- ibm.com/db2/lab chats

Data Management

The Latest in Advanced Performance Diagnostics for DB2 LUW

May 31, 2011

ibm.com/db2/labchats

> Executive's Message

Sal Vella

Vice President, Development,
Distributed Data Servers and Data Warehousing

IBM

IBM

> Featured Speaker

Steve Rees

Senior Performance Manager
DB2 for Linux, UNIX, and Windows

IBM

Agenda

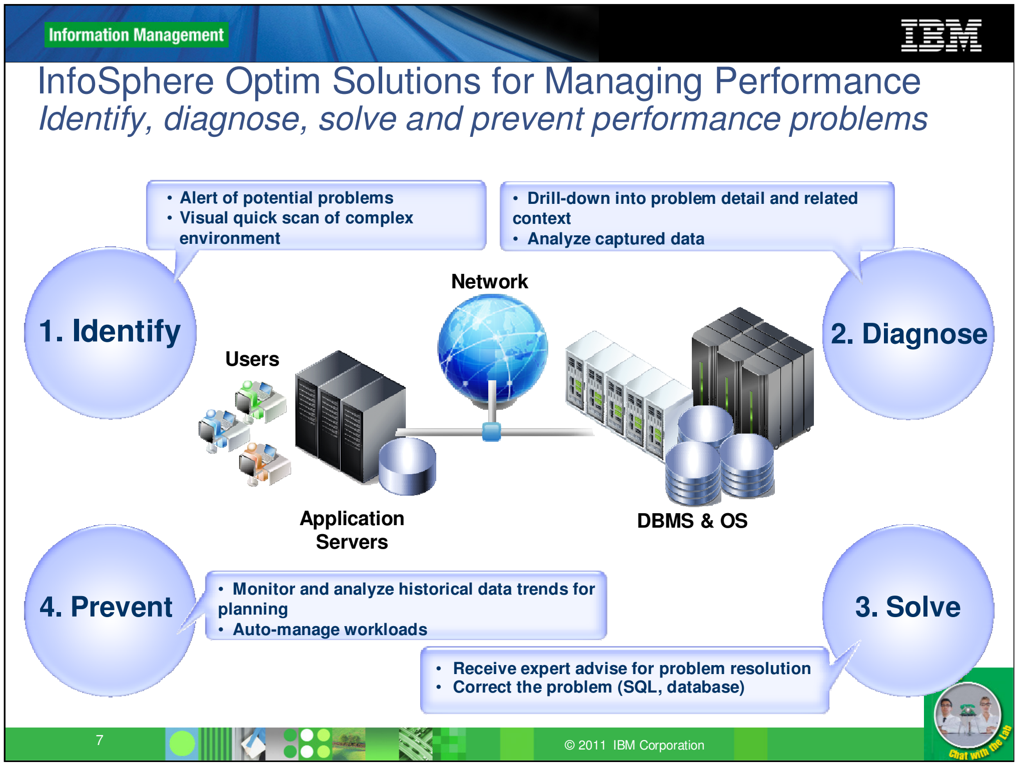
- **Monitoring DB2 – DIY or GUI?**
- **Overview of performance diagnostics in DB2 LUW 9.7**
- **SQL monitoring for the snapshot addicted**
- **Choices, choices … what should I pay attention to?**
  - Key operational metrics for system-level performance
  - Drilling down with statement-level monitoring
- **Summary**

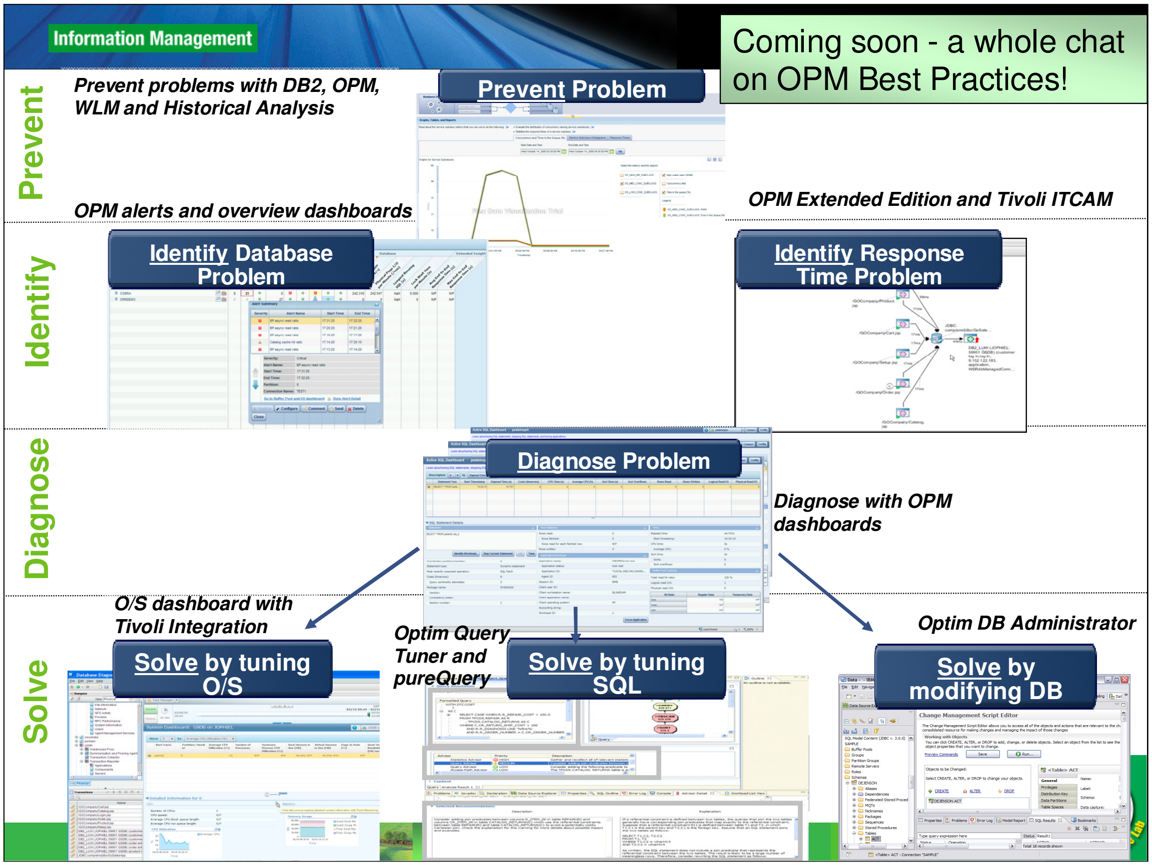
Goals

- Smooth the path from snapshots to SQL monitoring
- Review the 'big hitters' - performance metrics you shouldn't be without for operational monitoring
- Provide tips & tricks & best practices for tracking them easily
- Provide guidelines for reasonable values in transactional and warehouse systems
- Show examples & sample code to get you started
- But first – we'll give a sneak peek of an upcoming Chat on performance monitoring with Optim Performance Manager

Monitoring DB2 – DIY is fine, but you may prefer GUI !

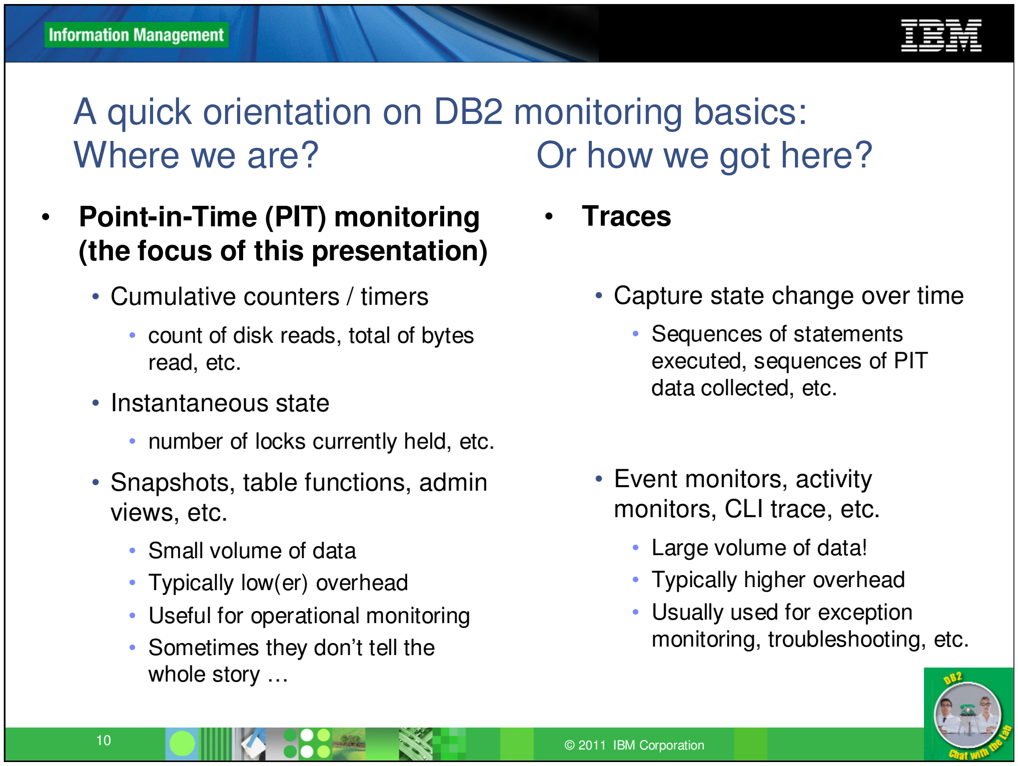
- **IBM Optim Performance Manager (OPM)**
  - Powerful, intuitive, easy-to-use performance monitoring
  - OPM scales from monitoring small individual instances to entire data centers of DB2 systems
  - Alert mechanisms inform the DBA of potential problems
  - Historical tracking & aggregation of metrics enable trending of system performance over time
- **OPM Extended Insight**
  - Sophisticated mechanisms to measure end-to-end response time to detect issues outside DB2!
- **Optim Query Tuner**
  - Deep-dive analysis to identify and solve many types of query bottlenecks


Information Management
IBM
InfoSphere Optim Solutions for Managing Performance
Identify, diagnose, solve and prevent performance problems
• Alert of potential problems
• Visual quick scan of complex environment
• Drill-down into problem detail and related context
• Analyze captured data
1. Identify
2. Diagnose
Network
Users
Application Servers
DBMS & OS
4. Prevent
3. Solve
• Monitor and analyze historical data trends for planning
• Auto-manage workloads
• Receive expert advise for problem resolution
• Correct the problem (SQL, database)
7
© 2011 IBM Corporation
Chat with the Lab

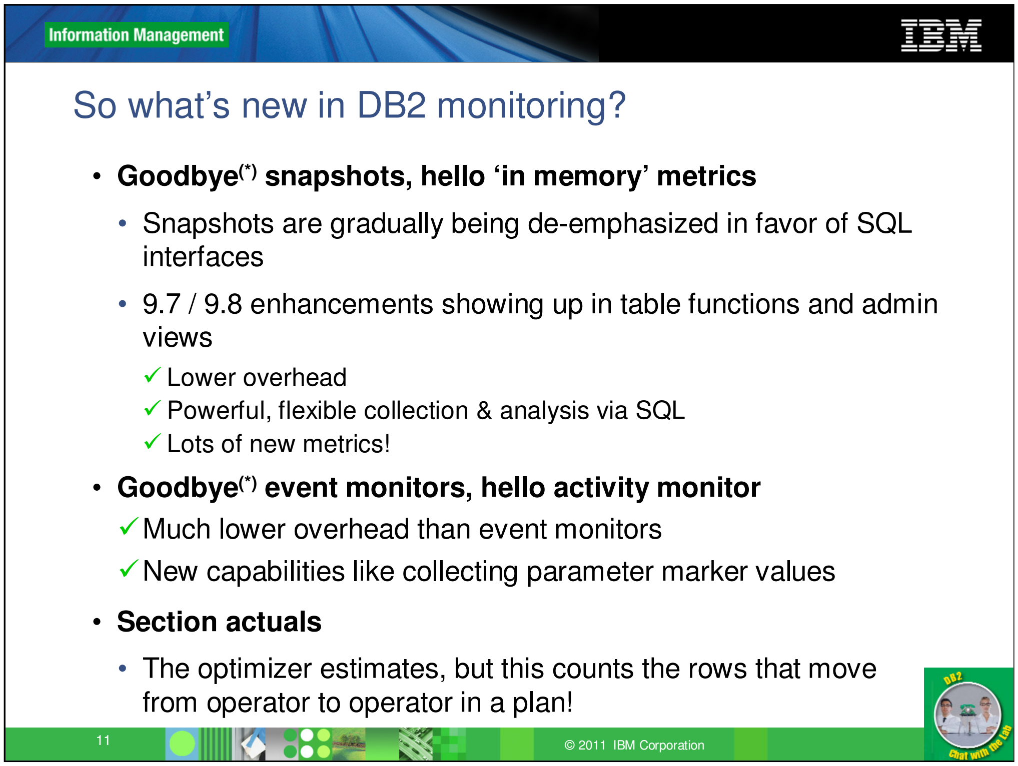

Information Management
Coming soon - a whole chat on OPM Best Practices!
Prevent
Prevent problems with DB2, OPM, WLM and Historical Analysis
Prevent Problem
OPM alerts and overview dashboards
OPM Extended Edition and Tivoli ITCAM
Identify
Identify Database Problem
Identify Response Time Problem
Diagnose
Diagnose Problem
Diagnose with OPM dashboards
Solve
O/S dashboard with Tivoli Integration
Optim Query Tuner and pureQuery
Optim DB Administrator
Solve by tuning O/S
Solve by tuning SQL
Solve by modifying DB

Agenda

- DB2 monitoring – DIY or GUI?
- **Overview of performance diagnostics in DB2 LUW 9.7**
- SQL monitoring for the snapshot addicted
- Choices, choices … what should I pay attention to?
  - Key operational metrics for system-level performance
  - Drilling down with statement-level monitoring
- Summary



Point-in-time monitoring is the new term for what we used to think of as 'snapshot' monitoring.



'In-memory' metrics is how many of the new DB2 9.7 monitors are described – meaning that the metrics are retrieved directly & efficiently from in-memory locations, rather than having to be maintained and accessed in more expensive ways – as the snapshots were.

Agenda

- DB2 monitoring – DIY or GUI?
- Overview of performance diagnostics in DB2 LUW 9.7
- **SQL monitoring for the snapshot addicted**
- Choices, choices … what should I pay attention to?
  - Key operational metrics for system-level performance
  - Drilling down with statement-level monitoring
- Summary

The 'Brave New World' of SQL access to perf data …

- **Snapshots are great for ad hoc monitoring, but not so great for ongoing data collection**
  - ✕ Parsing & storing snapshot data often requires messy scripts
  - ✕ Snapshots are fairly difficult to compare
  - ✕ Snapshots tend to be 'all or nothing' data collection – difficult to filter
- **Table function wrappers for snaps have existed since v8.2**
  - Great introduction to the benefits of SQL access
  - Had some limitations…
    - ✕ Layers of extra processing – not free …
    - ✕ Sometimes wrappers got out-of-sync with snapshots
    - ✕ Some different behaviors, compared to snapshots

What's so great about SQL access to monitors?

1. **pick and choose just the data you want**
   - One or two elements, or everything that's there
2. **store and access the data in its native form**
   - DB2 is a pretty good place to keep SQL-sourced data ☺
3. **apply logic to warn of performance problems during collection**
   - Simple range checks really are simple!
4. **perform sophisticated analysis on the fly, or on saved data**
   - Joining different data sources, trending, temporal analysis, normalization, …

"But I *like* snapshots!" - Advice for the snapshot-addicted

- **Relax – snapshots will be around for a while!**
- **Good basic monitor reports available from the new MONREPORT modules (new in v9.7 fp1)**
  - Available in new created or migrated databases
    - Tip: coming from v9.7 GA? use db2updv97
  - Display information over a given interval (default 10s)
  - Implemented as stored procedures, invoked with CALL

monreport.dbsummary	**monreport.connection**
monreport.pkgcache	**monreport.lockwait**
monreport.currentsql	**monreport.currentapps**

The monreport modules can be a very handy way of getting text-based reports of monitor values

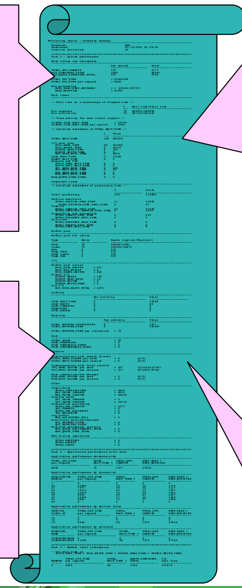
monreport.dbsummary - Commits/s, wait & processing time summary, bufferpool stats, etc.

monreport.pkgcache - Top-10 SQL by CPU, IO, etc.

The others are fairly self-explanatory.

Example – monreport.dbsummary

```
Work volume and throughput
--------------------------------------------
                             Per second   Total
------------------------    -----------  -------
TOTAL_APP_COMMITS            137          1377
ACT_COMPLETED_TOTAL          3696         36963
APP_RQSTS_COMPLETED_TOTAL    275          2754

TOTAL_CPU_TIME                       = 23694526
TOTAL_CPU_TIME per request           = 8603

Row processing
ROWS_READ/ROWS_RETURNED       = 1 (25061/20767)
ROWS_MODIFIED                 = 22597
```



```
Wait times
---------------------------------------
-- Wait time as a percentage of elapsed time
                       %    Wait time/Total time
                      ---   --------------------
For requests          70    286976/406856
For activities        70    281716/401015
-- Time waiting for next client request --
CLIENT_IDLE_WAIT_TIME                 = 13069
CLIENT_IDLE_WAIT_TIME per second      = 1306
-- Detailed breakdown of TOTAL_WAIT_TIME
                           %    Total
                          ---   -------------
  TOTAL_WAIT_TIME         100   286976

  I/O wait time
    POOL_READ_TIME        88    253042
    POOL_WRITE_TIME       6     18114
    DIRECT_READ_TIME      0     100
    DIRECT_WRITE_TIME     0     0
    LOG_DISK_WAIT_TIME    1     4258
  LOCK_WAIT_TIME          3     11248
  AGENT_WAIT_TIME         0     0
  :
```

```
Component times
-----------------------------------------------
-- Detailed breakdown of processing time --
                                   %     Total
                                  ----   --------
Total processing                  100    119880

Section execution
  TOTAL_SECTION_PROC_TIME         11     13892
    TOTAL_SECTION_SORT_PROC_TIME  0      47
Compile
  TOTAL_COMPILE_PROC_TIME         22     27565
  TOTAL_IMPLICIT_COMPILE_PROC_TIME 2     3141
Transaction end processing
  TOTAL_COMMIT_PROC_TIME          0      230
  TOTAL_ROLLBACK_PROC_TIME        0      0
:
```

```
Buffer pool
----------------------------------------------
Buffer pool hit ratios

Type        Ratio      Reads (Logical/Physical)
----------  --------   ------------------------
Data        72         54568/14951
Index       79         223203/45875
XDA         0          0/0
Temp data   0          0/0
Temp index  0          0/0
Temp XDA    0          0/0
```

Information Management IBM

Tips on migrating to SQL access from snaps

- **Monitor switches are for snapshots**
  - Database config parameters control what's collected in the new system.
- **Can't remember all the table function, or want to know what's new?   Ask the catalog …**

```
db2 select substr(funcname,1,30) from syscat.functions
where funcname like 'MON_%'
   or funcname like 'ENV_%'
```

- **Include the timestamp of collection with the data**
  - Most snapshots automatically included one
  - SQL interfaces let you exclude this, even if it's there
- **RESET MONITOR is a snapshot thing**
  - SQL-based PIT data is not affected by RESET MONITOR
  - Delta values (*after* minus *before*) achieve the same thing

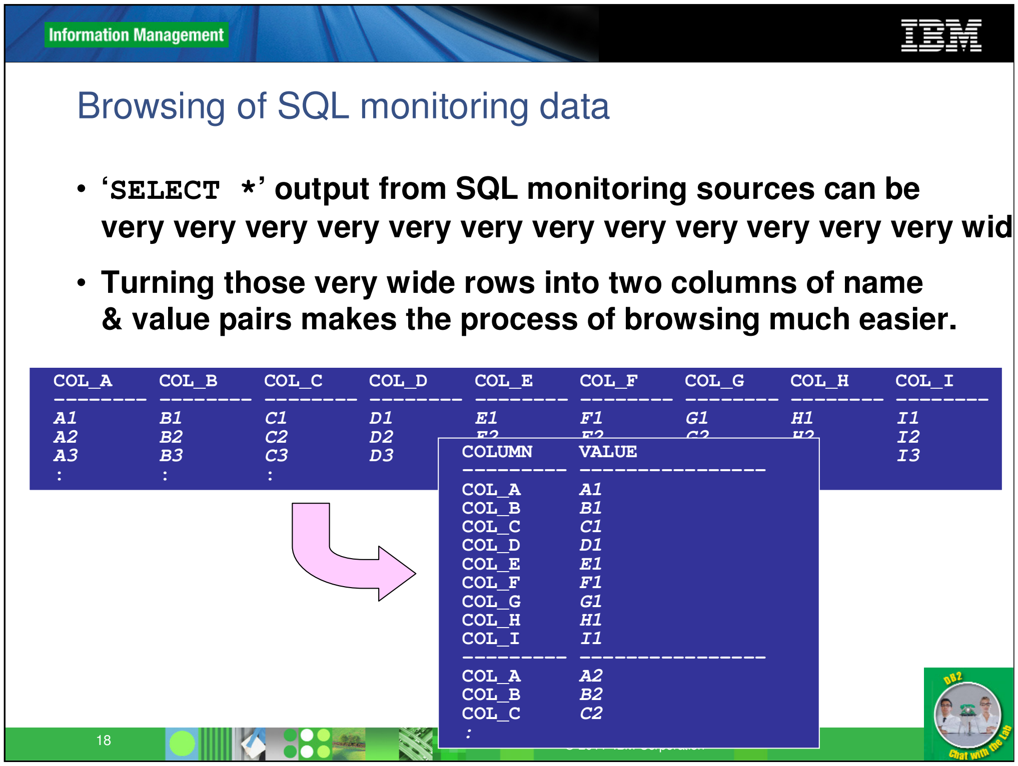
17

Things like 'update monitor switches', and the settings of instance-level defaults like DFT_MON_BUFFERPOOL, are only for snapshots, and don't effect what's collected in the new PIT monitoring.

The new PIT monitoring interfaces are controlled by 3 dynamically-changeable db config switches

Request metrics	(MON_REQ_METRICS) = BASE
Activity metrics	(MON_ACT_METRICS) = BASE
Object metrics	(MON_OBJ_METRICS) = BASE

They can be set to NONE – which provides very little data, or BASE, which is the default and is generally adequate.



The sheer width of the new SQL monitoring data can be a little discouraging, if you're used to being able to page down through a snapshot.

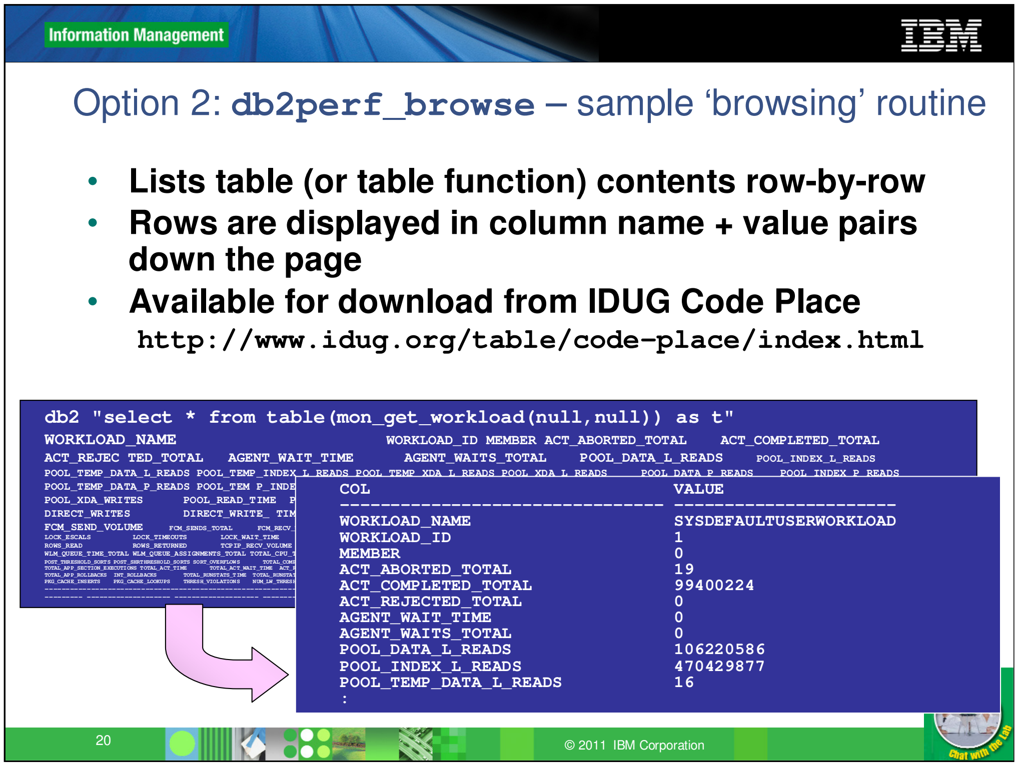
Option 1: Filter with row-based table functions

- `mon_format_xml_metrics_by_row` **formats 'detailed' monitor & event XML documents and returns the fields in name/value pairs**

```
db2 "select substr(M.METRIC_NAME, 1, 25) as METRIC_NAME, M.VALUE
   from table( MON_GET_WORKLOAD_DETAILS( null,-2 ) ) AS T,
        table( MON_FORMAT_XML_METRICS_BY_ROW(T.DETAILS)) AS M
where T.WORKLOAD_NAME = 'SYSDEFAULTUSERWORKLOAD'
order by METRIC_NAME asc"

METRIC_NAME               VALUE
------------------------- --------------------
ACT_ABORTED_TOTAL                            8
ACT_COMPLETED_TOTAL                     474043
ACT_REJECTED_TOTAL                           0
ACT_RQSTS_TOTAL                         490478
:                                            :
```

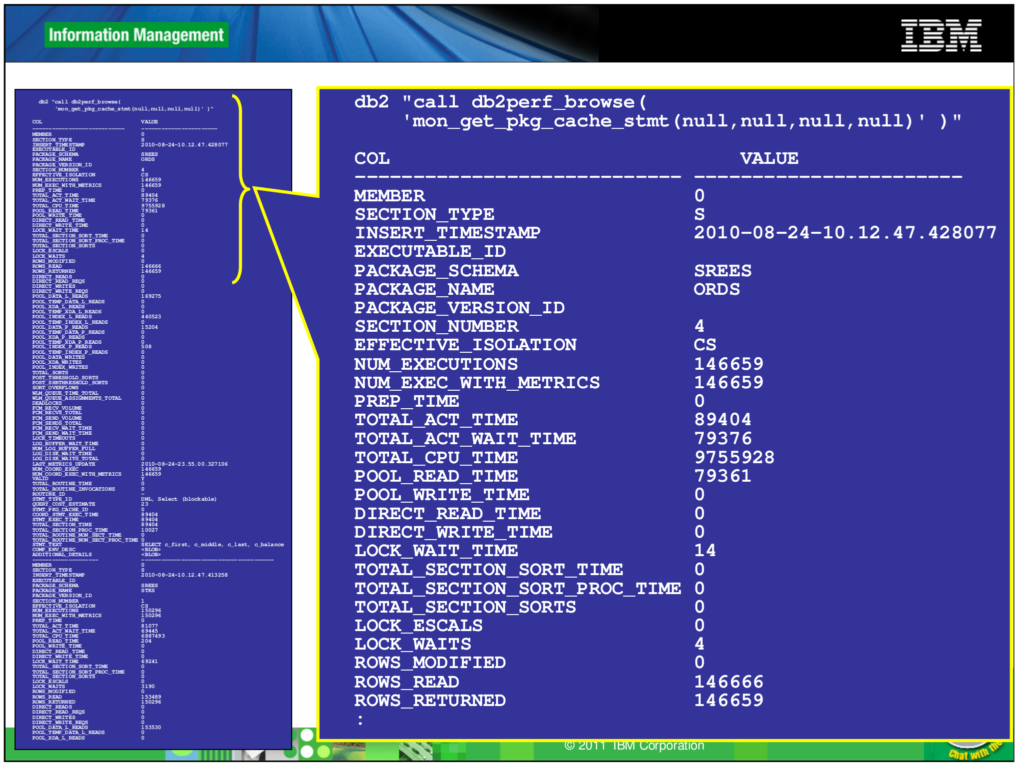
- **Can be used with any of the '`_DETAIL`' table functions**

MON_GET_CONNECTION_DETAILS
MON_GET_SERVICE_SUBCLASS_DETAILS
MON_GET_UNIT_OF_WORK_DETAILS
MON_GET_WORKLOAD_DETAILS

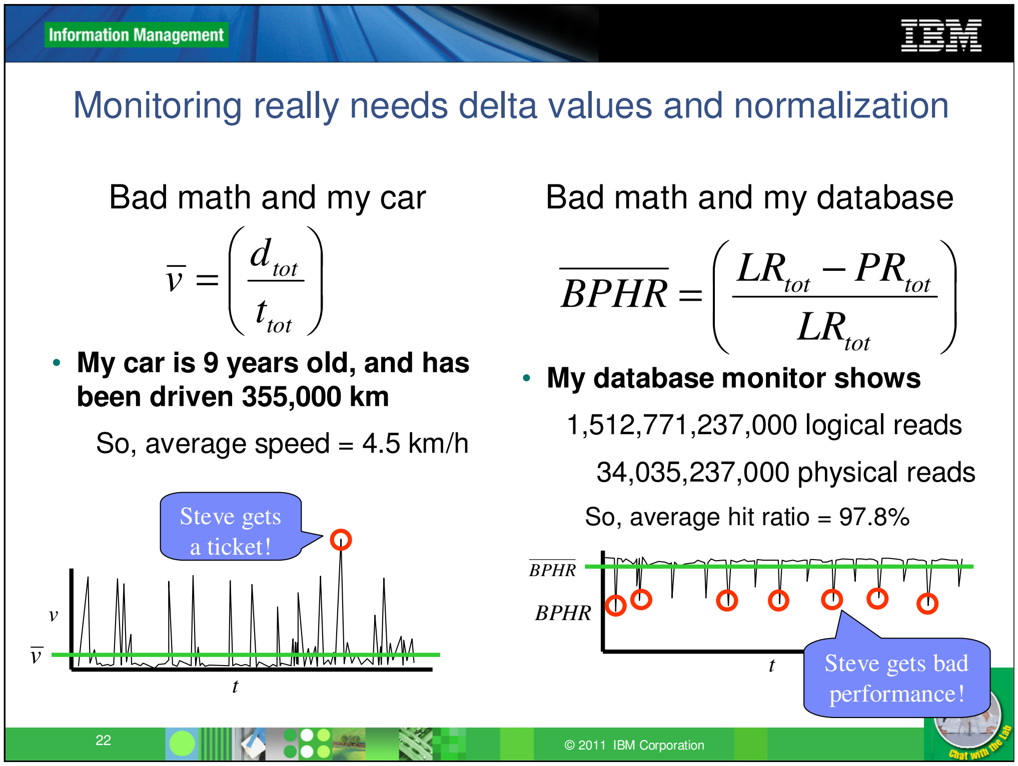


This is a very useful little tool. It comes as a SQL stored procedure which can be downloaded from IDUG Code Place (search for db2perf_browse.)

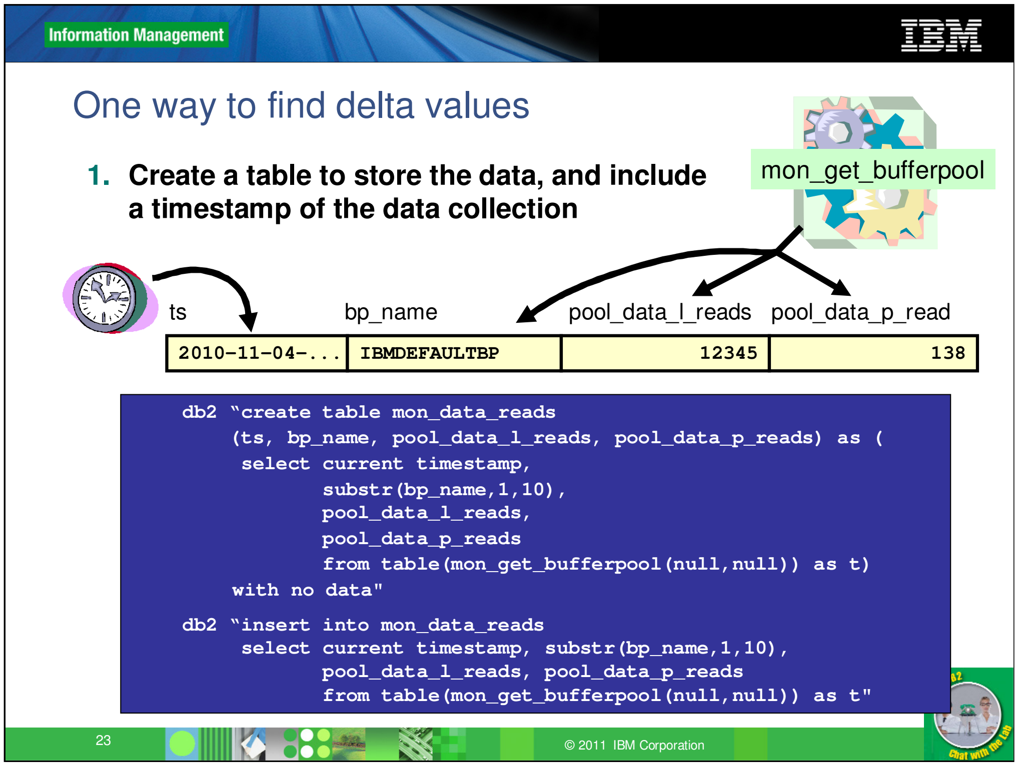
1. Run the CLP script to create the stored procedure
   db2 –td@ -f db2perf_browse.db2
2. Call db2perf browse to see column names & values of any table displayed in name/value pairs down the screen
   e.g. db2 "call db2perf_browse('mon_get_workload(null,-2)')"



This page just shows what a full-size 'browse' on mon_get_pkg_cache_stmt looks like.



Unless we get 'delta' values when we monitor, we're looking at what could be a very very long average – which might miss all the interesting intermittent stuff!



Because we're running in the database itself when we collect data, we can easily take a few steps to collect delta values instead of the usual 'unresettable' values we get from the table functions.

Basically, the idea is to bring samples of the monitor data into two tables. Note that we use CREATE .. AS to get the template table definition, and we include CURRENT TIMESTAMP to be able to tell when the data was collected

Finding delta values

2. **Use a view defined over 'before' and 'after' tables to find the delta between collections**

		ts	bp_name	pool_data_l_reads	pool_data_p_read
After:		..34.19.100	IBMDEFAULTBP	17889	202
		minus	*copy*	*minus*	*minus*
Before:		..33.17.020	IBMDEFAULTBP	12345	138
	copy	*gives*	*gives*	*gives*	*gives*
Delta:	.34.19.100	62.08	IBMDEFAULTBP	5544	64

```
db2 "create table mon_data_reads_before like mon_data_reads"
db2 "create view mon_data_reads_delta as select
     after.ts as time,
     after.ts - before.ts as delta,
     after.bp_name as bp_name,
     after.pool_data_l_reads - before.pool_data_l_reads
       as pool_data_l_reads
     from mon_data_reads        as after,
          mon_data_reads_before as before,
     where after.bp_name = before.bp_name"
```


The basic principle here is that for numeric columns, we subtract the 'Before' values from the 'After' values – based on the assumption that numerics are generally counters or times that increase in most cases. Even if they stay the same or decrease, it's still reasonable to calculate a delta in this way. For non-numeric columns, we simply use the 'After' value, to show the latest data.

See `db2perf_browse` in the appendix … sample SQL routine to find table function delta values

Finding delta values

3. **Insert monitor data into 'before' and 'after' tables, and (presto!) extract the delta using the view**

```
db2 "insert into mon_data_reads_before
     select current timestamp, substr(bp_name,1,10),
            pool_data_l_reads, pool_data_p_reads
            from table(mon_get_bufferpool(null,null)) as t"
sleep 60    # ...or whatever your favorite time span is
db2 "insert into mon_data_reads
     select current timestamp, substr(bp_name,1,10),
            pool_data_l_reads, pool_data_p_reads
            from table(mon_get_bufferpool(null,null)) as t"
db2 "select * from mon_data_reads_delta"
```

Tip - Instead of creating the 'before' and 'after' tables and 'delta' view for each query you build, do it once for the base table functions like MON_GET_WORKLOAD, etc.

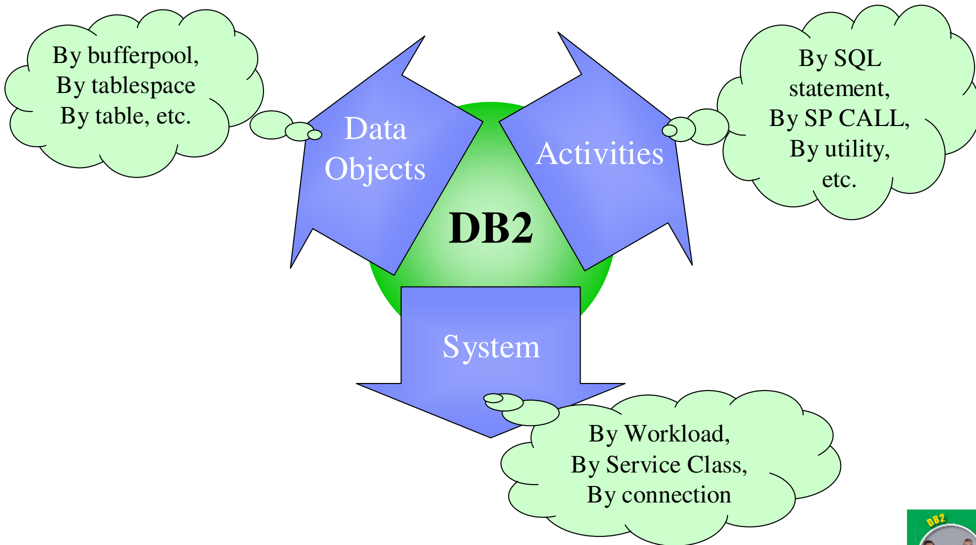
- Then custom monitor queries simply use the delta views instead of the table functions

Once we have the view over 'After' minus 'Before', all we need to do is insert data into them (with an appropriate delay between), and we automatically get the delta.

Agenda

- DB2 monitoring – DIY or GUI?
- Overview of performance diagnostics in DB2 LUW 9.7
- SQL monitoring for the snapshot addicted
- **Choices, choices … what should I pay attention to?**
  - Key operational metrics for system-level performance
  - Drilling down with statement-level monitoring
- Summary

Different angles on monitoring data in v9.7



Information Management

IBM

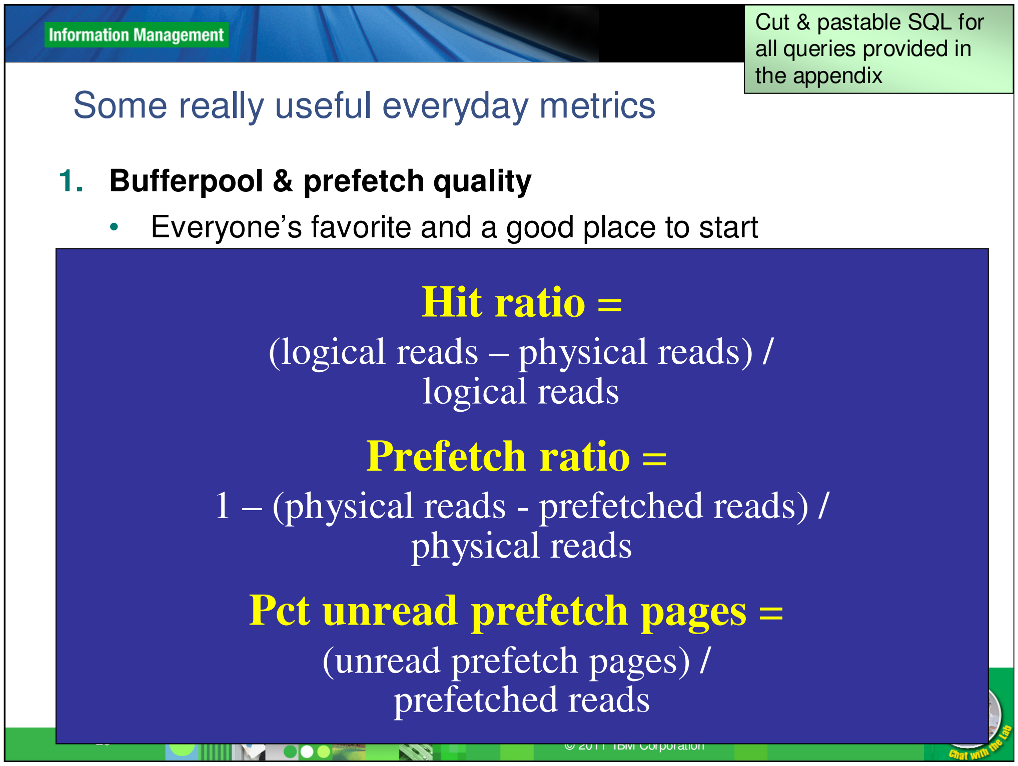
Top-level monitoring: How are things going, overall?

Snapshot for database ≈ Σ MON_GET_WORKLOAD or MON_GET_SERVICE_SUBCLASS or MON_GET_CONNECTION and MON_GET_BUFFERPOOL

- **Choose the table functions and columns which give you the monitor elements you want**
  - Sum over rows (all workloads, all service subclasses, etc.) to get a system view
  - Simpler still in a non-WLM environment
    - `db2 select * from table(mon_get_workload(null,null)) as t`
- **Augment 'system' PIT monitor table functions with other 'object' data on bufferpools, tablespaces, connections, etc.**

28 © 2011 IBM Corporation

If you're used to something like a 'Snapshot for database' in previous levels of DB2, you can obtain the same information by aggregating over the rows in *either* mon_get_workload *or* mon_get_service_subclass, *or* mon_get_connection. Plus … mon_get_bufferpool, which provides the remaining few bits of information that you could get from a snapshot.



See 'Extra Stuff' section for full SQL

Information Management

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

1. **Bufferpool & prefetch quality**
   - Everyone's favorite and a good place to start

```
select current timestamp as "Time", member,
 substr(bp_name,1,20) as "BP name",
 case when POOL_DATA_L_READS < 1000 then null else cast
    (100*(float(POOL_DATA_L_READS - POOL_DATA_P_READS)) / POOL_DATA_L_READS
 as decimal(4,1)) end as "Data H/R")),
 cast( 100 * case
       when pool_data_p_reads+pool_temp_data_p_reads
          +pool_index_p_reads+pool_temp_index_p_reads < 1000 then null else
   1.0 - ( float(pool_data_p_reads+pool_temp_data_p_reads
                 +pool_index_p_reads+pool_temp_index_p_reads)
           -float(pool_async_data_reads+pool_async_index_reads))
        /float(pool_data_p_reads+pool_temp_data_p_reads
              +pool_index_p_reads+pool_temp_index_p_reads) end
 as decimal(4,1)) as "Prefetch h/r",
 cast( 100 * case
   when pool_async_index_reads+pool_async_data_reads < 1000 then null else
   unread_prefetch_pages
      /float(pool_async_index_reads+pool_async_data_reads) end
 as decimal(4,1)) as "Pct P/F unread"
 from table(mon_get_bufferpool(null,null)) as t
 where t.bp_name not like 'IBMSYSTEMBP%'
```

30

See 'Extra Stuff' section for full SQL

Some really useful everyday metrics

1. Bufferpool & prefetch quality cont'd

- Query notes
  - Tip - timestamp included in each record
  - CASE used to avoid divide-by-zero, and filter out trivial cases
  - Index, temp and XML data for hit ratios also available (full SQL in the appendix)
  - We exclude IBMSYSTEMBP bufferpools to reduce clutter
  - Many of the same elements available in MON_GET_TABLESPACE ('object' dimension) and MON_GET_WORKLOAD ('system' dimension)
- Desired ranges

Transactional systems	Complex query systems
Data HR: 75-90% good; 90%+ great	Temp Data HR: 70-90% good; 90%+ great
Index HR: 80-95% good; 95%+ great	Temp Index HR: 80-90% good; 90%+ great
Prefetch ratio: expect to be very low	Prefetch ratio: 85-95% good; 95%+ great
Unread prefetch: N/A	Unread prefetch: 3-5% or less

Regarding trivial cases – it makes sense to avoid reporting calculated hit ratios, etc., when the numbers involved are too low to be significant. For example, with 4 logical reads and 2 physical reads, we have a hit ratio of 50%. This is low! But do we panic? No! Because the amount of expensive physical reads here is too low to be a problem.

Note that we make a distinction for transaction & complex query systems. Transactional systems can potentially have very good hit ratios, so on that side we're looking for high regular data & index hit ratios. Complex query systems often have poor hit ratios, because the data is moving through the bufferpool & may not be reread. Likewise for index pages (although they're somewhat less likely to be only read once & then leave the bufferpool.) More interesting on the complex query side is the hit ratio on temporary data and index, so we set our targets on that instead.

Note that these are just guidelines. Many systems exhibit aspects of both transaction & complex query behavior, and so we might have to blend the targets accordingly.

Some really useful everyday metrics

2. Core activity

- Transactions, statements, rows

Total # of transactions (UOW or commits)

 # activities per UOW =
Total activities / total app commits

Deadlocks / 1000 UOW =
deadlocks / total app commits

Rows read / Rows returned

IBM

Some really useful everyday metrics

2. Core activity

- Transactions, statements, rows

```
select
 current timestamp as "Timestamp",
 substr(workload_name,1,32) as "Workload",
 sum(TOTAL_APP_COMMITS) as "Total app. commits",
 sum(ACT_COMPLETED_TOTAL) as "Total activities",
 case when sum(TOTAL_APP_COMMITS) < 100 then null else
  cast( sum(ACT_COMPLETED_TOTAL) / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
  as "Activities / UOW",
 case when sum(TOTAL_APP_COMMITS) = 0 then null else
  cast( 1000.0 * sum(DEADLOCKS)/ sum(TOTAL_APP_COMMITS) as decimal(8,3)) end
  as "Deadlocks / 1000 UOW",
 case when sum(ROWS_RETURNED) < 1000 then null else
  sum(ROWS_READ)/sum(ROWS_RETURNED) end as "Rows read/Rows ret",
 case when sum(ROWS_READ+ROWS_MODIFIED) < 1000 then null else
  cast(100.0 * sum(ROWS_READ)/sum(ROWS_READ+ROWS_MODIFIED) as decimal(4,1)) end
    as "Pct read act. by rows"
 from table(mon_get_workload(null,-2)) as t
 group by rollup ( substr(workload_name,1,32) );
```

Information Management

IBM

Some really useful everyday metrics

2. Core activity

- Query notes
  - Picking up top-level metrics from MON_GET_WORKLOAD, but also works with …SERVICE_SUBCLASS and …CONNECTION
  - Use ROLLUP to get per-workload stats, plus at overall system level
  - Deadlocks don't usually happen much, so we normalize to 1000 UOW
  - Rows read / rows returned gives a feel of whether scans or index accesses dominate
- Desired ranges

	Transactional systems	Complex query systems
Total Transactions	Depends on the system…	
Activities per UOW	Typically 5-25 Beware 1 per UOW!	Low – typically 1-5
Deadlocks per 1000 UOW	Less then 5 good, under 1 great	Should be less than 1
Rows read / rows selected	5-20 good, 1-5 great, showing good use of indexes	Usually quite high due to use of scans

Rollup is handy here as a powerful & simple GROUP BY – it gives us information per workload, plus 'rolled up' to the overall total.

Normalization is important, since it removes the need to make sure all our monitoring intervals are exactly the same. Sometimes we normalize 'per transaction' – but for rare things like deadlocks, we normalize by longer term things, like 'per 1000 transactions'

Information Management

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

3. Disk I/O performance

- Count & time of tablespace I/Os, log I/Os

BP physical I/O per UOW =
Total BP reads + writes / total app commits

milliseconds per BP physical I/O =
Total BP read + write time / total BP reads + writes

Direct I/O per UOW =
Total Direct reads + writes / total app commits

milliseconds per 8 Direct I/Os (4kB) =
Total Direct read + write time / total Direct reads + writes

Log I/O per UOW =
Total Log reads + writes / total app commits

milliseconds per Log I/O =
Total Log read + write time / total Log reads+writes

35

See 'Extra Stuff' section for full SQL

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

3. Disk I/O performance

- Count & time of tablespace I/Os, log I/Os

```
select
  current timestamp,
  substr(workload_name,1,24) as "Workload",
  case when sum(TOTAL_APP_COMMITS) < 100 then null else
    cast( float(sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                    POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS))
              / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "BP phys rds / UOW",
 :
 from table(mon_get_workload(null,-2)) as t
 group by rollup ( substr(workload_name,1,24) );

select
  current timestamp,
  case when COMMIT_SQL_STMTS < 100 then null else
    cast( float(LOG_WRITES) / COMMIT_SQL_STMTS as decimal(6,1)) end
    as "Log wrts / UOW",
  :
  from sysibmadm.snapdb;
```


See 'Extra Stuff' section for full SQL

Some really useful everyday metrics

3. Disk I/O performance

- Query notes
  - Picking up top-level metrics from MON_GET_WORKLOAD, but also very useful with MON_GET_TABLESPACE (see appendix for SQL)
  - Currently roll together data, index, temp physical reads, but these could be reported separately (along with XDA)
    - Breaking out temporary reads/writes separately is a good idea
  - We separate
    - Bufferpool reads (done by agents and prefetchers)
    - Bufferpool writes (done by agents and page cleaners)
    - Direct reads & writes (non-bufferpool, done by agents)
  - Direct IOs are counted in 512-byte sectors in the monitors
    - We multiply out to calculate time per 4K bytes (8 sectors)
  - Transaction log times are available in **`MON_GET_WORKLOAD.LOG_DISK_WAIT_TIME`** & friends but lower level values from **`SYSIBMADM.SNAPDB.LOG_WRITE_TIME_S`** & friends are more precise

The LOG_DISK_WAIT_TIME metric in MON_GET_WORKLOAD measures some additional pathlength, etc. – more than just the IO. In the current level, SNAPDB.LOG_WRITE_TIME is generally more accurate.

Some really useful everyday metrics

3. Disk I/O performance

- Desired / typical ranges

	Transactional systems	Complex query systems
Physical IO per UOW	**Typically quite small e.g. less than 5 but depends on the system**	**Async data & index reads, especially from temp, are generally very high**
ms per bufferpool read	**Random: under 10 ms good, under 5ms great**	**Sequential: under 5 ms good, under 2 ms great**
ms per bufferpool write	**Random: under 8 ms good, under 3 ms great**	**Sequential temps: under 6 ms good, under 3 ms great**
ms per 4KB of direct I/O	**Direct I/Os are typically in much larger chunks than 4KB Reads: under 2 ms good, under 1 ms great Writes: under 4 ms good, under 2 ms great**	
ms per log write	**Typically: under 6 ms good, under 3 ms great Large log operations (e.g. bulk inserts, etc.) can take longer**	

Information Management

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

4. **'Computational' performance**
   - Sorting, SQL compilation, commits, catalog caching, etc.

Pct of sorts which spilled =
spilled sorts / total sorts

Pct of total processing time in sorting

Pct of total processing in SQL compilation

Package Cache hit ratio =
(P.C. lookups – P.C. inserts) / P.C. lookups

See 'Extra Stuff' section for full SQL

Information Management

Cut & pastable SQL for all queries provided in the appendix

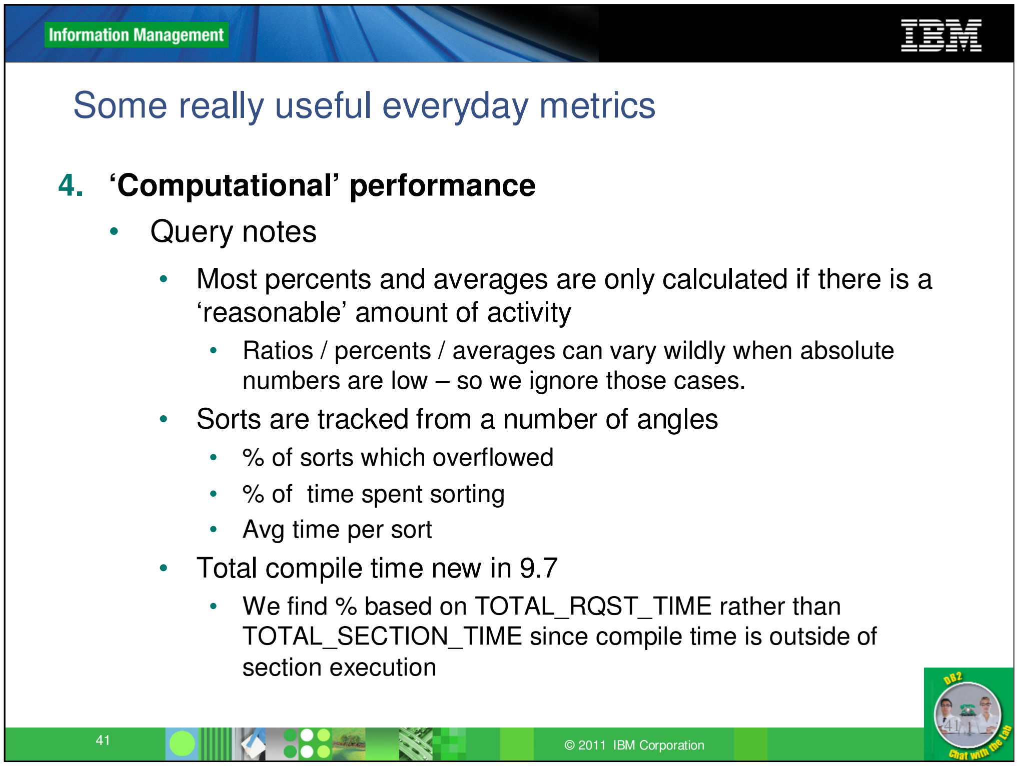
Some really useful everyday metrics

4. 'Computational' performance

- Sorting, SQL compilation, commits, catalog caching, etc.

```
select current timestamp as "Timestamp",
       substr(workload_name,1,32) as "Workload", ...

 case when sum(TOTAL_SECTION_SORTS) < 1000 then null else cast(
  100.0 * sum(SORT_OVERFLOWS)/sum(TOTAL_SECTION_SORTS)
    as decimal(4,1)) end as "Pct spilled sorts",

 case when sum(TOTAL_SECTION_TIME) < 100 then null else cast(
  100.0 * sum(TOTAL_SECTION_SORT_TIME)/sum(TOTAL_SECTION_TIME)
    as decimal(4,1)) end as "Pct section time sorting",

 case when sum(TOTAL_SECTION_SORTS) < 100 then null else cast(
  100.0 * sum(TOTAL_SECTION_SORT_TIME)/sum(TOTAL_SECTION_SORTS)
    as decimal(6,1)) end as "Avg sort time",

 case when sum(TOTAL_RQST_TIME) < 100 then null else cast(
  100.0 * sum(TOTAL_COMPILE_TIME)/sum(TOTAL_RQST_TIME)
    as decimal(4,1)) end as "Pct request time compiling",

 case when sum(PKG_CACHE_LOOKUPS) < 1000 then null else cast(
  100.0 * sum(PKG_CACHE_LOOKUPS-PKG_CACHE_INSERTS)/sum(PKG_CACHE_LOOKUPS)
    as decimal(4,1)) end as "Pkg cache h/r",

 case when sum(CAT_CACHE_LOOKUPS) < 1000 then null else cast(
  100.0 * sum(CAT_CACHE_LOOKUPS-CAT_CACHE_INSERTS)/sum(CAT_CACHE_LOOKUPS)
    as decimal(4,1)) end as "Cat cache h/r"
```


See 'Extra Stuff' section for full SQL



Compile time is a great new metric in 9.7. Previously, it was quite difficult to find out how much time was being spent in statement compilation. Note that with the new metrics, statement compilation comes outside of section execution (must compile before we execute!), so in terms of finding a percent of time, we use TOTAL_RQST_TIME rather than TOTAL_SECTION_TIME instead. The DB2 Information Center has a good description of the hierarchy of timing elements here -

http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.mon.doc/doc/c0055434.html

Information Management

IBM

Some really useful everyday metrics

4. 'Computational' performance

- Desired / typical ranges

	Transactional systems	Complex query systems
Percent of sorts spilled	**Usually low, but high % not a worry unless sort time is high too**	**Large sorts typically spill, so fraction could be 50% or more**
Percent of time spent sorting	**Usually < 5%. More than that? look at indexing**	**Usually < 25%. More than that? Look at join types & indexing**
Average sort time	**Needs to be less than desired tx response time / query response time. Drill down by statement**	
Percent of time spent compiling	**< 1% - expect few compiles, and simple ones when they occur**	**< 10% - very complex queries & high optimization can drive this up, but still not dominating. Much higher than 10? Maybe optlevel is too high?**
Pkg cache hit ratio	**> 98%**	**Can be very low (e.g. < 25%)**
Cat cache hit ratio	**> 95%**	**> 90%**

42

A high percentage of spilled sorts isn't necessarily something to worry about, unless we're spending a lot of time doing it.

Regarding compilation & package cache hits, it's generally the case that transactional systems generally do less on-the-fly compilation than complex query systems, so we tend to have more aggressive goals about the amount of time we spend compiling, etc. Compilation drives the greater activity we see in the package cache & catalog cache, which tends to drive down the hit ratios there.

Information Management

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

5. Wait times

- New in 9.7 – where are we spending non-processing time?

Total wait time

Pct of time spent waiting =

Total wait time / Total request time

Breakdown of wait time into types

… lock wait time
… bufferpool I/O time
… log I/O time
… communication wait time

Count of lock waits, lock timeouts, deadlocks

See 'Extra Stuff' section for full SQL

Information Management

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

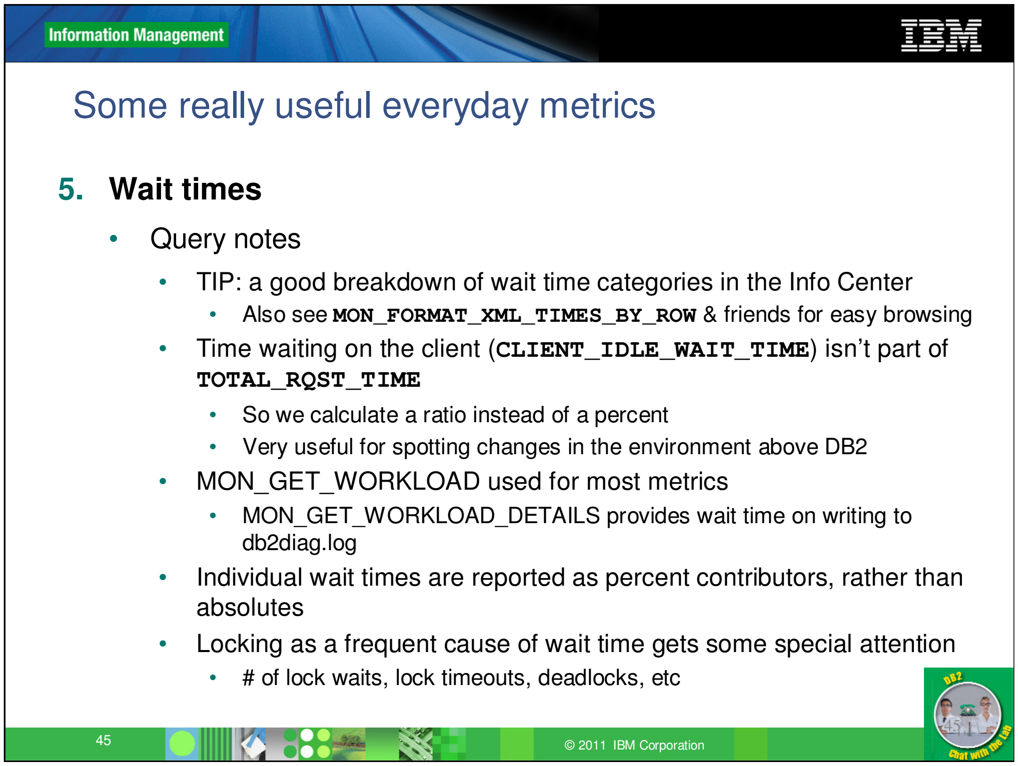
5. Wait times

- New in 9.7 – where are we spending non-processing time?

```
select current timestamp as "Timestamp",substr(workload_name,1,32) as "Workload",
 sum(TOTAL_RQST_TIME) as "Total request time",
 sum(CLIENT_IDLE_WAIT_TIME) as "Client idle wait time",
 case when sum(TOTAL_RQST_TIME) < 100 then null else
  cast(float(sum(CLIENT_IDLE_WAIT_TIME))/sum(TOTAL_RQST_TIME) as decimal(10,2)) end
    as "Ratio of client wt to request time",
 case when sum(TOTAL_RQST_TIME) < 100 then null else
  cast(100.0 * sum(TOTAL_WAIT_TIME)/sum(TOTAL_RQST_TIME) as decimal(4,1)) end
    as "Wait time pct of request time",
 case when sum(TOTAL_WAIT_TIME) < 100 then null else
      cast(100.0*sum(LOCK_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
          as "Lock wait time pct of Total Wait",
... sum(POOL_READ_TIME+POOL_WRITE_TIME)/ ... as "Pool I/O pct of Total Wait",
... sum(DIRECT_READ_TIME+DIRECT_WRITE_TIME)/ ... as "Direct I/O pct of Total Wait",
... sum(LOG_DISK_WAIT_TIME)/ ... as "Log disk wait pct of Total Wait",
... sum(TCPIP_RECV_WAIT_TIME+TCPIP_SEND_WAIT_TIME)/ ... as "TCP/IP wait pct ...",
... sum(IPC_RECV_WAIT_TIME+IPC_SEND_WAIT_TIME)/ ... as "IPC wait pct of Total Wait",
... sum(FCM_RECV_WAIT_TIME+FCM_SEND_WAIT_TIME)/ ... as "FCM wait pct of Total Wait",
... sum(WLM_QUEUE_TIME_TOTAL)/ ... as "WLM queue time pct of Total Wait",
... sum(XML_DIAGLOG_WRITE_WAIT_TIME)/ ... as "diaglog write  pct of Total Wait"
```

44

See 'Extra Stuff' section for full SQL



Great breakdown of wait time in the Info Center at

http://publib.boulder.ibm.com/infocenter/db2luw/v9r7/index.jsp?topic=/com.ibm.db2.luw.admin.mon.doc/doc/c0055434.html

Why is client_idle_wait_time used in a ratio instead of a percent? Because it's not contained within total_rqst_time (rather, it's between requests.) So we still do basically the same calculation (finding a quotient), except that it can be greater than 100% or 1x.

One interesting new metric comes from MON_GET_WORKLOAD_DETAILS, which provides time spent writing to db2diag.log. This is rarely a problem, but it's a good thing to keep track of, in case issues crop up which start causing lots of writes there.

Information Management

Some really useful everyday metrics

5. **Wait times**
   - Desired / typical ranges

	Transactional systems	Complex query systems
Ratio of client idle time to request time	**Highly variable, but can be quite high (2-10x) depending on layers above DB2**	**Generally quite low on a heavily loaded system**
Wait time pct of request time	**Typically 10-30%, depending on system load & tuning**	**Typically 20-40% depending on system load & tuning**
Disk I/O time pct of total wait time	**60-80% - usually quite high if other factors like lock & log wait are reasonably under control**	
Lock wait time pct of total wait time	**10% or less; if higher than 20-30%, look into CURRENTLY COMMITTED & friends**	**Typically very low**
Log disk wait pct of total wait time	**Low-med – if above 20%, tuning logs is required**	**Very low – less than 5%**

46

Client idle time is likely to be higher if there are real end-users attached to the system. However, if application servers are used, the connections tend to drive the database much more constantly, and thereby keep the idle time lower.

Note that the last 3 – disk I/O wait time, lock wait time & log disk wait time, are reported as a percent of **total wait time** – not of total request time. So we could have only 10% wait time, but 80% (0.8, or 8% in absolute terms) of that might be disk IO wait.

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

6. Per-statement SQL performance data for drilling down

- Looking for SQL that need to go on a diet …

Top 20 statements

… by CPU time & elapsed time

… by rows read & sort activity

… by wait time breakdown

Cut & pastable SQL for all queries provided in the appendix

Some really useful everyday metrics

6. Per-statement SQL performance data for drilling down

- Looking for SQL that need to go on a diet …

```
select MEMBER, TOTAL_ACT_TIME, TOTAL_CPU_TIME,
  (TOTAL_CPU_TIME+500)/1000 as "TOTAL_CPU_TIME (ms)",
  TOTAL_SECTION_SORT_PROC_TIME,
  NUM_EXEC_WITH_METRICS, substr(STMT_TEXT,1,40) as stmt_text
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  order by TOTAL_CPU_TIME desc fetch first 20 rows only;

select ROWS_READ, ROWS_RETURNED,
  case when ROWS_RETURNED = 0 then null
    else ROWS_READ / ROWS_RETURNED end as "Read / Returned",
  TOTAL_SECTION_SORTS, SORT_OVERFLOWS, TOTAL_SECTION_SORT_TIME,
  case when TOTAL_SECTION_SORTS = 0 then null
    else TOTAL_SECTION_SORT_TIME / TOTAL_SECTION_SORTS end as "Time / sort",
  NUM_EXECUTIONS, substr(STMT_TEXT,1,40) as stmt_text ...

select TOTAL_ACT_TIME, TOTAL_ACT_WAIT_TIME, LOCK_WAIT_TIME,
  FCM_SEND_WAIT_TIME+FCM_RECV_WAIT_TIME as "FCM wait time",
  LOCK_TIMEOUTS, LOG_BUFFER_WAIT_TIME, LOG_DISK_WAIT_TIME,
  TOTAL_SECTION_SORT_TIME-TOTAL_SECTION_SORT_PROC_TIME as "Sort wait time",
  NUM_EXECUTIONS, substr(STMT_TEXT,1,40) as stmt_text ...
```

Some really useful everyday metrics

6. Per-statement SQL performance data for drilling down

- Query notes
  - Proper ranges are tricky to identify
    - Usually decide if there's a problem using higher-level data
  - Related metrics are grouped together into queries
    - Activity, CPU and wait time
    - Row counts & sorts
    - Getting everything at once works too, but results get pretty wide
  - For bufferpool query, we order by descending total physical disk reads
    - Hit ratio is interesting, but physical reads are where the time goes
  - It can be useful to have the same query multiple times with different **ORDER BY** clauses
    - E.g. once each by activity time and CPU time
    - Due to FETCH FIRST n ROWS clause, you can get different row sets

With most of the previous PIT metrics, we've been looking at a high level. Here, generally after we've found a problem at a higher level, we drill down to the statement level, looking for which statements have similar symptoms. So we basically look at the same queries as for the system level.

Some really useful everyday metrics

6. Per-statement SQL performance data for drilling down

- Query notes cont'd
  - Times are in milliseconds
    - Microseconds of CPU time is also reported in ms in the query
  - Total counts/times per statement are most important, but getting per-execution values can be helpful too
    - Digging out from under boulders or grains of sand requires different tools!
  - Tip: overlaps between different sort orders (and different queries) help identify the most interesting statements!
  - Tip: there is a constant flow of statements through the package cache
    - Pull all of **`MON_GET_PKG_CACHE_STMT`** out to a separate table for querying to get consistent raw data from query to query

Almost all the times we collect are in milliseconds – except CPU time, which is in microseconds. So just to be consistent, we report CPU in ms too.

It's can be useful to look at both total metrics (for all executions), and for individual executions, depending on the situation. We report both, just to cover all the bases.

We have multiple statements AND multiple sort orders. The most interesting statements tend to be the ones which come near the top of the list in multiple queries – e.g. longest running AND most physical IO, etc.

Because the queries we use are based on MON_GET_PKG_CACHE_STMT, which gets its information from the package cache, we have to pay attention to the possibility that interesting statements might flow out of the package cache before we see them. Two ways to guard against this – larger package cache, and fairly frequent querying, pulling records out of the table function and storing them in a table for ongoing analysis.

PIT information summarized with monitor views

- **DB2 9.7 provides several administrative views which pull summary & highlight information from the monitor table functions**
- **Good for quick command-line queries**
  - No parameters to pass
  - Basic derived metrics (e.g. hit ratio, I/O time, wait time percentages) already provided
- **Tip: for best accuracy, use delta monitor values & calculate derived metrics in your queries**

Admin view – `sysibmadm.xxx`	Short description
`MON_DB_SUMMARY`	**Overall database activity; detailed wait time breakdown; total BP hit ratio**
`MON_CURRENT_SQL`	**CPU & activity stats for all currently executing SQL**
`MON_LOCKWAITS`	**List of details on current lock waits – item being locked, participants, statements, etc.**
`MON_BP_UTILIZATION`	**I/O stats including hit ratio, etc., for all bufferpools**
`MON_PKG_CACHE_SUMMARY`	**Per-statement information, mostly in terms of averages vs. totals;**

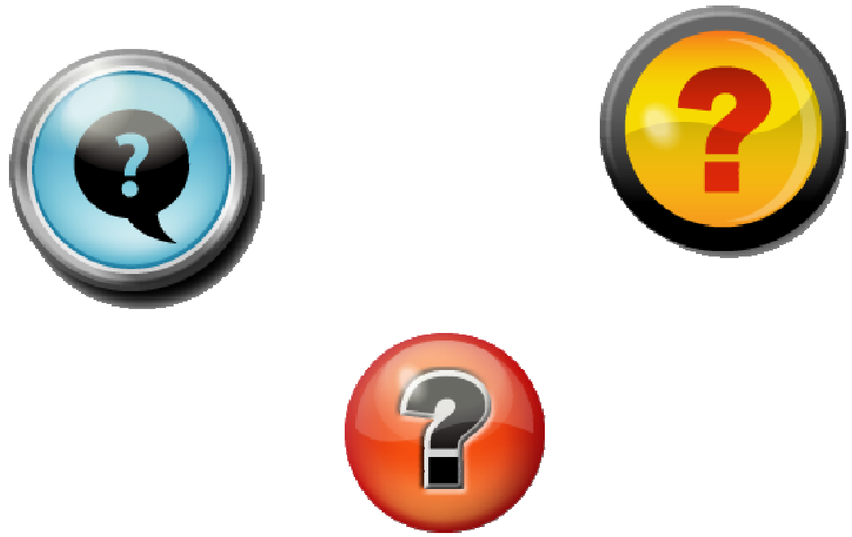
Summary

- **DIY or GUI - DB2 & OPM have the performance monitoring bases covered**
  - Watch for an upcoming Chat dedicated to OPM Best Practices
- **DB2 9.7 brings many big improvements in monitoring**
  - Component processing & wait times
  - Static SQL / SQL procedure monitoring
  - Improvements in low-overhead activity monitoring
  - Transitioning from snapshots to SQL monitoring with the MONREPORT module

Summary cont'd

- **A good set of monitor queries makes diagnosing problems much easier**
  - Choosing & tracking the most important metrics
  - Calculating derived values (hit ratio, time per IO, etc.)
  - Comparing against baseline values
- **A typical performance workflow based on DB2 9.7 metrics**
  1. PIT system metrics
     - CPU & disk utilization
  2. PIT top-level metrics
     - Bufferpool quality, prefetching, tablespace metrics, package cache, catalog cache, etc.
  3. PIT statement-level metrics
     - Similar to system-level, but broken down per statement & per statement execution

> Questions



Thank You!

ibm.com/db2/labchats

Thank you for attending!

Extra stuff

- **Cut & paste SQL source for all queries**

- **`db2perf_delta`**
  - SQL procedure to calculate deltas for monitor table function output

- **Steve's DB2 performance blog at IDUG.org**

  `http://www.idug.org/blogs/steve.rees/index.html`

Cut & paste queries – Bufferpool & Prefetch (p. 29)

```
Select current timestamp as "Time",
  member,
  substr(bp_name,1,20) as bp_name,
  case when POOL_DATA_L_READS < 1000 then null else
  cast (100*(float(POOL_DATA_L_READS - POOL_DATA_P_READS)) / POOL_DATA_L_READS as decimal(4,1)) end
    as "Data H/R" ,
  case when POOL_INDEX_L_READS < 1000 then null else
  cast (100*(float(POOL_INDEX_L_READS - POOL_INDEX_P_READS)) / POOL_INDEX_L_READS as decimal(4,1)) end
    as "Index H/R" ,
  case when POOL_TEMP_DATA_L_READS < 1000 then null else
  cast (100*(float(POOL_TEMP_DATA_L_READS - POOL_TEMP_DATA_P_READS)) / POOL_TEMP_DATA_L_READS as decimal(4,1)) end
    as "Temp Data H/R",
  case when POOL_TEMP_INDEX_L_READS < 1000 then null else
  cast (100*(float(POOL_TEMP_INDEX_L_READS - POOL_TEMP_INDEX_P_READS)) / POOL_TEMP_INDEX_L_READS as decimal(4,1)) end
    as "Temp Index H/R",
  case when POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS
              +POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS < 1000 then null else
  cast(100*1.0-(float(POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS+POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS)
                    - float(POOL_ASYNC_DATA_READS+POOL_ASYNC_INDEX_READS)) /
                      float(POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS+POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS)
                    as decimal(4,1)) end
  as "Prefetch Ratio",
  case when POOL_ASYNC_INDEX_READS+POOL_ASYNC_DATA_READS < 1000 then null else
  cast(100*float(UNREAD_PREFETCH_PAGES)/float(POOL_ASYNC_INDEX_READS+POOL_ASYNC_DATA_READS) as decimal(4,1)) end
  as "Pct P/F unread"
  from table(mon_get_bufferpool(null,-2)) as t where bp_name not like 'IBMSYSTEMBP%';

select current timestamp as time, member,
substr(tbsp_name,1,20) as tbsp_name,
  case when POOL_DATA_L_READS < 1000 then null else
  cast (100*(float(POOL_DATA_L_READS - POOL_DATA_P_READS)) / POOL_DATA_L_READS as decimal(4,1)) end
  as "Data H/R" ,
  case when POOL_INDEX_L_READS < 1000 then null else
  cast (100*(float(POOL_INDEX_L_READS - POOL_INDEX_P_READS)) / POOL_INDEX_L_READS as decimal(4,1)) end
  as "Index H/R" ,
  case when POOL_TEMP_DATA_L_READS < 1000 then null else
  cast (100*(float(POOL_TEMP_DATA_L_READS - POOL_TEMP_DATA_P_READS)) / POOL_TEMP_DATA_L_READS as decimal(4,1)) end
  as "Temp Data H/R",
  case when POOL_TEMP_INDEX_L_READS < 1000 then null else
  cast (100*(float(POOL_TEMP_INDEX_L_READS - POOL_TEMP_INDEX_P_READS)) / POOL_TEMP_INDEX_L_READS as decimal(4,1)) end
  as "Temp Index H/R",
  case when POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS+POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS < 1000 then null else
  cast(100 * 1.0-(float(POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS+POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS)
                     - float(POOL_ASYNC_DATA_READS+POOL_ASYNC_INDEX_READS)) /
                       float(POOL_DATA_P_READS+POOL_TEMP_DATA_P_READS+POOL_INDEX_P_READS+POOL_TEMP_INDEX_P_READS) as
    decimal(4,1)) end as "Prefetch H/R",
  case when POOL_ASYNC_INDEX_READS+POOL_ASYNC_DATA_READS < 1000 then null else
  cast(100*float(UNREAD_PREFETCH_PAGES)/float(POOL_ASYNC_INDEX_READS+POOL_ASYNC_DATA_READS) as decimal(4,1)) end
  as "Pct P/F unread"
from table(mon_get_tablespace(null,null)) as t;
```

Cut & paste queries – Disk & IO (p. 35)

```
select
  current timestamp as "Time",
  substr(workload_name,1,24) as "Workload",
  case when sum(TOTAL_APP_COMMITS) < 100 then null else
    cast( float(sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                    POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS))
              / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "BP phys rds / UOW",
  case when sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS) < 1000 then null else
    cast( float(sum(POOL_READ_TIME))
              / sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                    POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS) as decimal(5,1)) end
    as "ms / BP rd",
  case when sum(TOTAL_APP_COMMITS) < 100 then null else
    cast( float(sum(POOL_DATA_WRITES+POOL_INDEX_WRITES))
              / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "BP wrt / UOW",
  case when sum(POOL_DATA_WRITES+POOL_INDEX_WRITES) < 1000 then null else
    cast( float(sum(POOL_WRITE_TIME))
              / sum(POOL_DATA_WRITES+POOL_INDEX_WRITES) as decimal(5,1)) end
    as "ms / BP wrt",
  case when sum(TOTAL_APP_COMMITS) < 100 then null else
    cast( float(sum(DIRECT_READS))
              / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "Direct rds / UOW",
  case when sum(DIRECT_READS) < 1000 then null else
    cast( 8.0*sum(DIRECT_READ_TIME)
              / sum(DIRECT_READS) as decimal(5,1)) end
    as "ms / 8 Dir. rd (4k)",
  case when sum(TOTAL_APP_COMMITS) < 100 then null else
    cast( float(sum(DIRECT_WRITES))
              / sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "Direct wrts / UOW",
  case when sum(DIRECT_WRITES) < 1000 then null else
    cast( 8.0*sum(DIRECT_WRITE_TIME)
              / sum(DIRECT_WRITES) as decimal(5,1)) end
    as "ms / 8 Dir. wrt (4k)"

  from table(mon_get_workload(null,null)) as t
  group by rollup ( substr(workload_name,1,24) );

select
  current timestamp as "Time",
  substr(tbsp_name,1,12) as "Tablespace",
  sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                    POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS)
    as "BP rds",
  case when sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS) < 1000 then null else
    cast( float(sum(POOL_READ_TIME))
              / sum(POOL_DATA_P_READS+POOL_INDEX_P_READS+
                    POOL_TEMP_DATA_P_READS+POOL_TEMP_INDEX_P_READS) as decimal(5,1)) end
    as "ms / BP rd",
  sum(POOL_DATA_WRITES+POOL_INDEX_WRITES)
    as "BP wrt",
  case when sum(POOL_DATA_WRITES+POOL_INDEX_WRITES) < 1000 then null else
    cast( float(sum(POOL_WRITE_TIME))
              / sum(POOL_DATA_WRITES+POOL_INDEX_WRITES) as decimal(5,1)) end
    as "ms / BP wrt",
  sum(DIRECT_READS)
    as "Direct rds",
  case when sum(DIRECT_READS) < 1000 then null else
    cast( 8.0*sum(DIRECT_READ_TIME)
              / sum(DIRECT_READS) as decimal(5,1)) end
    as "ms / 8 Dir. rd (4k)",
  sum(DIRECT_WRITES)
    as "Direct wrts ",
  case when sum(DIRECT_WRITES) < 1000 then null else
    cast( 8.0*sum(DIRECT_WRITE_TIME)
              / sum(DIRECT_WRITES) as decimal(5,1)) end
    as "ms / 8 Dir. wrt (4k)"

  from table(mon_get_tablespace(null,-2)) as t
  group by rollup ( substr(tbsp_name,1,12) );

select
  current timestamp as "Time",
  case when COMMIT_SQL_STMTS < 100 then null else
    cast( float(LOG_WRITES)
              / COMMIT_SQL_STMTS as decimal(6,1)) end
    as "Log wrts / UOW",
  case when LOG_WRITES < 100 then null else
    cast( (1000.0*LOG_WRITE_TIME_S + LOG_WRITE_TIME_NS/1000000)
          / LOG_WRITES as decimal(6,1)) end
    as "ms / Log wrt",
  case when COMMIT_SQL_STMTS < 100 then null else
    cast( float(LOG_READS)
          / COMMIT_SQL_STMTS as decimal(6,1)) end
    as "Log rds / UOW",
  case when LOG_READS < 100 then null else
    cast( (1000.0*LOG_READ_TIME_S + LOG_READ_TIME_NS/1000000)
          / LOG_READS as decimal(6,1)) end
    as "ms / Log rd",
  NUM_LOG_BUFFER_FULL as "Num Log buff full"

  from sysibmadm.snapdb;
```

Cut & paste queries – Computational performance (p. 39)

```
select
 current timestamp as "Timestamp",
 substr(workload_name,1,32) as "Workload",
 sum(TOTAL_APP_COMMITS) as "Total application commits",
 sum(TOTAL_SECTION_SORTS) as "Total section sorts",
 case when sum(TOTAL_APP_COMMITS) < 100 then null else
  cast(float(sum(TOTAL_SECTION_SORTS))/sum(TOTAL_APP_COMMITS) as decimal(6,1)) end
    as "Sorts per UOW",
 sum(SORT_OVERFLOWS) as "Sort overflows",
 case when sum(TOTAL_SECTION_SORTS) < 1000 then null else
  cast(100.0 * sum(SORT_OVERFLOWS)/sum(TOTAL_SECTION_SORTS) as decimal(4,1)) end
    as "Pct spilled sorts",

 sum(TOTAL_SECTION_TIME) as "Total section time",
 sum(TOTAL_SECTION_SORT_TIME) as "Total section sort time",
 case when sum(TOTAL_SECTION_TIME) < 100 then null else
  cast(100.0 * sum(TOTAL_SECTION_SORT_TIME)/sum(TOTAL_SECTION_TIME) as decimal(4,1)) end
    as "Pct section time sorting",

 case when sum(TOTAL_SECTION_SORTS) < 100 then null else
  cast(100.0 * sum(TOTAL_SECTION_SORT_TIME)/sum(TOTAL_SECTION_SORTS) as decimal(6,1)) end
    as "Avg sort time",

 sum(TOTAL_RQST_TIME) as "Total request time",
 sum(TOTAL_COMPILE_TIME) as "Total compile time",
 case when sum(TOTAL_RQST_TIME) < 100 then null else
  cast(100.0 * sum(TOTAL_COMPILE_TIME)/sum(TOTAL_RQST_TIME) as decimal(4,1)) end
    as "Pct request time compiling",

 case when sum(PKG_CACHE_LOOKUPS) < 1000 then null else
  cast(100.0 * sum(PKG_CACHE_LOOKUPS-PKG_CACHE_INSERTS)/sum(PKG_CACHE_LOOKUPS) as decimal(4,1)) end
    as "Pkg cache h/r",

 case when sum(CAT_CACHE_LOOKUPS) < 1000 then null else
  cast(100.0 * sum(CAT_CACHE_LOOKUPS-CAT_CACHE_INSERTS)/sum(CAT_CACHE_LOOKUPS) as decimal(4,1)) end
    as "Cat cache h/r"

from table(mon_get_workload(null,-2)) as t
group by rollup ( substr(workload_name,1,32) );
```

Cut & paste queries – Wait times (p. 43)

```
with workload_xml as (
        select
         substr(wlm.workload_name,1,32) as XML_WORKLOAD_NAME,
         case when sum(detmetrics.TOTAL_WAIT_TIME) < 1 then null else
          cast(100.0*sum(detmetrics.DIAGLOG_WRITE_WAIT_TIME)/sum(detmetrics.TOTAL_WAIT_TIME) as decimal(4,1)) end
            as XML_DIAGLOG_WRITE_WAIT_TIME
        FROM TABLE(MON_GET_WORKLOAD_DETAILS(null,-2)) as wlm,
        XMLTABLE (XMLNAMESPACES( DEFAULT 'http://www.ibm.com/xmlns/prod/db2/mon'),
                   '$detmetrics/db2_workload' PASSING
                   XMLPARSE(DOCUMENT wlm.DETAILS)
                   as "detmetrics"
        COLUMNS "TOTAL_WAIT_TIME" INTEGER PATH 'system_metrics/total_wait_time',
                "DIAGLOG_WRITE_WAIT_TIME" INTEGER PATH 'system_metrics/diaglog_write_wait_time'
        ) AS DETMETRICS
        group by rollup ( substr(workload_name,1,32) )
)

select
 current timestamp as "Timestamp",
 substr(workload_name,1,32) as "Workload",

 sum(TOTAL_RQST_TIME) as "Total request time",
 sum(CLIENT_IDLE_WAIT_TIME) as "Client idle wait time",
 case when sum(TOTAL_RQST_TIME) < 100 then null else
  cast(float(sum(CLIENT_IDLE_WAIT_TIME))/sum(TOTAL_RQST_TIME) as decimal(10,2)) end
    as "Ratio of clnt wt to Total Rqst",

 sum(TOTAL_WAIT_TIME) as "Total wait time",
 case when sum(TOTAL_RQST_TIME) < 100 then null else
  cast(100.0 * sum(TOTAL_WAIT_TIME)/sum(TOTAL_RQST_TIME) as decimal(4,1)) end
    as "Wait time pct of request time",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(LOCK_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "Lock wait time pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(POOL_READ_TIME+POOL_WRITE_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "Pool I/O       pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(DIRECT_READ_TIME+DIRECT_WRITE_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "Direct I/O     pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(LOG_DISK_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "Log disk wait  pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(TCPIP_RECV_WAIT_TIME+TCPIP_SEND_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "TCP/IP wait    pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(IPC_RECV_WAIT_TIME+IPC_SEND_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "IPC wait       pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(FCM_RECV_WAIT_TIME+FCM_SEND_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "FCM wait       pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(WLM_QUEUE_TIME_TOTAL)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "WLM queue time pct of Total Wait",

 case when sum(TOTAL_WAIT_TIME) < 100 then null else
  cast(100.0*sum(XML_DIAGLOG_WRITE_WAIT_TIME)/sum(TOTAL_WAIT_TIME) as decimal(4,1)) end
    as "diaglog write  pct of Total Wait",

 sum(LOCK_WAITS) as "Lock waits",
 sum(LOCK_TIMEOUTS) as "Lock timeouts",
 sum(DEADLOCKS) as "Deadlocks",
 sum(LOCK_ESCALS) as "Lock escalations"

 from table(mon_get_workload(null,-2)) as t, workload_xml
 where t.workload_name = workload_xml.xml_workload_name
 group by rollup ( substr(workload_name,1,32) );
```

IBM

Cut & paste queries – Per-statement PIT (p. 47)

```
select
  MEMBER,
  ROWS_READ,
  ROWS_RETURNED,
  case when ROWS_RETURNED = 0 then null
    else ROWS_READ / ROWS_RETURNED end as "Read / Returned",
  TOTAL_SECTION_SORTS,
  SORT_OVERFLOWS,
  TOTAL_SECTION_SORT_TIME,
  case when TOTAL_SECTION_SORTS = 0 then null
    else TOTAL_SECTION_SORT_TIME / TOTAL_SECTION_SORTS end as "Time / sort",
  NUM_EXECUTIONS,
  substr(STMT_TEXT,1,40) as stmt_text
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  order by rows_read desc fetch first 20 rows only;

select
  MEMBER,
  TOTAL_ACT_TIME,
  TOTAL_CPU_TIME,
  (TOTAL_CPU_TIME+500) / 1000 as "TOTAL_CPU_TIME (ms)",
  TOTAL_SECTION_SORT_PROC_TIME,
  NUM_EXECUTIONS,
  substr(STMT_TEXT,1,40) as stmt_text
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  order by TOTAL_CPU_TIME desc fetch first 20 rows only;

select
  MEMBER,
  TOTAL_ACT_TIME,
  TOTAL_ACT_WAIT_TIME,
  LOCK_WAIT_TIME,
  FCM_SEND_WAIT_TIME+FCM_RECV_WAIT_TIME as "FCM wait time",
  LOCK_TIMEOUTS,
  LOG_BUFFER_WAIT_TIME,
  LOG_DISK_WAIT_TIME,
  TOTAL_SECTION_SORT_TIME-TOTAL_SECTION_SORT_PROC_TIME as "Sort wait time",
  NUM_EXECUTIONS,
  substr(STMT_TEXT,1,40) as stmt_text
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  order by TOTAL_ACT_WAIT_TIME desc fetch first 20 rows only;


select
  MEMBER,
  ROWS_READ / NUM_EXEC_WITH_METRICS as "ROWS_READ / exec",
  ROWS_RETURNED / NUM_EXEC_WITH_METRICS as "ROWS_RETURNED / exec",
  case when ROWS_RETURNED = 0 then null
    else ROWS_READ / ROWS_RETURNED end as "Read / Returned",
  TOTAL_SECTION_SORTS / NUM_EXEC_WITH_METRICS as "TOTAL_SECTION_SORTS / exec",
  SORT_OVERFLOWS / NUM_EXEC_WITH_METRICS as "SORT_OVERFLOWS / exec",
  TOTAL_SECTION_SORT_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_SECTION_SORT_TIME / exec",
  case when TOTAL_SECTION_SORTS = 0 then null
    else TOTAL_SECTION_SORT_TIME / TOTAL_SECTION_SORTS end as "Time / sort",
  NUM_EXEC_WITH_METRICS,
  substr(STMT_TEXT,1,40) as STMT_TEXT
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  where NUM_EXEC_WITH_METRICS > 0
  order by ROWS_READ / NUM_EXEC_WITH_METRICS desc fetch first 20 rows only;

select
  MEMBER,
  TOTAL_ACT_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_ACT_TIME / exec",
  TOTAL_CPU_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_ACT_TIME / exec",
  (TOTAL_CPU_TIME+500) / NUM_EXEC_WITH_METRICS / 1000 as "TOTAL_CPU_TIME / exec (ms)",
  TOTAL_SECTION_SORT_PROC_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_SECTION_SORT_PROC_TIME / exec",
  NUM_EXEC_WITH_METRICS,
  substr(STMT_TEXT,1,40) as STMT_TEXT
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  where NUM_EXEC_WITH_METRICS > 0
  order by TOTAL_CPU_TIME / NUM_EXEC_WITH_METRICS desc fetch first 20 rows only;


select
  MEMBER,
  TOTAL_ACT_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_ACT_TIME / exec",
  TOTAL_ACT_WAIT_TIME / NUM_EXEC_WITH_METRICS as "TOTAL_ACT_WAIT_TIME / exec",
  LOCK_WAIT_TIME / NUM_EXEC_WITH_METRICS as "LOCK_WAIT_TIME / exec",
  (FCM_SEND_WAIT_TIME+FCM_RECV_WAIT_TIME) / NUM_EXEC_WITH_METRICS as "FCM wait time / exec",
  LOCK_TIMEOUTS / NUM_EXEC_WITH_METRICS as "LOCK_TIMEOUTS / exec",
  LOG_BUFFER_WAIT_TIME / NUM_EXEC_WITH_METRICS as "LOG_BUFFER_WAIT_TIME / exec",
  LOG_DISK_WAIT_TIME / NUM_EXEC_WITH_METRICS as "LOG_DISK_WAIT_TIME / exec",
  (TOTAL_SECTION_SORT_TIME-TOTAL_SECTION_SORT_PROC_TIME) / num_executions as "Sort wait time / exec",
  NUM_EXEC_WITH_METRICS,
  substr(STMT_TEXT,1,40) as STMT_TEXT
  from table(mon_get_pkg_cache_stmt(null,null,null,-2)) as t
  where NUM_EXEC_WITH_METRICS > 0
  order by TOTAL_ACT_WAIT_TIME / NUM_EXEC_WITH_METRICS desc fetch first 20 rows only;
```

Information Management

Available for download
from IDUG Code Place
`http://www.idug.org/table/code-place/index.html`

`db2perf_delta` –
SQL procedure to build delta views automatically

- **Procedure name: `db2perf_delta`**
- **Arguments**
  - **`tabname`** – name of table function or admin view or table providing data
  - **`keycolumn`** (optional) – name of column providing values to match rows
    - (e.g. **`bp_name`** for **`mon_get_bufferpool`**)
    - Tip - db2perf_delta 'knows' the appropriate delta column for most DB2 monitor table functions!
- **Result set output**
  - SQL statements to create tables & views
  - Error / warning messages if any
- **Side-effects**
  - Working tables created in default schema: ***<tabname>*`_capture`**, ***<tabname>*`_before`**, ***<tabname>*`_after`**, **`db2perf_log`**
  - Delta view created in default schema: ***<tabname>*`_delta`**

62

Because this is a fairly common requirement, I wrote a SQL stored procedure to produce the required 'Before' and 'After' tables, and the 'Delta' view, given any input table or table function.

It can be downloaded with instructions from IDUG Code Place at **http://www.idug.org/table/code-place/index.html**

IBM

Example of building a delta view with `db2perf_delta`

```
# Creating the view ...
db2 "call db2perf_delta('mon_get_bufferpool(null,null)')”

# Populating with the first two monitor samples 60s apart
db2 "insert into mon_get_bufferpool_capture
       select * from table(mon_get_bufferpool(null,null)) as t”

sleep 60
db2 "insert into mon_get_bufferpool_capture
       select * from table(mon_get_bufferpool(null,null)) as t”

# Finding the rate of data logical reads over the 60 seconds
db2 `select ts,ts_delta,
     substr(bp_name,1,20) as "BP name",
     pool_data_p_reads/ts_delta as "Data PR/s",
     pool_data_l_reads/ts_delta as "Data LR/s"
     from mon_get_bufferpool_delta’

TS                   TS_DELTA  BP name      Data PR/s Data LR/s
-------------------- --------- ------------ --------- ---------
...-09.41.21.824270         60 IBMDEFAULTBP      3147     48094
:
```


Note that once we have the delta view, we can select it all, or parts of it, or join it with some other table(s) , etc.