motivation, goals & framework
 - The tools
 - ▶ Configuration analysis
 - ▶ **Handy utilities**
 - ▶ Snapshot analysis
 - ▶ Plan analysis
 - ▶ Event monitor analysis
 - Wrap up
- Part 1
- Part 2



#2: db2perf_utils

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
 - Etc.
 2. 'Quiet' SQL **DROP** function
 - Suppresses 'not found' errors but lets other types of errors return to the caller

Implementation

SQL/PL Routines, UDFs



#2: db2perf_utils

Why use it?

- ▶ Translation UDFs make the output of the snapshot table functions and statement event monitors more usable & more similar to the text output from **GET SNAPSHOT**
- ▶ **db2perf_quiet_drop** is useful in CLP scripts when doing proactive cleanup of objects that might not yet exist

```
:  
-- We'll either get an error on the DROP if we  
-- haven't run the script before, or on the CREATE if  
-- we have. :-(  
DROP TABLE FOO;  
CREATE TABLE FOO ( c1 int );  
:  
-- Unless there's a *real* problem (e.g. authorization),  
-- both the DROP and CREATE will succeed :-(  
CALL db2perf_quiet_drop( 'TABLE BAR' );  
CREATE TABLE BAR ( c1 int );  
:
```



#2: db2perf_utils

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
```



#2: db2perf_utils

How to use it, cont'd

Use

1. Simply call translation UDFs 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_1kobj2str
	LOCK_MODE LOCK_MODE_REQUESTED	db2perf_1kmode2str
	LOCK_STATUS	db2perf_1kstat2str
SNAPSHOT_TABLE	TABLE_TYPE	db2perf_tabtyp2str
SNAPSHOT_APPL_INFO	APPL_STATUS	db2perf_apstat2str
SNAPSHOT_TBS_CFG	TABLESPACE_TYPE	db2perf_tbstyp2str
	TBS_CONTENTS_TYPE	db2perf_tbscon2str
Statement Event monitor	STMT_OPERATION	db2perf_op2str
	STMT_TYPE	db2perf_type2str

#2: db2perf_utils

How to use it, cont'd

2. 'Quiet drop' function

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

for example

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



#2: db2perf_utils

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



#2: db2perf_utils

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')"
```

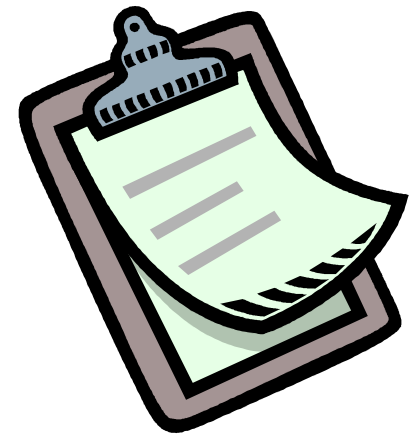
Return Status = 0

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

Table does not exist
but quiet_drop
suppresses the error

Agenda

- Motivation, goals & framework
 - The tools
 - ▶ Configuration analysis
 - ▶ Handy utilities
 - ▶ **Snapshot analysis**
 - ▶ Plan analysis
 - ▶ Event monitor analysis
 - Wrap up
- Part 1
- Part 2



- ## Implementation

#3: db2perf_bufferpool

Why use it?

- ▶ Checks snapshot data for common bufferpool performance issues
 - Both OLTP and complex query flavors
 - Encapsulates well-established calculations
- ▶ Provides severity ratings to issues detected
 - From '*hardly worth worrying about*' to '*fix this soon!*'
- ▶ Saves timestamped assessments in the database for ongoing monitoring



#3: db2perf_bufferpool

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



#3: db2perf_bufferpool

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

#3: db2perf_bufferpool

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.

#3: **db2perf_bufferpool**

Some 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



#4: db2perf_dynsql

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



#4: db2perf_dynsql

Why use it?

- ▶ Drilling down into most bottlenecks – CPU, I/O, etc. - eventually requires that culprit statements be identified
 - This tool speeds analysis of dynamic SQL snapshot data
- ▶ Automatically ranks statements by several important types of resources
 - By individual metrics
 - By combined metrics
- ▶ High-traffic statements using literals can combine to consume significant CPU
 - Candidate statements for parameter markers aren't always easy to recognize from snapshots
 - Low per-statement resource consumption excludes them from most normal queries
 - Tool helps do 'what if' analysis on current statements
 - How many distinct statements could be replaced by one with parameter markers?

Hint - statements which consume high amounts of CPU *and* IO are a good place to start looking for problems



#4: db2perf_dynsql

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`



#4: db2perf_dynsql

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

```
CREATE VIEW db2perf_dynsql_view AS
    SELECT * FROM table(snap_get_dyn_sql(CAST (NULL as varchar(256)),-1)) as t;
CREATE TABLE db2perf_dynsql LIKE db2perf_dynsql_view;

INSERT INTO db2perf_dynsql
    SELECT * FROM table(snap_get_dyn_sql(CAST (NULL as varchar(256)),-1)) as t;

ALTER TABLE db2perf_dynsql
    ADD COLUMN top10_elapsed CHAR(2)
    ADD COLUMN top10_CPU CHAR(2)
    ADD COLUMN top10_phys_read CHAR(2)
    ADD COLUMN top10_rows_read CHAR(2)
    ADD COLUMN top10_sorts CHAR(2)
    ADD COLUMN top10_spilled CHAR(2)
    :
```

New column to contain
1-to-10 ranking of this
statement by Rows Read



#4: db2perf_dynsql

How it works, cont'd

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) );
  
```

3) Outer SELECT prints out the metrics and the ranking

2) Row_number() from inner SELECT gives us the rank of each statement within the result set... which is then UPDATED back into the ranking column in the base table, so we can use it later

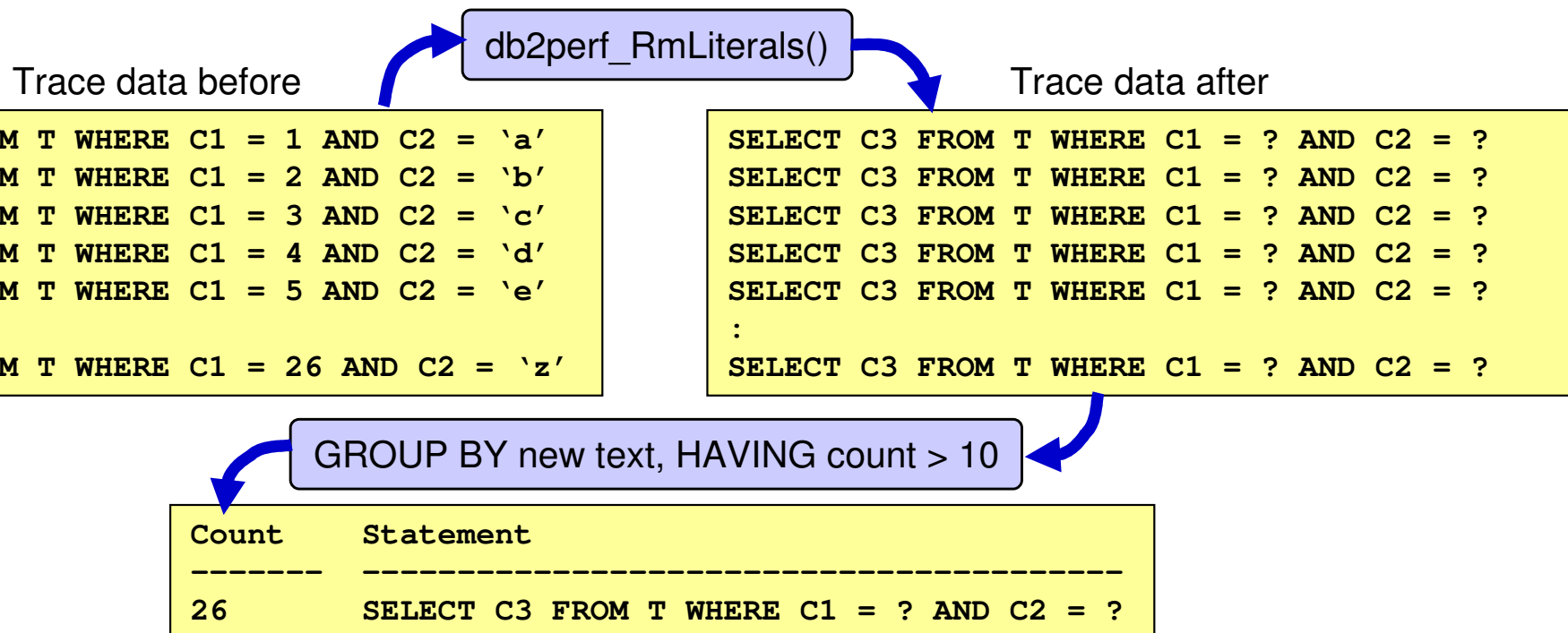
1) Finds top 10 statements in descending order by this metric

4. Pulls out all statements from our work table which are "Top 10" for *any* metric – chances are that some of them are Top 10 for more than one metric.

#4: db2perf_dynsql

How it works, cont'd

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.



#4: db2perf_dynsql

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
statements (and PREPAREs !) with
just one statement that uses
parameter markers

#4: db2perf_dynsql

Some 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



#5: db2perf_locktree

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



#5: db2perf_locktree

Why use it?

- ▶ A tree is the natural representation of lock waits in a busy system
- ▶ The tool provides easy visualization of lock wait relationships
 - Includes information joined from other snapshots, e.g. application name
- ▶ Much quicker than pen-and-paper translation of lock snapshot data into a diagram
 - Repeated calls to show evolving lock wait states are trivial to obtain!



#5: db2perf_locktree

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
```

```
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**



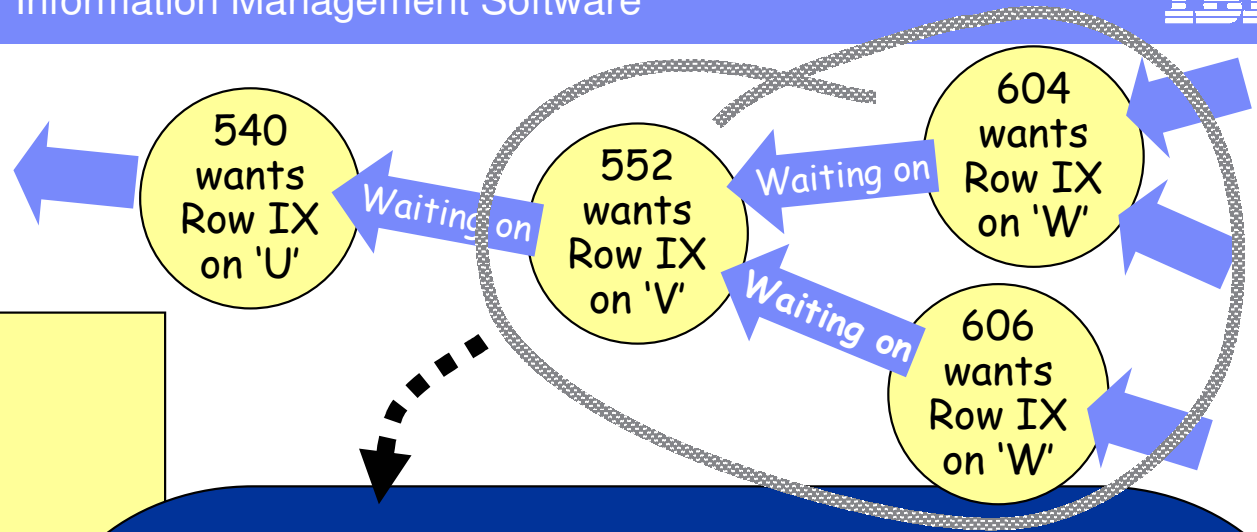
#5: `db2perf_locktree`

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. Recursively processes each instance of lock wait as follows
 - a) Draws 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.
 - c) 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

```

Application name

```

:-----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 Exclu...
: | Lock wait time (ms): 44941
: | Lock escalation: N
: | Table name: SREES.W
:
:-----Waiter appl handle: 606 (getlock)
: | Holder appl handle: 552 (getlock)
:
:
:

```

#5: `db2perf_locktree`

Some notes:

- ▶ Recursive SQL/PL calls are an ideal choice here
 - Maximum SQL/PL nesting depth 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



End of Part 1

Ten First 5 sample tools to simplify performance work on DB2!

db2perf_sanity	Configuration sanity check
db2perf_utils	Translate numeric monitor elements to strings
db2perf_bufferpool	Bufferpool snapshot analysis
db2perf_dynsql	Dynamic SQL snapshot analysis
db2perf_locktree	“Graphical” lockwait display

db2perf_snapdiff	Collect / compare snapshots
db2perf_plandiff	Highlight differences in plans
db2perf_plans	Explain table analysis
db2perf_procevmon	Translate db2evmon output for analysis in DB2
db2perf_evmon	Statement event monitor analysis

Coming up in Part 2





IBM Software Group

Session LU30

DB2 Performance Toolbox:

Ten Tools for Faster Systems

Part 2

Steve Rees

srees@ca.ibm.com

DB2 Information Management Software



ON DEMAND BUSINESS™

Quick Recap of Part 1

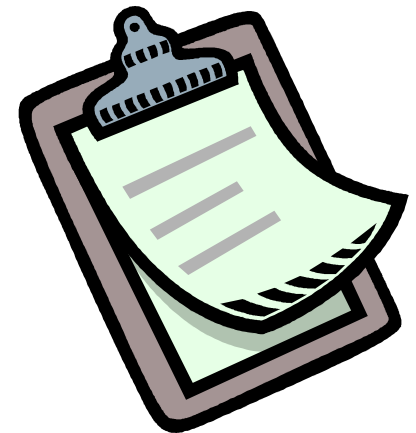
Ten First 5 sample tools to simplify performance work on DB2!

db2perf_sanity	Configuration sanity check
db2perf_utils	Translate numeric monitor elements to strings
db2perf_bufferpool	Bufferpool snapshot analysis
db2perf_dynsql	Dynamic SQL snapshot analysis
db2perf_locktree	“Graphical” lockwait display




Agenda

- Motivation, goals & framework
 - The tools
 - ▶ Configuration analysis
 - ▶ Handy utilities
 - ▶ **Snapshot analysis**
 - ▶ Plan analysis
 - ▶ Event monitor analysis
 - Wrap up
- } Part 1
- } Part 2



#6: `db2perf_snapdiff``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



Very easy to
extend to other
snapshot types!

Implementation:

SQL/PL stored procedures



#6: db2perf_snapdiff

Why use it?

- ▶ Simplifies the process of collecting and tracking snapshot data over time
 - Keeps data organized inside DB2, instead of in ordinary text files from GET SNAPSHOT
 - Exploits DB2's capabilities for analyzing data
 - Automatically identifies snapshot elements which have changed significantly
- ▶ Improves usability of snapshot table functions
 - Determines 'baseline' values without requiring that counters be reset to zero
- ▶ Implementation is extremely flexible and extendible
 - To other snapshot data types
 - To other types of time-based data



#6: db2perf_snapdiff

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
```

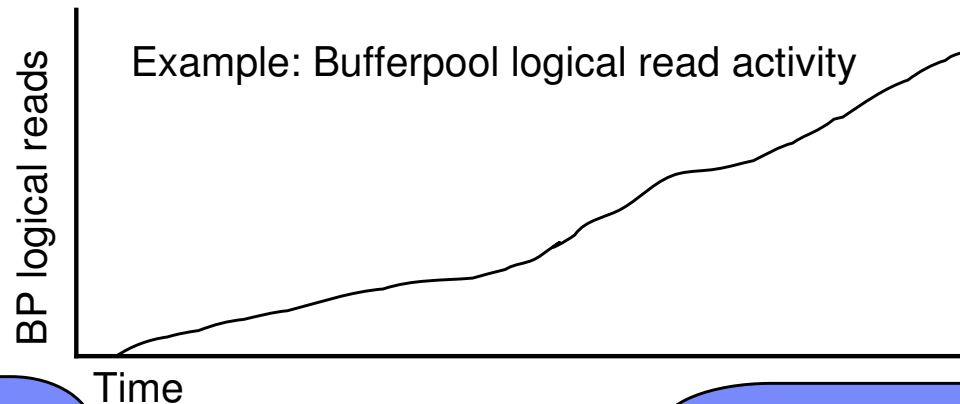
```
db2 -td@ -f db2perf_snapdiff.db2
```

3. Turn on monitoring by default with DFT_MON_xxx database manager configuration switches

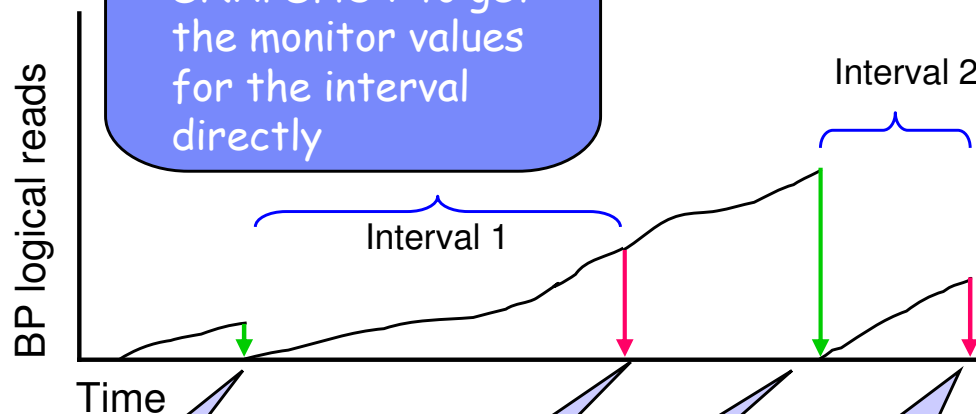


#6: db2perf_snapdiff

First, two ways to use snapshots



1) Alternate RESET MONITOR and GET SNAPSHOT to get the monitor values for the interval directly



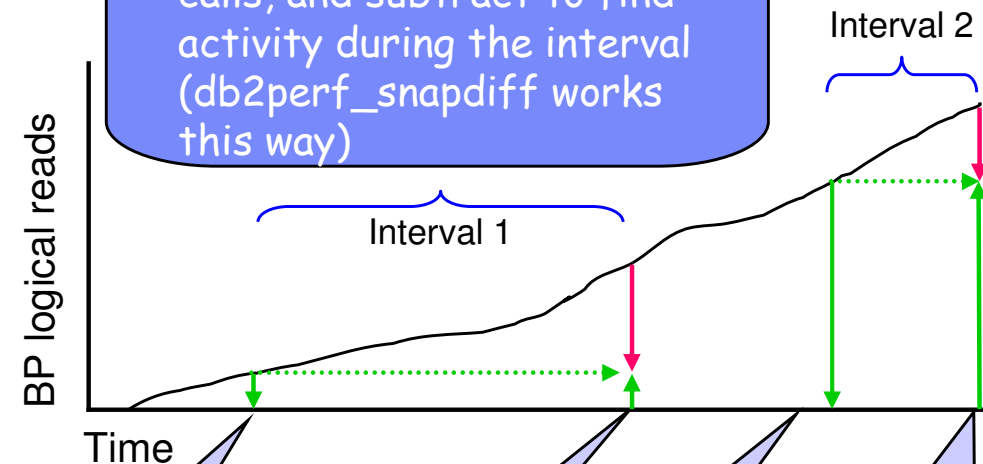
Reset monitor at start of 1st interval

GET SNAPSHOT gives activity for just the 1st interval

Reset monitor at start of 2nd interval

GET SNAPSHOT gives activity for just the 2nd interval

2) Repeat GET SNAPSHOT calls, and subtract to find activity during the interval (db2perf_snapdiff works this way)



GET SNAPSHOT at start of 1st interval

SNAPSHOT again and calculate differences since last s/s

GET SNAPSHOT at start of 2nd interval

SNAPSHOT again and calculate differences since last s/s

#6: db2perf_snapdiff

How to use db2perf_snapdiff - 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

#6: db2perf_snapdiff

How to use it - more 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('compare')"
```



#6: db2perf_snapdiff

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('compare',  
    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%

#6: db2perf_snapdiff

How it works - collecting data:

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

The 'TOC' (table of contents) 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')

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	...

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	...

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')

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	...

#6: db2perf_snapdiff

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 snapshot rows from this table which have:
 - 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



Too 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

0 - 0 = 0

(45,654 - 27,657) / 1000

0 - 0 =
0 change
between
Intervals 1 & 2

Interval 2 post thresh sorts / 1k Tx

0 - 0 = 0

(253,236 - 232,945) / 1000

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

(836,199 - 504,265) / 1000 = 18,443

Interval 2 rows read / 1k Tx

(4,627,251 - 4,253,835) / 1000 = 18,403

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

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

(20,574 - 12,460) / 1000 = 451

Int. 1 **STOCK** rows written / 1k Tx

(205,900 - 124,600) / 1000 = 4,517

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

Int. 2 **DISTRICT** rows written / 1k Tx

(114,106 - 104,881) / 1000 = 455

Int. 2 **STOCK** rows written / 1k Tx

(1,140,702 - 1,048,906) / 1000 = 4,524

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

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

#6: db2perf_snapdiff

Sample output

```
$ 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%

```
Normalizing to 1K Tx per Interval
```

```
*** ROWS_READ
*** POOL_DATA_L_READS
*** POOL_DATA_P_READS
** POOL_DATA_WRITES
* POOL_INDEX_L_READS
* POOL_INDEX_P_READS
** POOL_INDEX_WRITES
** POOL_READ_TIME
** POOL_WRITE_TIME
```

```
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
```

40.9	0.1	-99.5 %
18228.0	412066692.7	2260516.9 %
27898.6	10196246.9	36447.4 %
1652.4	15433.7	833.9 %
1665.8	0.0	-100.0 %
108994.1	73801.2	-32.2 %
96.9	114.4	18.0 %
262.7	0.0	-100.0 %
3025.5	4753.0	57.0 %
14882.7	0.0	-100.0 %

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

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

#6: db2perf_snapdiff

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 %
:

```


#6: db2perf_snapdiff

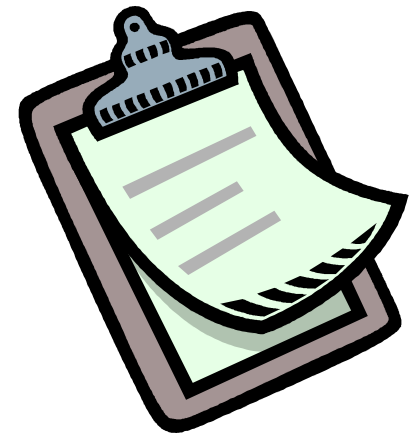
Some notes:

- ▶ 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
 - ▶ Handy utilities
 - ▶ Snapshot analysis
 - ▶ **Plan analysis**
 - ▶ Event monitor analysis
 - Wrap up
- Part 1
- Part 2



#7: db2perf_plandiff

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



#7: db2perf_plandiff

Why use it?

- ▶ Plan changes can have a *dramatic* change in performance and they aren't always easy to predict
 - Caused by changes in configuration
 - ... by changes in data volume
 - ... by changes in DB2 code level
 - ... by many other things too
- ▶ Comparing plans of individual statements isn't hard, but searching large sets of plans for changes is tedious & error-prone
 - Costs, filter factors, rows returned, etc. can (and do often) change even when the plan stays the same
 - Straight text comparison of db2expln / db2exfmt output typically yields many 'false positives'
- ▶ db2perf_plandiff saves time by giving a quick "changed / didn't change" indication
 - Lets you choose which plans to dig into in more detail



#7: db2perf_plandiff

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`
`db2 -td@ -f db2perf_plandiff.db2`
3. 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>)"
```



#7: **db2perf_plandiff**

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

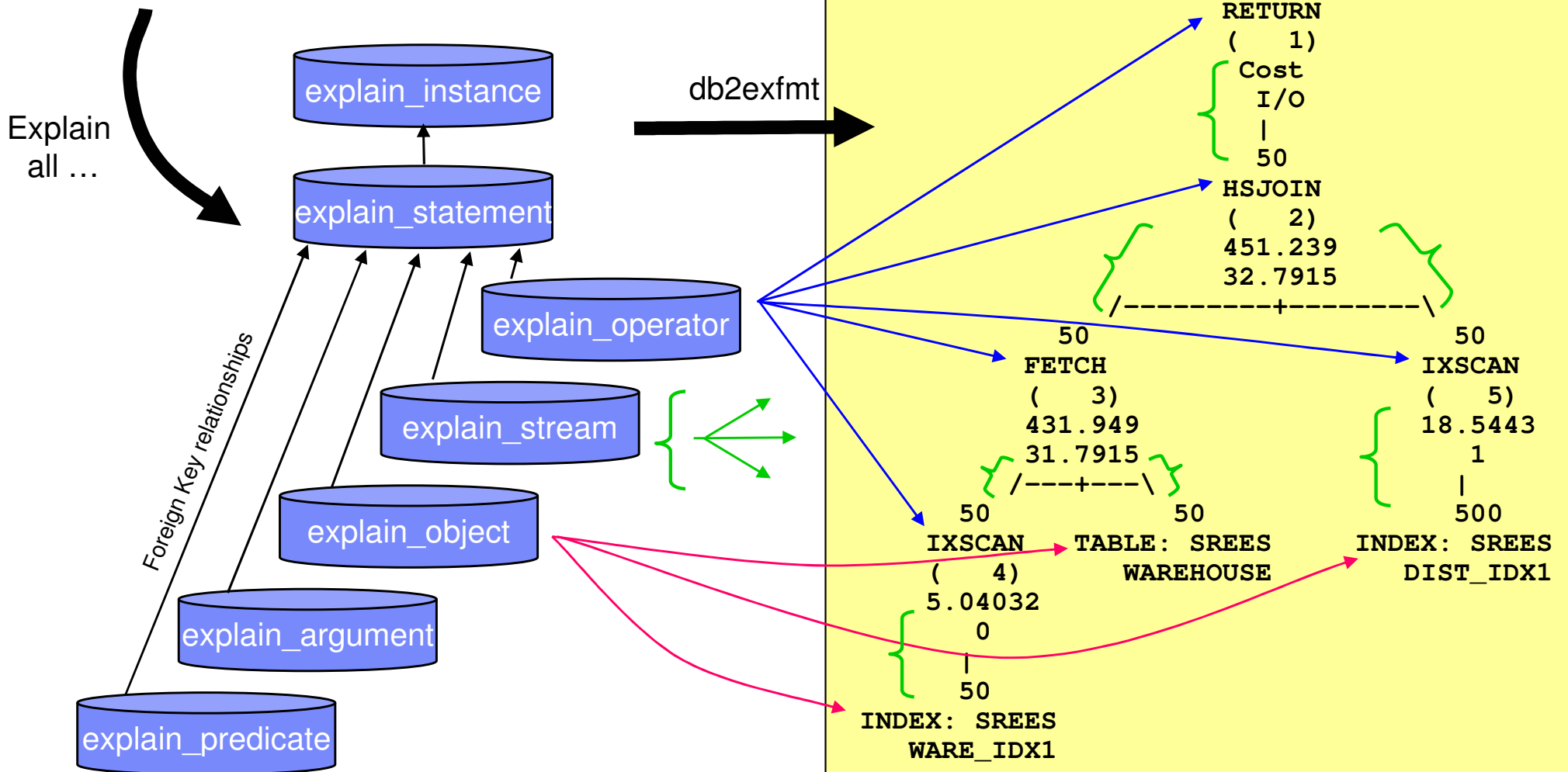
3. If differences are reported, use the contents of **db2perf_plandiff_report** to determine which statements to examine with **db2exfmt**



#7: db2perf_plandiff

The explain tables

```
"select w_city from warehouse, district
where w_id = d_w_id and d_id = 5"
```



#7: db2perf_plandiff

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)

#7: `db2perf_plandiff`

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



#7: db2perf_plandiff

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
Yes	SREES.DELS	2	2006-...296586	38	
	SREES.DELS	2	2006-...132561	28251	
					DELETE FROM new_order WHERE no_w_id = :H0 0001 AND no_d_id = :H00008 AN D 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 AN D o_w_id = :H00001 AND o_d_id = :H00008
:					
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.					

Cost of SELECT & DELETE on NEW_ORDER has skyrocketed - check out plans for these statements in db2exfmt

No apparent change in plan for UPDATE on ORDERS

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

#7: db2perf_plandiff

Some 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



#8: db2perf_plans

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



#8: db2perf_plans

Why use it?

- ▶ Explain plans complement runtime information from snapshots, etc., in search for expensive statements
 - Expensive statements identified here are prime candidates for investigation with db2exfmt
 - Are you getting the plan you think you should get?
- ▶ Indexes can tend to accumulate over time
 - Superfluous indexes still consume storage and have to be maintained even when they provide no value
 - Eliminating unneeded indexes on highly updated tables can reduce statement cost
 - db2perf_plans can provide useful clues as to which indexes aren't being used



#8: db2perf_plans

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
```

```
db2 -td@ -f db2perf_plans.db2
```

3. 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()"
```



#8: 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 topmost operator in the plan; its IO cost represents the whole plan

#8: db2perf_plans

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	

```
:
```


#8: db2perf_plans

Sample output, cont'd

:

Comparative ranking by total cost & I/O cost

Total Cost Rank	I/O 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

#8: db2perf_plans

Some 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 will usually be required



Agenda

- Motivation, goals & framework
 - The tools
 - ▶ Configuration analysis
 - ▶ Handy utilities
 - ▶ Snapshot analysis
 - ▶ Plan analysis
 - ▶ **Event monitor analysis**
 - Wrap up
- } Part 1
- } Part 2



#9: db2perf_procevmon

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



#9: db2perf_procevmon

Why use it?

- ▶ Event monitor data is difficult and cumbersome to analyze one event at a time
 - DB2 is well suited to this task!
 - Aggregation
 - Filtering
 - Time-series analysis
 - etc.
- but we have to find a way to get the data inside
- ▶ The **WRITE TO TABLE** option is the best way, but ...
 - **WRITE TO FILE** is somewhat faster
 - Text format is sometimes quite useful
 - Initial hands-on review of the data
 - Remote data collection
- ▶ db2perf_procevmon makes event monitor data collected as text just as useful as that initially written into a DB2 table



#9: db2perf_procevmon

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
```



#9: db2perf_procevmon

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
```



#9: db2perf_procevmon

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
:

```


#9: db2perf_procevmon

Some notes:

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



#10: db2perf_evmon

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



Why use it?

- ▶ Statement event monitors provide detailed data that snapshots only summarize
 - Per-execution resource consumptions
 - Static SQL execution information
 - COMMIT and ROLLBACK occurrences
 - Inter- and intra-statement timings
 - Trends in execution behavior
- ▶ Statement event monitor data is difficult to consume 'by inspection'
 - Proper tooling provides insight into system execution that is very unlikely to be obtained manually, for example
 - Filtering by statement content
 - Aggregation of time spent 'above' DB2 in the client
 - Detection of variance in execution times



#10: db2perf_evmon

How to use it:

Preparation

1. Create the stored procedures

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



#10: db2perf_evmon

How it works:

1. Builds dynamic SQL `'count (*)'` statements to summarize overall event monitor data
 - # of events
 - # of transactions
 - COMMIT time
2. For each of our 'Top N' categories (elapsed time, rows read, etc.)
 - a) Builds dynamic SQL SELECT to aggregate the metrics we're after, across all events with matching SQL text / package / section
 - Fetches only the first N aggregations (default to 10)
 - Translates statement type codes, operation codes, etc., to words
e.g. `WHERE db2perf_op2str(stmt_operation) IN ('PREPARE','EXECUTE'...`
 - b) For each aggregate row fetched
 - If the statement is static, retrieves the SQL text from **SYSCAT.STATEMENTS**

Uses our utility UDFs from db2perf_utilities: db2perf_op2str, db2perf_type2str, etc.

#10: db2perf_evmon

Sample output

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

Statistics on event monitor table e_stmt_view

Number of events:..... 189798

Number of connections:..... 21

Number of transactions:..... 5580

Number of rollbacks:..... 0

Start / stop timestamps:..... 2006...-15.06.23.975777 to ...-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 C1 CURSOR FOR SELECT
5.7	SYSSN400	58	8856	0.000	DYNAMIC	Select S_QUANTITY, S_DIST_01,

:

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,

:

Top 5 statements by rows read / written

:

Top 5 statements by sort time

:

#10: db2perf_evmon

Some 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!	
db2perf_sanity	Configuration sanity check
db2perf_utils	Translate numeric monitor elements to strings
db2perf_bufferpool	Bufferpool snapshot analysis
db2perf_dynsql	Dynamic SQL snapshot analysis
db2perf_locktree	“Graphical” lockwait display
db2perf_snapdiff	Collect / compare snapshots
db2perf_plandiff	Highlight differences in plans
db2perf_plans	Explain table analysis
db2perf_procevmon	Translate db2evmon output for import into DB2
db2perf_evmon	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_bufferpool
 - ▶ 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)
- For comprehensive, robust, fully-supported, enterprise-level performance monitoring, there are many tools on the market, such as DB2 Performance Expert, that work extremely well with DB2, and which implement many of the features shown in these samples.



Steve Rees
IBM Canada Laboratory
srees@ca.ibm.com