In this installment of the 'performance FAQs' series, we shine a light on some new areas, like how to sort out the subtleties of DB2 monitoring terminology, and how to get Application Snapshot information with the new monitor table functions. And what about pureScale - how is monitoring and tuning it different from single-partition DB2 ESE and multi-partition DPF? Should you create indexes before loading a table, or vice versa? What's the best way to find out which of your indexes aren't being used? And what are these 'latch' things, and how much do I have to worry about them? Join us for an hour of useful and engaging tips & insider information about DB2 performance.

We're going to tackle a variety of performance questions…

- Breaking down some everyday terms used in the performance monitor – like 'logical read' and 'activity' vs. 'request'…
- I like the 'snapshot for applications' – how do I get all that information now?
- How much impact do unused indexes have and how do I find them?
- Should I create indexes before or after LOAD?
- pureScale: what's different about performance vs. ESE, and how performance critical is the interconnect that's used?
- What are latches and should I worry about them?

1. What do monitoring terms like 'logical read' mean?

- Most metrics have an obvious, natural interpretation
  - Physical reads – we go to disk to get a page
  - Rows read – number of rows read from perm or temp tables
- Simple interpretation of a logical read:
  - DB2 goes to the buffer pool to read a page (could drive a physical read if the page isn't there)
- More correct: a logical read means DB2 is 'fixing' a page in the BP
  - Operations like a scan of all rows or all index keys in the page only require one 'fix'
  - Once we're done with a page we 'unfix' it to make the memory available for other data
- Logical reads are part of DB2 infrastructure and are used for more than reading user data

Why do we care about logical reads?

- They are a good portable, generic measure of 'work done' in DB2
  - Higher numbers mean more work going on
  - Expensive plans tend to do lots more logical reads than cheaper ones
- They are independent of whether access is to the table or index, whether row or column organized tables, etc.
- Caveat:
  - Other things beside row & index key access drive logical reads. DB2 uses the same mechanism to access internal metadata
    - space map pages (SMPs)
    - extent map pages (EMPs)
    - table control blocks (TCBs)

Calling Dr. Heisenberg

- Selecting information from MON_GET_xxx via SQL will cause some logical reads itself
  - Retro monitoring tip: snapshots don't do this…
- Tip: see the big picture! It's always best to average logical reads over many executions of a statement to factor out 'noise', rather than just one execution
  - First execution of a statement from CLP will probably cause SQL compilation & catalog access, which can bloat LR counts
  - Even a physical disk read can cause an extra (meta data) logical read

Activity vs. Request – which to pay more attention to?

- In snapshots, DB2 just reports things in terms of statements
- Monitor table functions talk about 'activities' and 'requests'

	Requests	Activities
Simple interpretation	Requests are calls to the DB2 SQL engine from applications	Activities represent internal processing of SQL statements or LOAD
More correct interpretation	There is roughly a 1:1 mapping between requests and activities (and statements), plus a request for every PREPARE, OPEN, DESCRIBE, FETCH, CLOSE on SELECTs, and COMMITs	One activity occurs for every DML, DDL, CALL and LOAD statement, **plus one activity from each statement in SQL stored procedures and each SMP and MPP subsection**

- Basic rule-of-thumb: the correlation with SQL statements means activity counts & times are usually more relevant than requests

2. Should I create indexes before or after LOAD?

- There are operational questions here
  - For very big tables with lots of indexes, it's common to LOAD without indexes
  - This keeps peak resource needs down, and to make it easier to restart if there's a problem
- But we'll focus mainly on which is faster …

- LOAD builds multiple indexes in parallel
  - During a single pass of the input data, keys from each row are inserted into all indexes
  - Particularly helpful for large tables and large numbers of indexes (provided enough temp space exists)
  - But each index is built by a single DB2 engine thread
- CREATE INDEX parallelizes creation of a single index
  - Typically up to 6 subagents work together, scanning rows and inserting keys into the new index
  - Prior to DB2 10 this was only available when INTRA_PARALLEL was enabled (but this had query plan implications, so wasn't typically used)

General tips on when to build indexes

- These days, the performance difference is quite small!
- For large tables, especially with several indexes, consider creating indexes before load
  - Here, IO is usually a bigger factor than CPU
  - CREATE INDEX CPU parallelism is outweighed by LOAD's ability to do everything in one pass of reading the table
  - **'But' #1** – make sure SHEAPTHRES_SHR divided by SORTHEAP is at least as large as the number of indexes you want to build
    - It is often better to have a smaller SORTHEAP and maintain this ratio
  - **'But' #2** – creating indexes *after* LOAD provides the opportunity for a step-by-step approach vs. all-or-nothing
    - Potentially smaller peak memory usage & ability to restart/resume if necessary
- For smaller tables & fewer indexes, where most or all of the table fits in the bufferpool, separate create index can be faster
  - If it gets down to a CPU foot-race, CREATE INDEX will usually win
  - But – make sure you have enough cores in the instance / partition to allow parallelism

Under the covers

- For LOAD, **`db2 list utilities show detail`** shows what phase we're in
  - The memory-intensive index 'heavy lifting' happens in LOAD

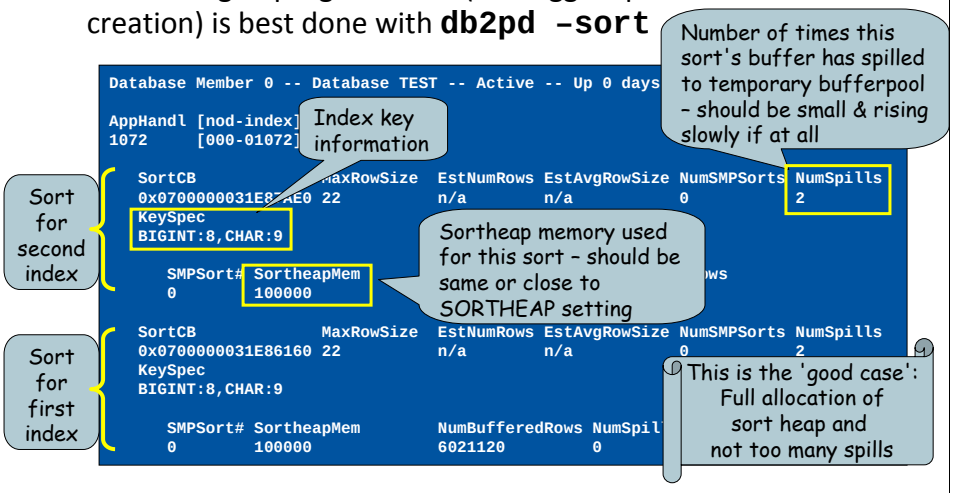
```
Database Name                    = TEST
Member Number                    = 0
Description                      = [LOADID: 14097.2013-08-16-14.24.01.926333.0 (2;13)] ...
Start Time                       = 08/16/2013 14:24:01.927663
State                            = Executing
Invocation Type                  = User
Progress Monitoring:
   Phase Number                  = 1
      Description                = SETUP
      Total Work                 = 0 bytes
      Completed Work             = 0 bytes
      Start Time                 = 08/16/2013 14:24:01.927670

   Phase Number [Current]        = 2
      Description                = LOAD
      Total Work                 = 66510899 rows
      Completed Work             = 17404136 rows
      Start Time                 = 08/16/2013 14:24:02.066185

   Phase Number                  = 3
      Description                = BUILD
      Total Work                 = 7 indexes
      Completed Work             = 0 indexes
      Start Time                 = Not Started
```

Keys are inserted into SORTHEAP-sized 'sorts' during LOAD phase

'sorts' are merged into the B-tree indexes later during the BUILD phase

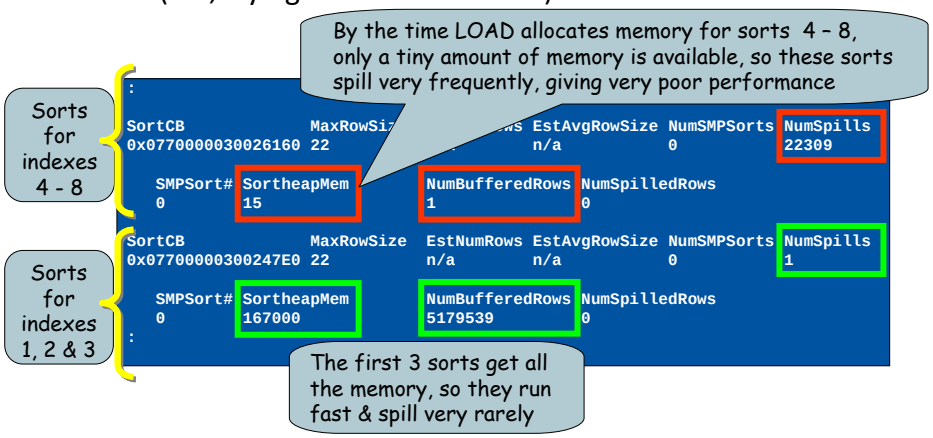
Under the covers: LOAD with indexes

- Monitoring in-progress sorts (the biggest portion of index creation) is best done with **`db2pd -sort`**



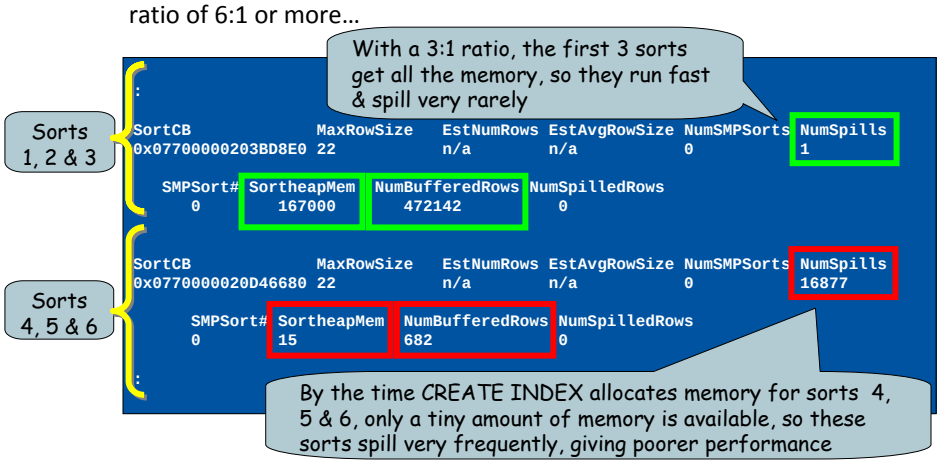
Under the covers: LOAD with indexes

- The 'bad case': SHEAPTHRES_SHR-to-SORTHEAP ratio is too small (3:1, trying to cover 8 indexes)



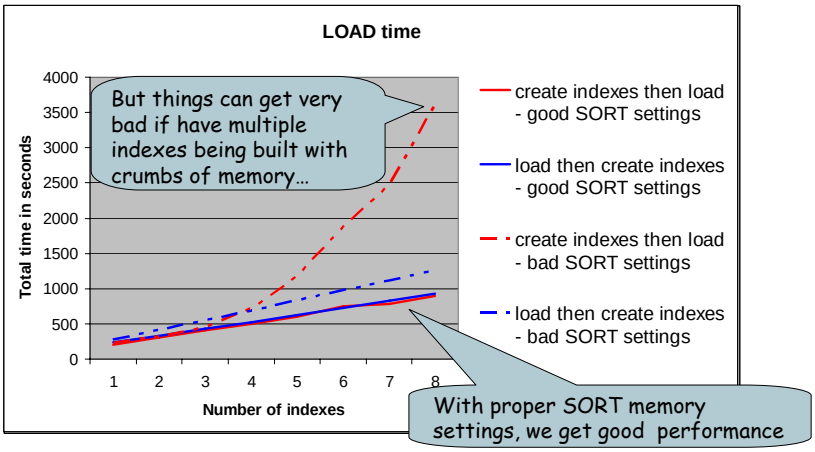
Under the covers: CREATE INDEX

- This ratio matters for CREATE INDEX too
  - Those (typically) 6 parallel streams each need a sort – so we need a ratio of 6:1 or more…



Speeds & feeds

- Test: LOAD 67M rows into a table with 8 indexes defined
  - In this case, LOAD first or create indexes first gives the same results - IF we have proper SORT settings



3. How to monitor & tune pureScale, compared to ESE?

- Most things to monitor and tune in pureScale are the same as for DB2 EE
  - CPU usage, SQL access plans, package cache activity, lock waits, deadlocks, sorting, I/O times, logging, etc.
  - The same monitoring interfaces (MON_GET_xxx table functions, monreport.dbsummary, Optim Performance Manager, etc.) are already tooled-up for pureScale
- A few things to keep an eye on in pureScale:
  1. Cluster interconnect utilization
  2. Page reclaims
  3. Group bufferpool activity
  4. CF utilization

Keep an eye on: Cluster interconnect utilization

- Why is this important?
  - Message response time between the members and the CFs is key to great pureScale performance
  - If the interconnect becomes a bottleneck, overall cluster performance will suffer
- How to monitor it?
  - MON_GET_CF_CMD returns times for the CrossInvalidate (XI) message
  - More than 20µs average XI time indicates a potential bottleneck
- What to do if there's a problem?
  - Reduce message traffic (disable autocommit? reduce isolation level?)
  - Add an extra interconnect adapter to the CF?

Keep an eye on: Page reclaims

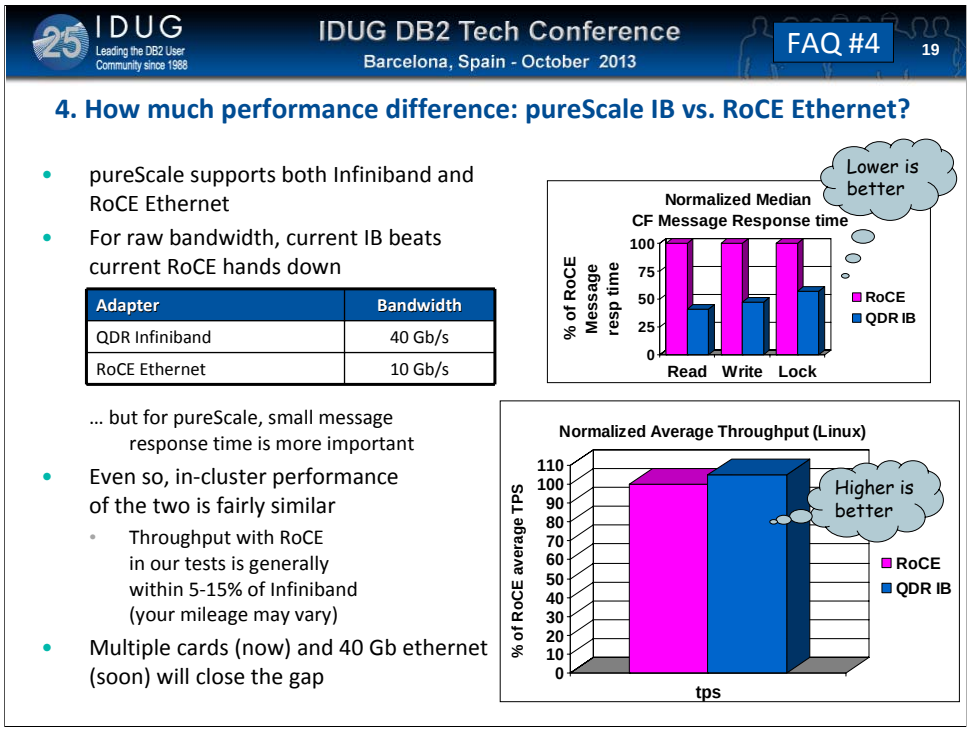
- Why is this important?
  - Preemptively moving a page from one member that's updating it to another member that needs it can help overall concurrency, but it's an expensive operation
  - In extreme cases, 'thrashing' of pages between members can hurt overall performance
- How to monitor it?
  - MON_GET_PAGE_ACCESS_INFO provides counts of different reclaim types
  - Reclaims of index pages are more common than for data / metadata pages
  - A sustained rate of more than 1 reclaim per 10 transactions may indicate a problem (rough RoT)
- What to do?
  - Range-partitioned tables with local indexes may reduce table or index page contention
  - Random indexes (DB2 10.5) or the use of CURRENT MEMBER hidden column

Keep an eye on: Group bufferpool (GBP) activity

- Why is this important?
  - The GBP contains modified pages from the committed transactions before they're written to disk, and a list of all pages in all cluster members
- How to monitor it?
  - MON_GET_BUFFERPOOL provides pureScale GBP metrics to calculate H/R
    `(pool_data_gbp_l_reads – pool_data_gbp_p_reads) / pool_data_gbp_l_reads`
    - If `pool_data_l_reads > 10 x pool_data_gbp_l_reads` then GBP isn't being used that much – don't worry about a low GBP H/R
    - If `pool_data_gbp_invalid_pages > 25% of pool_data_gbp_l_reads` then GBP is helping out, so making it big enough to get a good H/R is worthwhile
  - MON_GET_GROUP_BUFFERPOOL has one very helpful metric
    - Num_gbp_full indicates how often the GBP ran out of available pages
- What to do?
  - GBP H/R can be controlled by adjusting GBP size (RoT: target 35% of total LBP size)
  - GBP full conditions are bad – typically minimize them by increasing GBP or reducing SOFTMAX

Keep an eye on: Cluster Caching Facility (CF) utilization

- Why is this important?
  - CF response time is key for overall cluster performance
  - Interconnect is the main thing, but we also want to keep an eye on CF CPU utilization
- How to monitor it?
  - Difficulty: vmstat & other O/S level tools see ~100% CPU usage on the CF, even when the cluster is idle
  - Admin view ENV_CF_SYS_RESOURCES gives a good picture of how busy the CF really is
- What to do?
  - If `CPU_USAGE_TOTAL is more than 75 or 80%` then consider adding additional CPUs to both the primary and secondary CFs
    - Tip #1: If CF_NUM_WORKERS is not AUTOMATIC, make sure to adjust it as well, to allow for the additional CPU resource
    - Tip #2: Make sure there are 1 – 2 extra hardware threads in the CF beyond what CF_NUM_WORKERS has, to allow for other ad hoc CF processing in the case of failovers



DB2 pureScale has supported Inifiniband interconnect since day 1, and over subsequent releases we've added support for 10 Gb RoCE ethernet on both AIX and Linux. If we just look at the raw throughput, IB seems to have a huge 4x advantage – 40 Gb/s vs 10Gb/s for RoCE ethernet. Fortunately, pureScale is more dependent on small message latnecy than on raw throughput, so if we look at message response times (top right graph), we see that the latency of IB messages is only about 2x faster than RoCE ethernet. Better than 4x – but still a pretty big gap. But the real test is, what's the impact on the throughput of a real workload? After all, RDMA performance is only one part – an important part, definitely – but only one part of the whole pictures.

Fortunately, when we've compared overall throughput between IB and RoCE in the lab, we typically see a fairly small 5-15% difference between them. This gap can grow up to about 25% when the cluster is running at 100% utilization (which is higher than most customers would run it.) So overall, RoCE does quite well, and is going to be sufficiently fast for many customers. But in performance-sensitive configurations, we would still recommend IB for absolute best performance.

5. How do I get Application Snapshot content in 'new' monitoring?

- DB2 10.5 adds MON_GET_AGENT, which helps us reproduce the most useful remaining parts of Application Snapshot

1. State of all agents (which ones are running, waiting on a lock, committing, etc.)
   - Previously: select & count 'Application status' from application snapshot
   - In 10.5, MON_GET_AGENT provides finer-grained information about agents, events and requests, across several columns

Metric	Roughly what it provides (see the IC for full details!)
AGENT_TYPE	Is the agent a coordinator or subagent
AGENT_SUBTYPE	Usually indicates the kind of subagent this is (see agent_type)
EVENT_OBJECT	Most recent 'event' acted on … REQUEST, ROUTINE, LOCK, WLM_QUEUE, …
EVENT_OBJECT_NAME	The name of the LOCK or WLM_QUEUE waited on (see event_object)
EVENT_TYPE	Is the agent executing (PROCESS) or waiting for something (IDLE / ACQUIRE)
EVENT_STATE	Is the agent EXECUTING or IDLE (heavy overlap with event_type)
REQUEST_TYPE	Current or previous request by this agent: OPEN, FETCH, COMMIT, etc. There are LOTS of possible values here, depending on agent TYPE and SUBTYPE

Creating a handy 'agent status' string

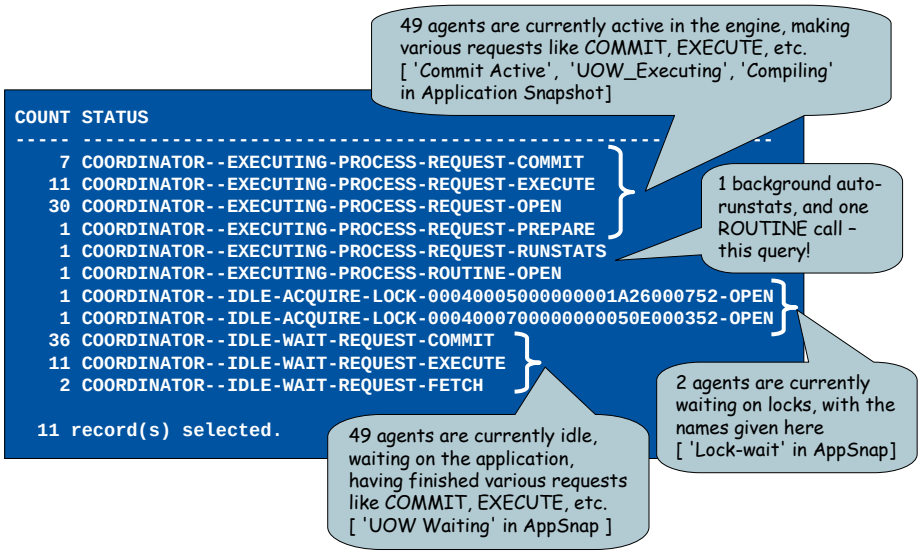
- Having all the details in separate columns is powerful, but sometimes we just want a summary we can aggregate on, like 'Application status'
  - You can customize the query to make it more or less specific

```
with agent(status) as
  (select substr( agent_type                  ||'-'||
          coalesce(agent_subtype,'')          ||'-'||
          event_state                         ||'-'||
          event_type                          ||'-'||
          event_object ||
            case when event_object in ('LOCK','WLM_QUEUE')
              then '-'|| substr(event_object_name,1,26)
              else ''
            end                               ||'-'||
          request_type ,1,80)
   from table(mon_get_agent(null,null,null,null))
   where entity = 'db2agent'
   and request_type <> 'INTERNAL0' )
select count(*) as count, status from agent group by status
```

We're excluding most 'internal' agents like prefetchers, etc.

Include the lock or WLM queue name, when we're waiting

Sample 'agent status' listing for a busy workload with 100 clients




```
COUNT STATUS
----- ------------------------------------------------------------
    7 COORDINATOR--EXECUTING-PROCESS-REQUEST-COMMIT
   11 COORDINATOR--EXECUTING-PROCESS-REQUEST-EXECUTE
   30 COORDINATOR--EXECUTING-PROCESS-REQUEST-OPEN
    1 COORDINATOR--EXECUTING-PROCESS-REQUEST-PREPARE
    1 COORDINATOR--EXECUTING-PROCESS-REQUEST-RUNSTATS
    1 COORDINATOR--EXECUTING-PROCESS-ROUTINE-OPEN
    1 COORDINATOR--IDLE-ACQUIRE-LOCK-00040005000000001A26000752-OPEN
    1 COORDINATOR--IDLE-ACQUIRE-LOCK-00040007000000000050E000352-OPEN
   36 COORDINATOR--IDLE-WAIT-REQUEST-COMMIT
   11 COORDINATOR--IDLE-WAIT-REQUEST-EXECUTE
    2 COORDINATOR--IDLE-WAIT-REQUEST-FETCH

  11 record(s) selected.
```

Digging a little deeper

2. Finding the statement that 'Lock wait' agents are running, and lock info

```
with agent as (
     select member,application_handle,uow_id,activity_id,
            substr(event_object_name,1,26) as event_object_name,executable_id
     from table (mon_get_agent(null,null,null,null)) where event_object = 'LOCK' ),
stmt as (
   select agent.application_handle,agent.member,mgpcs.executable_id,mgpcs.stmt_text
   from agent,
          table (mon_get_pkg_cache_stmt(null,agent.executable_id,null,agent.member)) as mgpcs),
lockwait as (
   select req_application_handle,lock_wait_start_time,
          substr(lock_object_type,1,16) as lock_object_type,lock_mode
   from agent, table( mon_get_appl_lockwait(agent.application_handle,agent.member ))),
lock_table_name as (
   select agent.application_handle,agent.event_object_name,
          coalesce(substr(mfln.value,1,32),'') as table_name
   from agent, table( mon_format_lock_name(agent.event_object_name) ) as mfln
   where mfln.name = 'TABNAME' )
select
   agent.application_handle, agent.uow_id, agent.activity_id,
   lockwait.lock_wait_start_time, lockwait.lock_object_type, lockwait.lock_mode,
   lock_table_name.table_name, substr(agent.event_object_name,1,26) as LOCK,
   substr(stmt.stmt_text,1,64) as STMT
   from agent, stmt, lockwait, lock_table_name
   where agent.executable_id      = stmt.executable_id
     and agent.member             = stmt.member
     and agent.application_handle = stmt.application_handle
     and agent.application_handle = lockwait.req_application_handle
     and agent.application_handle = lock_table_name.application_handle
     and agent.event_object_name  = lock_table_name.event_object_name
```

Digging a little deeper

2. Finding the statement that 'Lock wait' agents are running, and lock info

```
with agent as (
   select member,application_handle,uow_id,activity_id
   from table (mon_get_activity(null,-2)) where event_object = 'LOCK' ),
stmt as (
   select agent.application_handle,agent.member,mgpcs.executable_id,mgpcs.stmt_text
   from table (mon_get_pkg_cache_stmt(null,null,null,-2)) as mgpcs),
lockwait as (
   select req_application_handle,lock_wait_start_time
   from table (mon_get_locks(null,-2)) where lock_object_type = 'TABLE' ),
lock_table_name as (
   select agent.application_handle,agent.event_object_name,
          substr(stmt.stmt_text,1,64) as STMT_TEXT,
          substr(agent.event_object_name,1,26) as LOCK
   from agent, stmt
   where agent.executable_id = stmt.executable_id
     and agent.member = stmt.member
     and agent.application_handle = stmt.application_handle
   where mfln.name = 'TABNAME' )
select
   agent.application_handle, agent.uow_id, agent.activity_id,
   lockwait.lock_wait_start_time, lockwait.lock_object_type, lockwait.lock_mode,
   lock_table_name.table_name, substr(agent.event_object_name,1,26) as LOCK,
   substr(stmt.stmt_text,1,64) as STMT_TEXT
from agent, stmt, lockwait, lock_table_name
where agent.executable_id = stmt.executable_id
  and agent.member = stmt.member
  and agent.application_handle = stmt.application_handle
  and agent.application_handle = lockwait.req_application_handle
  and agent.application_handle = lock_table_name.application_handle
  and agent.event_object_name  = lock_table_name.event_object_name
```

Find all the agents waiting on locks …

Find the statements being run by those agents …

Get basic info about the lock requests blocking them …

Get detailed info about the lock (e.g. table name, etc.)

And return one row for each, joining on member and application_handle, etc.

Sample output

```
APPL..._HANDLE UOW_ID  ACTIVITY_ID LOCK_WAIT_START_TIME       LOCK_OBJECT_TYPE
-------..------- ------- ----------- -------------------------- ----------------
         17683     136           2 2013-08-26-11.35.06.839380 ROW
         17705     175           1 2013-08-26-11.35.06.667209 ROW
         17803     153           1 2013-08-26-11.35.06.936559 ROW
         17893     140           8 2013-08-26-11.35.06.839413 ROW
```

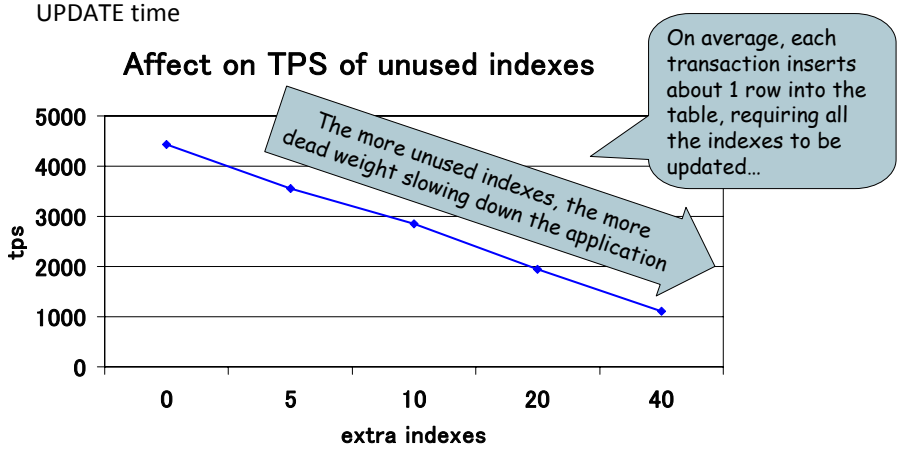
```
LOCK_MODE TABLE_NAME LOCK                         STMT
--------- ---------- ---------------------------- ------------------------------
S         CUSTOMER   00060004000000001B015000252 Update CUSTOMER set C_BALAN...
X         CUSTOMER   00060004000000005B6A000252 Select C_ID, C_FIRST from C...
X         DISTRICT   000300060000000001BD000B52 Select D_NEXT_O_ID from DIS...
S         CUSTOMER   00060004000000001B015000252 Update CUSTOMER set C_BALAN...
```

```
select substr(name,1,24) as name,substr(value,1,24) as value
from table(mon_format_lock_name('00060004000000001B015000252')),

NAME                     VALUE
------------------------ ------------------------
LOCK_OBJECT_TYPE         ROW
ROWID                    2
DATA_PARTITION_ID        0
PAGEID                   110613
TBSP_NAME                TBS_CST
TABSCHEMA                DTW
TABNAME                  CUSTOMER
```

We can go further still by pulling out more lock details via MON_FORMAT_LOCK_NAME, e.g. to get ROWID, etc. to get the content of the affected row

6. How do I identify unused indexes?

- One or two extra indexes might not be a big problem, but the more indexes you have, the more overhead there is at LOAD & INSERT / DELETE / UPDATE time



One of DB2 10's best-kept secrets: usage lists

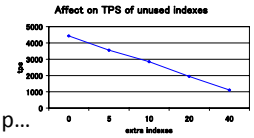
- These track the most recent uses of a table or index
  - They capture the ID of the statement(s) affecting the object
  - Query mon_get_index_usage_list to get the details

```
db2 "create usage list UL_i for index I list when full deactivate"
db2 "set usage list UL_i state active"
< workload runs for a while ... >
db2 "select * from table(mon_get_index_usage_list(null,null,null)"
```

- Notes & tips
  - Usage Lists give basic information like statement EXECUTABLE_ID and reference count at MON_OBJ_METRICS = BASE, and additional metrics at MON_OBJ_METRICS = EXTENDED
  - Use the ON FULL DEACTIVATE clause to shut down usage list once we've seen (by default) 100 different statements touching this index
  - Usage lists add some CPU & memory overhead (esp. with EXTENDED monitoring)
    - So it makes sense to use this on small groups of indexes at a time

Drilling down on our 'unused indexes' example

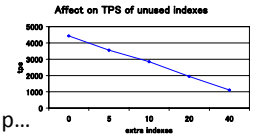
1. We add some extra unused indexes on a fairly busy table
2. We create usage lists against these indexes, plus the one index that the workload actually uses
3. We use MON_GET_INDEX_USAGE_LIST to see what shows up…




```
with ul as (
    select usagelistname,member,indschema,indname,
           num_ref_with_metrics,executable_id
    from table(mon_get_index_usage_list(null,null,null))),
  indexes as (
    select sys.indschema,sys.indname,sys.colnames,ul.executable_id
    from ul, syscat.indexes sys
    where sys.indname = ul.indname and sys.indschema = ul.indschema),
  stmt as (
    select ul.usagelistname,mgpcs.member,mgpcs.executable_id,mgpcs.stmt_text
    from ul,table(mon_get_pkg_cache_stmt(null,ul.executable_id,null,ul.member))
      as mgpcs)
select
  substr(ul.indname,1,16) as ul_indname, substr(colnames,1,24) as colnames,
  num_ref_with_metrics, substr(stmt_text,1,50) as stmt_text, ul.executable_id
from indexes, ul left outer join stmt
    on stmt.usagelistname = ul.usagelistname and stmt.member = ul.member
    and stmt.executable_id = ul.executable_id
where ul.indschema = indexes.indschema and ul.indname = indexes.indname
  and ul.executable_id = indexes.executable_id
```

Drilling down on our 'unused indexes' example

1. We add some extra unused indexes on a fairly busy table
2. We create usage lists against these indexes, plus the one index that the workload actually uses
3. We use MON_GET_INDEX_USAGE_LIST to see what shows up…




```
with ul as (
    select usagelistname, member, indschema, indname,
      from table(mon_get_index_usage_list(null,null,null))),
  indexes as (
    select sys.indschema, sys.indname, sys.colnames, ul.executable_id
      where sys.indname = ul.indname and sys.indschema = ul.indschema),
  stmt as (
    select ul.usagelistname, mgpcs.member, mgpcs.executable_id, mgpcs.stmt_text
      as mgpcs)
select
  substr(ul.indname,1,16) as ul_indname, substr(colnames,1,24) as colnames,
  num_ref_with_metrics, substr(stmt_text,1,50) as stmt_text, ul.executable_id
from
      and ul.executable_id = indexes.executable_id
```

Find all the uses of indexes we're tracking ...

Get the column definition of each index ...

Get the SQL text affecting each index ...

And return a row for each use, joining on executable_id, index name, etc.

Drilling down on our 'unused indexes' example

Currently the only index used for WHERE clauses in SELECT and UPDATE

Maintaining 6 indexes on INSERT when only 1 of them is useful on SELECT & UPDATE!

UL_INDNAME	COLNAMES	NUM_REF_...	STMT_TEXT
OLINE_IDX1	+OL_W_ID+OL_D_ID+OL_O_ID	484	Select OL_I_ID, O SUPPLY_W_
OLINE_IDX1	+OL_W_ID+OL_D_ID+OL_O_ID	542322	Select sum(OL_A NT) from O
OLINE_IDX1	+OL_W_ID+OL_D_ID+OL_O_ID	542322	Update ORDER_LI set OL_DEL
OLINE_IDX1	+OL_W_ID+OL_D_ID+OL_O_ID	2797439	Select Count(Di inct S_I_ID
OLINE_IDX1	+OL_W_ID+OL_D_ID+OL_O_ID	5548680	Insert into ORDER_LINE value
TEST_INDEX_0	+OL_NUMBER	0	Update ORDER_LINE set OL_DEL
TEST_INDEX_0	+OL_NUMBER	5548680	Insert into ORDER_LINE value
TEST_INDEX_1	+OL_NUMBER+OL_I_ID	0	Update ORDER_LINE set OL_DEL
TEST_INDEX_1	+OL_NUMBER+OL_I_ID	5548680	Insert into ORDER_LINE value
TEST_INDEX_2	+OL_NUMBER+OL_SUPPLY_W_I	0	Update ORDER_LINE set OL_DEL
TEST_INDEX_2	+OL_NUMBER+OL_SUPPLY_W_I	5548680	Insert into ORDER_LINE value
TEST_INDEX_3	+OL_NUMBER+OL_DELIVERY_D	542322	Update ORDER_LINE set OL_DEL
TEST_INDEX_3	+OL_NUMBER+OL_DELIVERY_D	5548680	Insert into ORDER_LINE value
TEST_INDEX_4	+OL_NUMBER+OL_QUANTITY	0	Update ORDER_LINE set OL_DEL
TEST_INDEX_4	+OL_NUMBER+OL_QUANTITY	5548680	Insert into ORDER_LINE value

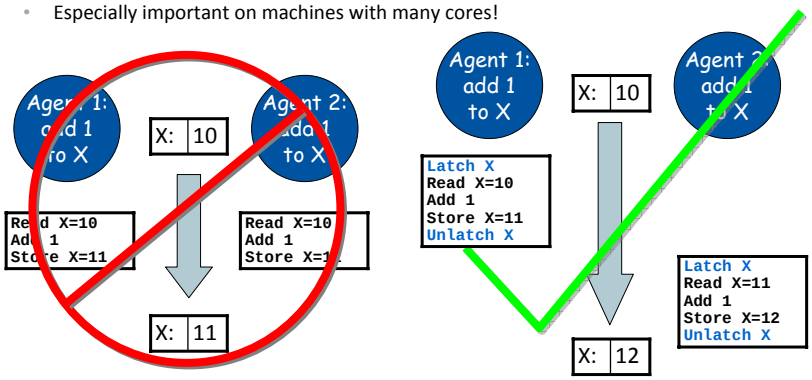
UPDATEs show up against all indexes – but only the ones with non-zero NUM_REF_WITH_METRICS really count. Here, OL_DELIVERY_ID is in the key.

Tips on hunting useless indexes with Usage Lists

- Call MON_GET_INDEX_USAGE_LIST regularly while the usage list is active
  - 'Regularly' callhing helps ensure that statements mentioned haven't been pushed out of the package cache before we harvest them
- If you're not using EXTENDED object monitoring already, you may get enough usage information without it
- Anything that's not used in a SELECT or in a WHERE clause in an UPDATE or DELETE is probably fair game
  - Indexes that only have INSERT and (especially non-zero) UPDATE activity should be top of your list
- If in doubt, check out the access plan with EXPLAIN_FROM_SECTION, using the EXECUTABLE_ID from the usage list

7. What are latches and how much do I have to worry about them?

- What it is: a latch prevents two threads from corrupting each others' work on a shared resource
  - ANY multi-thread program (like DB2) has many internal "latch-like" mechanisms to make sure internal threads work well together
  - Without latches, DB2 agents could overwrite each others' work, on things like internal monitoring counters, control blocks, etc.
  - Especially important on machines with many cores!

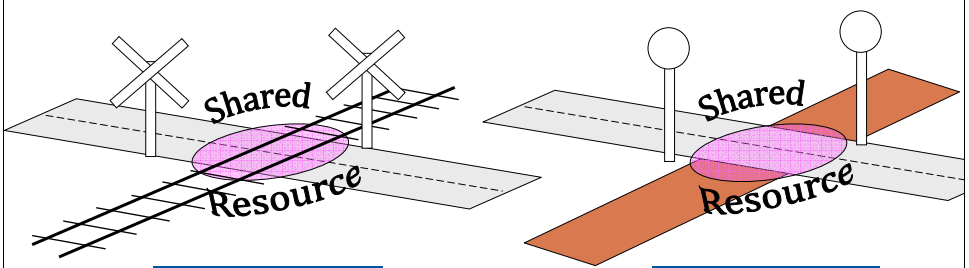


Latching basics

- What they are NOT: a latch is not a "lock" – as in what we think about in database terms, e.g. lock wait, deadlock, lock escalation, etc.
- A latch is different – it is internal to the implementation of any multi-threaded program with shared resources, like DB2
- Multiple readers of a shared resource – no problem
- A writer + anyone else (reader or writer) – Houston, we have a problem…
  - What if the reader reads the data partway through a write?

Different kinds of latches

- DB2 supports different kinds of latches, chosen for how they'll be used
  - How long is access to the shared resource needed?
  - Do we need to be completely fair based on who requested the resource first, or are there other priorities, like readers first?

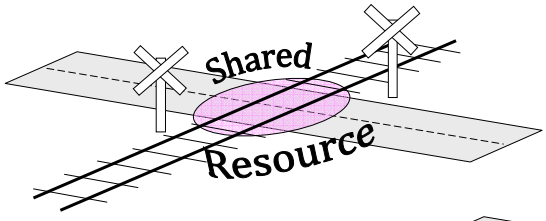


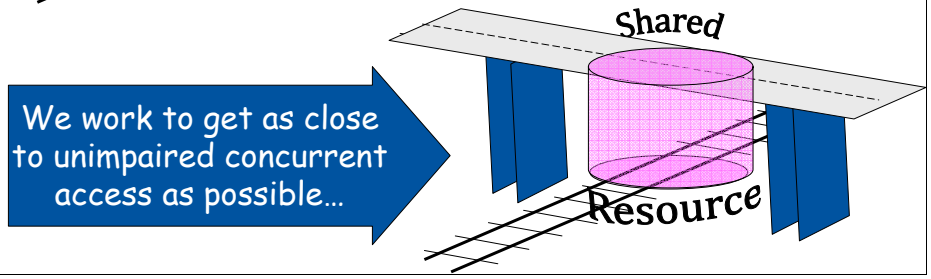
Suppose writers have priority

Suppose readers have priority

DB2's goal for latches is to make them as invisible as possible

- DB2 exploits the best latch technology native to every platform
- Every release brings improvements in latch implementation & usage





Where do you notice latches as a user of DB2?

- Heavy, persistent latch contention tends to show up as
  - Lower than expected throughput (especially at very high levels of concurrency / parallelism)
  - Higher than expected CPU usage, and long run queues
- **`db2pd -latches`** gives quick & easy (but generally not very conclusive) latch info
  - Common misunderstanding: that db2pd -latches shows *contended* latches
  - Actually shows a point-in-time view of certain *currently held* latches
    - Ie, a DB2 thread is using a shared resource - this typically changes significantly from call to all
  - Not all held latches are causing contention!
  - Tip: if you see the same latch (latch address) with a holder & many waiters, and you see this over several calls to **`db2pd -latches`**, it may indicate latch contention

```
Latches:
Address            Holder  Waiter ... LatchType
0x0780000001568440 1029    0      ... sqeWLDispatcher__m_tunerLatch
0x07700000C8453580 3543    0      ... SQLB_HASH_BUCKET_GROUP_HEADER__groupLatch
.
0x07700000C8420480 7965    14384  ... SQLB_HASH_BUCKET_GROUP_HEADER__groupLatch
0x07700000C8420480 7965    14641  ... SQLB_HASH_BUCKET_GROUP_HEADER__groupLatch
0x07700000C8420480 7965    15155  ... SQLB_HASH_BUCKET_GROUP_HEADER__groupLatch
:
```

"Extended Latch Wait"

- Shows up in MON_GET_EXTENDED_LATCH_WAIT and in some other table functions
  - 'Extended' means the requester yielded the CPU while waiting for the latch
  - i.e., it took slightly more than the 'blink of an eye' we'd normally expect – but still not necessarily something to be worried about without more evidence
  - Gets included in OTHER wait time in monreport.dbsummary
- If you suspect latch contention, look in MON_GET_WORKLOAD & friends for high portion of overall wait time accounted for by TOTAL_EXTENDED_LATCH_WAIT_TIME

```
select
  decimal(100.0*total_extended_latch_wait_time/total_wait_time,5,1)
    as PCT_LATCH
from table(mon_get_workload(null,null)) where total_wait_time > 0

PCT_LATCH
---------
     90.1
```

We're seeing some latch wait time – what now?

- In our example, we query MON_GET_EXTENDED_LATCH_WAIT to see where the wait time is accumulating

```
select
  substr(latch_name,1,32) as latch_name,
  TOTAL_EXTENDED_LATCH_WAIT_TIME
from table(mon_get_extended_latch_wait(null))

LATCH_NAME                       TOTAL_EXTENDED_LATCH_WAIT_TIME
-------------------------------- ------------------------------
SQLO_LT_NO_IDENTITY                                           3
SQLO_LT_SMemPool__MemLatchType__                              7
SQLO_LT_SQLB_HASH_BUCKET_GROUP_H                       10848050
SQLO_LT_SQLP_LHSH__hshlatch                                  15
SQLO_LT_sqlra_anchor_package__an                              1
SQLO_LT_SQLD_PAGE_CACHE__pageCac                              0
SQLO_LT_SQLB_BPD__bpdLatch_SX                              3266
SQLO_LT_SQLD_CHAIN__fullChainLat                              0
SQLO_LT_SQLD_TCB__loadInProgress                              6
SQLO_LT_sqlra_cached_env__latch                               0
SQLO_LT_sqlra_cached_var__worksp                              0
SQLO_LT_sqlra_workspace_sibling_                              1
```

Warning: cryptic names!

The two heavier hitters here are both SQLB (buffer pool) latches. This gives us some clues – for one, we can rule out SQL access plan management ("sqlra") latches

Drilling down on the example: find the most affected statements

- Knowing what statements are seeing latch contention may give us a clue to what shared resources (database objects) are hot

```
select
  substr(stmt_text,1,50) as stmt_text,
  decimal(float(total_extended_latch_wait_time)/num_executions,10,5)
    as avg_latch_time
from table(mon_get_pkg_cache_stmt(null,null,null,null))
where num_executions > 0
order by avg_latch_time desc fetch first 10 rows only

STMT_TEXT                                  AVG_LATCH_TIME
------------------------------------------ --------------
Select Count(Distinct S_I_ID) from STOCK          48.60632
Select S_QUANTITY, S_DIST_01, S_DIST_02,           0.26249
Update STOCK set S_QUANTITY = ?, S_YTD =           0.26086
Update ORDER_LINE set OL_DELIVERY_D = ?            0.00067
Select OL_I_ID, OL_SUPPLY_W_ID, OL_DELIV           0.00027
Select O_OL_CNT, O_C_ID from ORDERS wher           0.00024
Update ORDERS set O_CARRIER_ID = ? where           0.00022
Select D_NAME, D_STREET_1, D_STREET_2, D           0.00017
Insert into NEW_ORDER values (?, ?, ?)             0.00011
Insert into ORDER_LINE values (?, ?, ?,            0.00008
```

In this example, all the hot statements are operating on a table called STOCK, which we happen to notice is hit by over 75% of statement executions

What next?

- In this simple example, all hot statements are heavily biased to index access (this can be seen in monitor data and access plans)
  - This suggests some internal contention around an index root page due to the extremely heavy activity
- **General tip: you can often reduce contention by breaking up a single resource into multiple (smaller) ones**
- In this case, we look at one technique (range-partitioned tables) to break up both the table itself and the index(es) into multiple pieces

Metric	Before (with single table & index)	After (with RPT & local indexes)
System throughput	1000 tps	3500 tps
% extended latch wait	90%	14%
Hottest statement average latch wait	49ms	0.34 ms
System CPU utilization	92%	70%

A few last notes on latches & latch wait

- Latch contention may only become noticeable when a workload scales up to many connections, or runs on a system with many CPUs
- If heavy latch wait is seen (high CPU, high EXTENDED_LATCH_WAIT), look for objects (tables & especially indexes) which could be contended
  - Is it possible to break up the object to avoid contention?
- Make sure you are up-to-date on DB2 & O/S maintenance, to take advantage of the latest improvements
- In case of a serious latch problem with no obvious culprit, DB2 support can usually help out

Parting thoughts on our FAQs

- The 'logical read' metrics are most useful as large values, since this eliminates the noise from meta data and other unexpected activity
- There may not be much to choose in performance between LOAD with indexes or CREATE INDEX after LOAD – but both will benefit if you make sure SHEAPTHRES_SHR is at least 10-20x larger than SORTHEAP, or set AUTOMATIC with STMM
- Monitoring pureScale is a lot like monitoring ESE – but also keep an eye out for an over-busy interconnect, excessive page reclaims, and GBP full conditions.
- If you like SNAPSHOT FOR AGENTS, check out MON_GET_AGENT to get similar information in the new monitoring system
- Usage lists will help you sort out unused indexes from the ones pulling their weight
- Use EXTENDED_LATCH_WAIT_TIME metrics to keep an eye out for latch contention, rather than db2pd -latches

Steve Rees
IBM Canada Ltd.
srees at ca.ibm.com

Session D13

Performance FAQs volume 4 – Demystifying more mysteries!