May 6–10, 2007

San Jose Convention Center

San Jose, California, USA

Session: A04

Troubleshooting Tuning for Dynamic SQL

IDUG® 2007 North America

Thomas Baumann
Swiss Mobiliar

May 07, 2007 04:20 p.m. – 05:20 p.m.

Platform: DB2 for z/OS

GoFurther

This presentation describes a methodical approach to database tuning, which has been developed, tested and used for emergency performance tuning after an application's rollout, but has also been shown to be quite helpful for routine database monitoring. We will discuss answers to the following three questions derived from commands, metrics and tools available in each shop:

a) Are the application's critical queries being served in the most effective manner?

b) Is DB2 making optimal use of resources so as to reach the desired performance levels?

c) Are there enough primary resources available for DB2's consumption and are they configured adequately, given the current workload?

Objectives

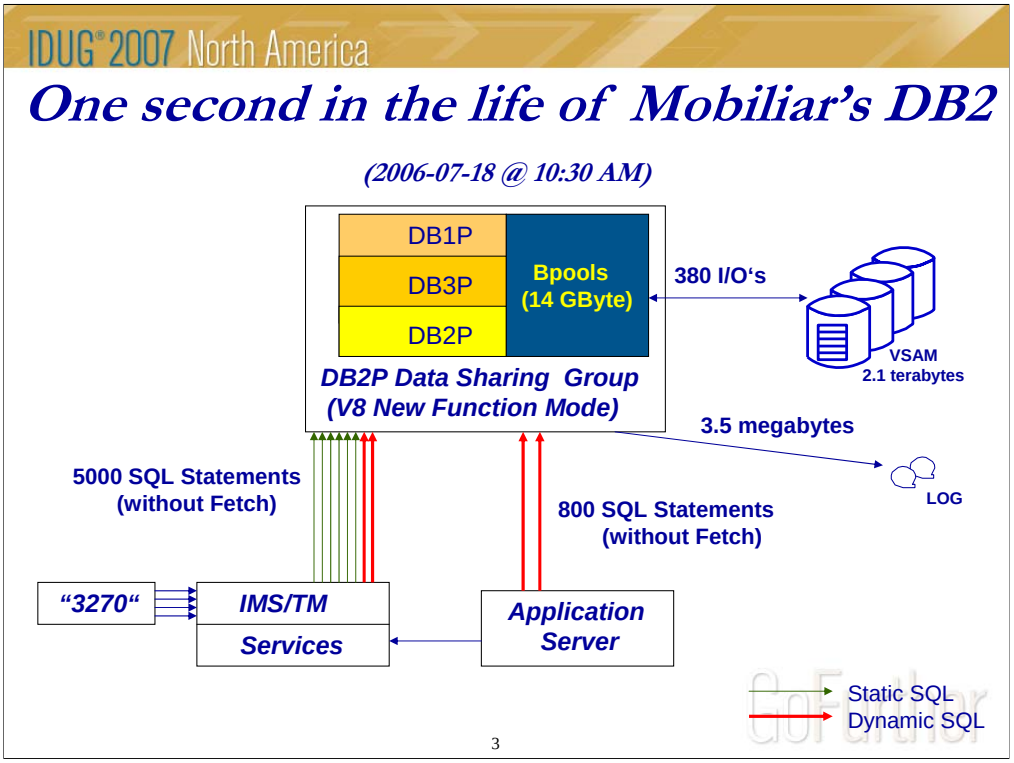
- Learn to apply a methodical tuning approach
- Hear about new performance indicators other than cache hit ratios etc.
- Find out what is specific for *dynamic* SQL tuning
- Look at new ways to trigger runstats based on your queries' performance numbers
- Detect the dynamic SQL statement cache's secrets

GoFurther

2

First, we will discuss how to identify those 'queries from hell' which could potentially monopolize your whole system. Then – with a finer granularity – we will discuss metrics and techniques to further optimize both your dynamic queries as your system as a whole.

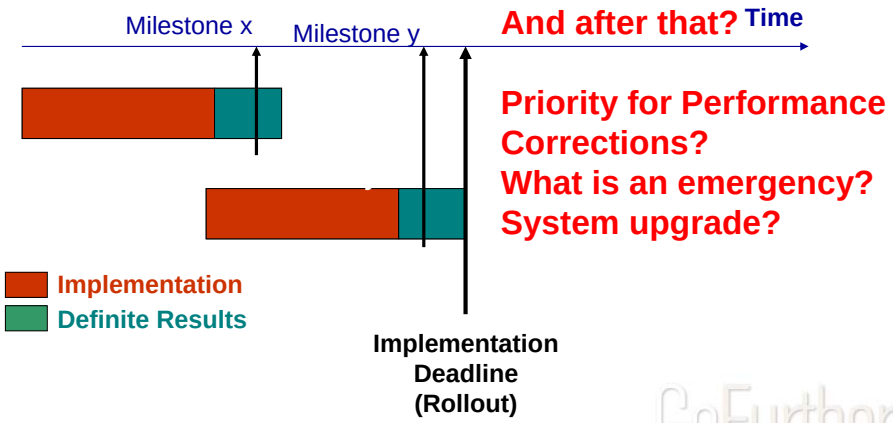
Finally, we will discuss the question of having enough system resources or if there is a need to increase your system's overall size.



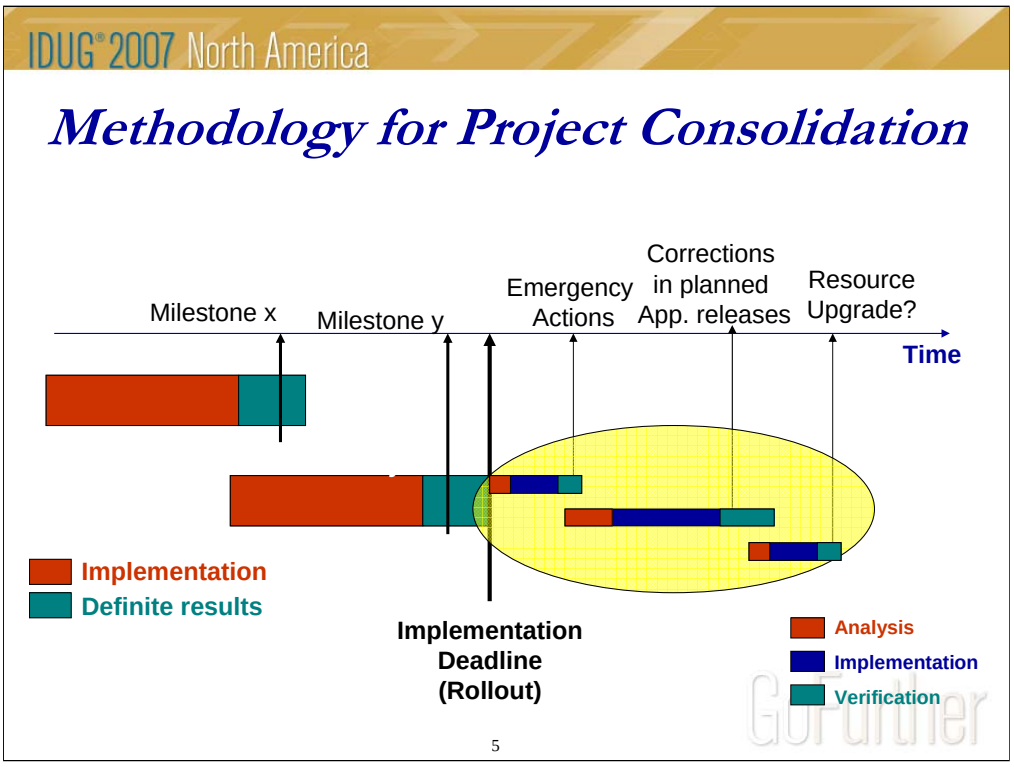
This slide shows the workload at Swiss Mobiliar's OLTP environment as measured 07/18/2006.

Methodology for Project Consolidation

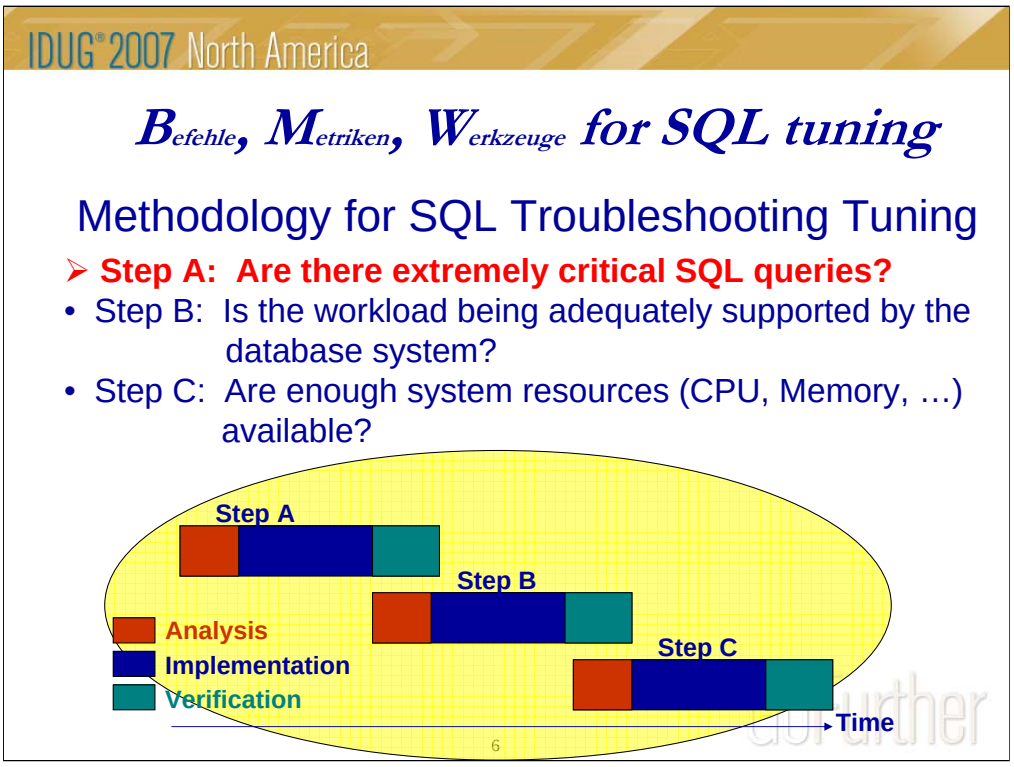
MvM (Mobiliar Action Methodology):



After the application‘s rollout, the structured project methodology ends. During project consolidation, it is often hard to get sufficient resources to fix performance problems, because priority is often given to correct functional deficiences only.

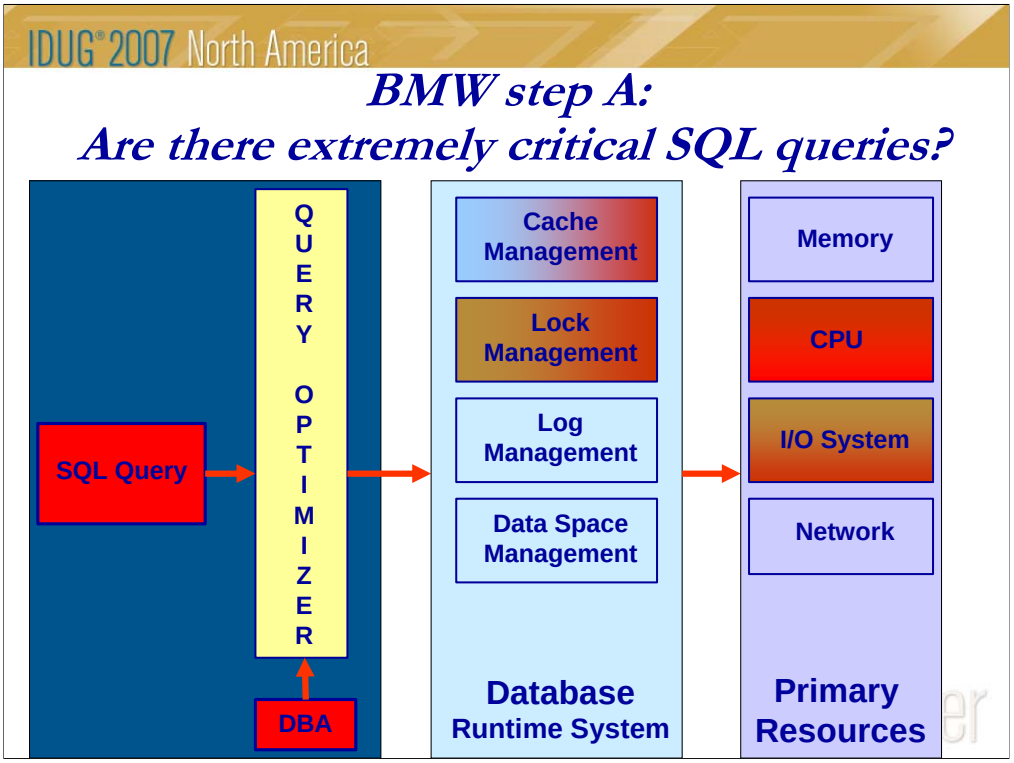


The methodology presented in this paper starts here: It defines milestones with performance metrics that need to be met. Milestones with a scope ranging from a single SQL statement to the overall performance of the application and the database subsystem. If the the criterions of the first step were missed, an emergency corrective action is automatically scheduled.



This BMW troubleshooting (or emergency database performance) tuning methodology is divided in three steps: In each of these steps, there is one single question which is analyzed in full detail, and the problems detected during this step must be resolved before entering into the next step.

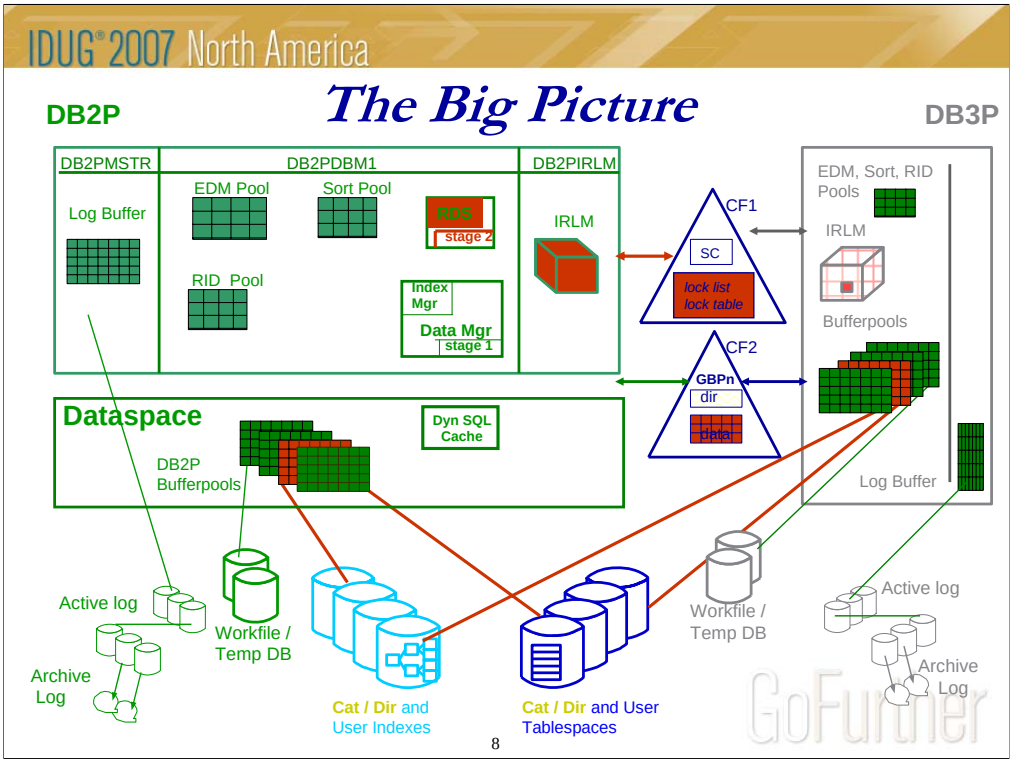
Step 1 detects extremely inefficient queries (*‚queries from hell*‘). Once they are corrected, the next steps is initiated: Different thresholds answer the question ‘a*re my queries served the optimal way?*‘. This might concern both SQL queries and database runtime system features and parameters. Only in the third step, after all the problems detected in the first two steps are fully resolved, the question *‚Do my queries get sufficient resources ?*‘ will be asked.



In a system with single very inefficient SQL statements, these might be the typical symptoms: A very high CPU usage rate, an I/O system close at the limits, bufferpools which do not reach their performance goals, and some locking problems.

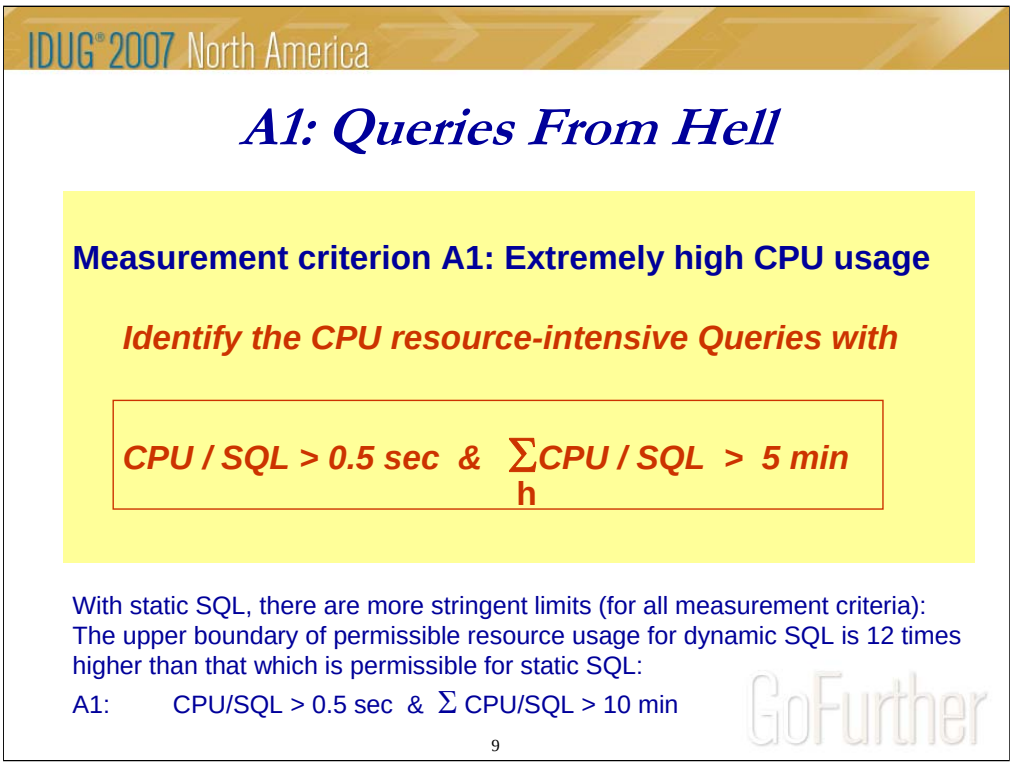
All these ‚trouble areas' are marked in red colour – the more red the more urgent they want your attention.

And all caused by a few inefficient SQL statements. That is why we start investigating this system from left to right, from SQL queries to basic resources and not vice versa.

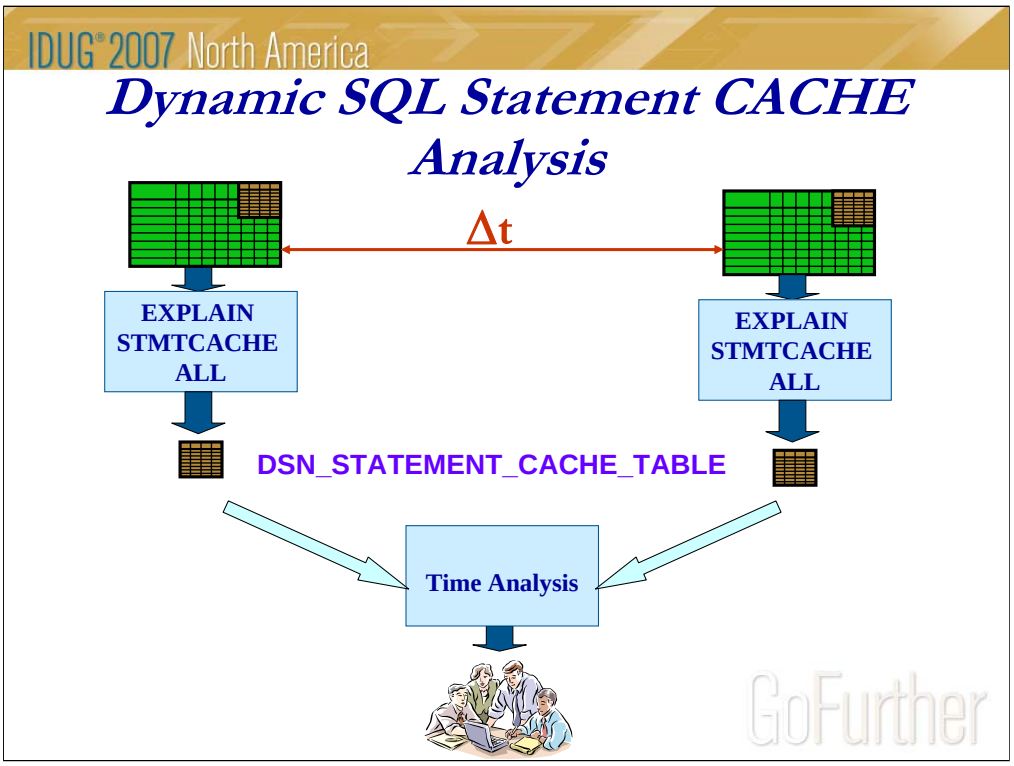


This slide contains exactly the same information as the one before – but is is more understandable for the programmer, whereas the slide before was the ‚management summary' of performance problems.

But once again – our basic problem is still the inefficient dynamic SQL statement still unknown and not yet detected.

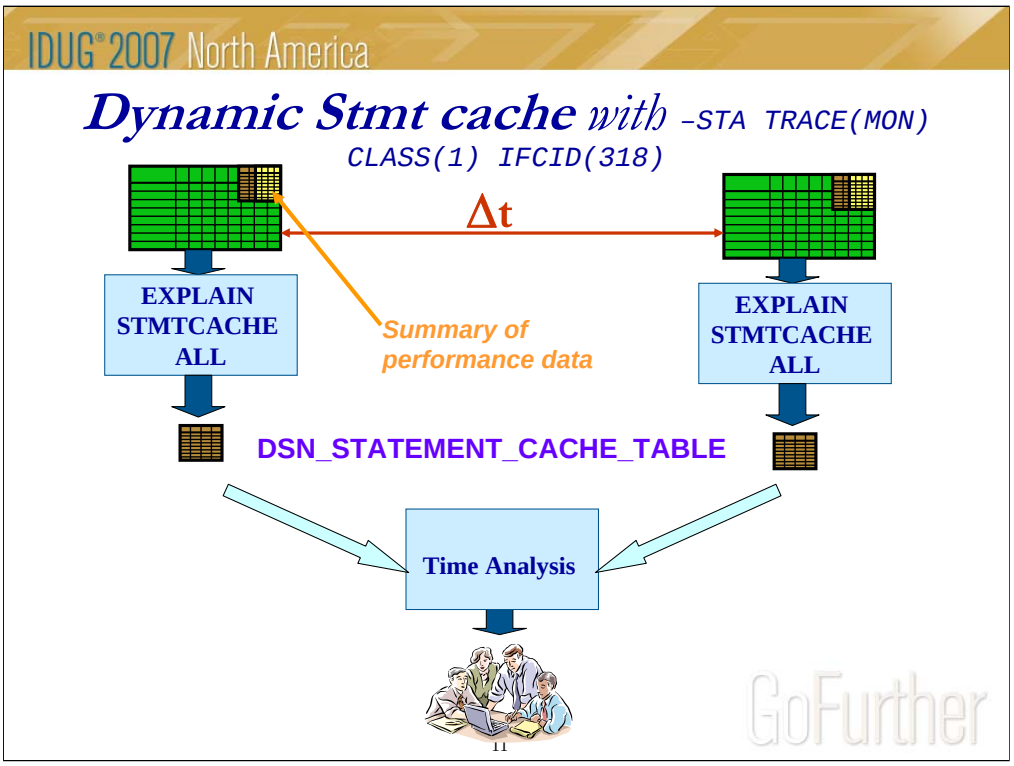


This verv first criterion identifies the most inefficient dynamic SQL statements the ones which should be attacked at first – the *queries from hell*.

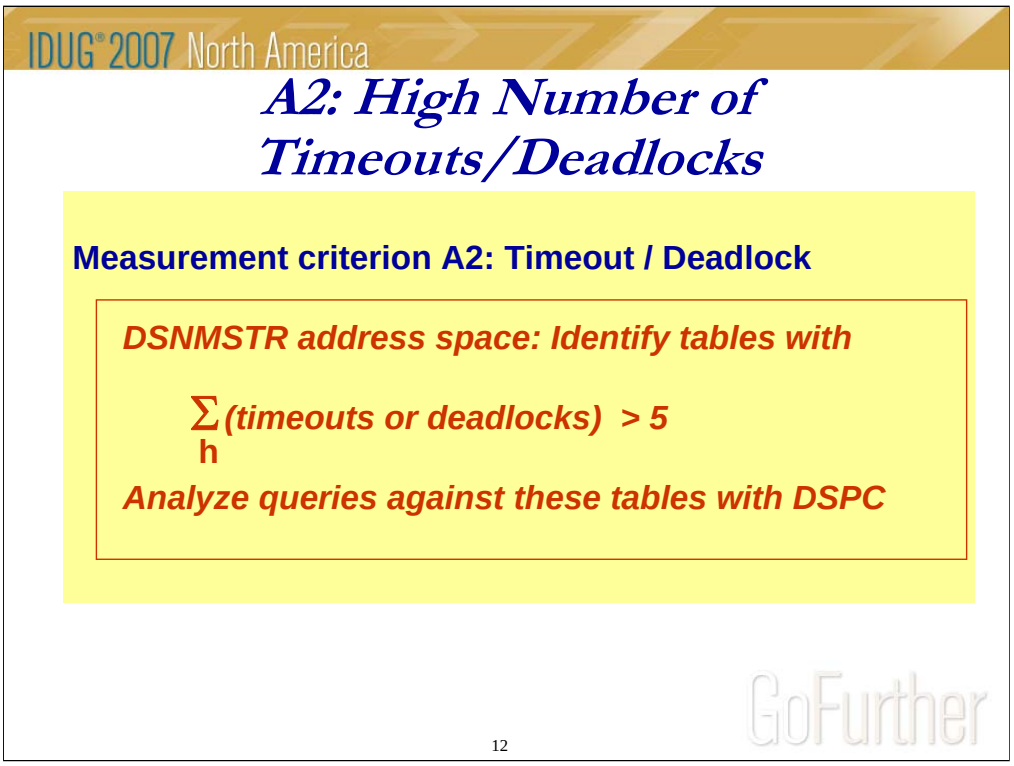


Analysis of the global dynamic statement cache has become easy in DB2 for z/OS V8: Simply define the dsn_statement_cache_table, and populate it by executing EXPLAIN STMTCACHE ALL.

If you repeat this command a certain time later you can subtract the performance numbers from the statements with the same stmt_id, thus producing a result of all queries exeuted between these two measurements. Make sure to have this time interval small enough in order to capture all statements.



"The collection of statistics for statements in the dynamic statement cache can increase the processing cost for those statements. When IFCID 0318 is inactive, DB2 tracks the statements in the dynamic statement cache, but does not accumulate the statistics as those statements are used. When you are not actively monitoring the cache, you should turn off the trace for IFCID 0318."



Another objective of emergency tuning is the detection of statements which cause timeouts and deadlocks.

A3: Missing Parameter Markers

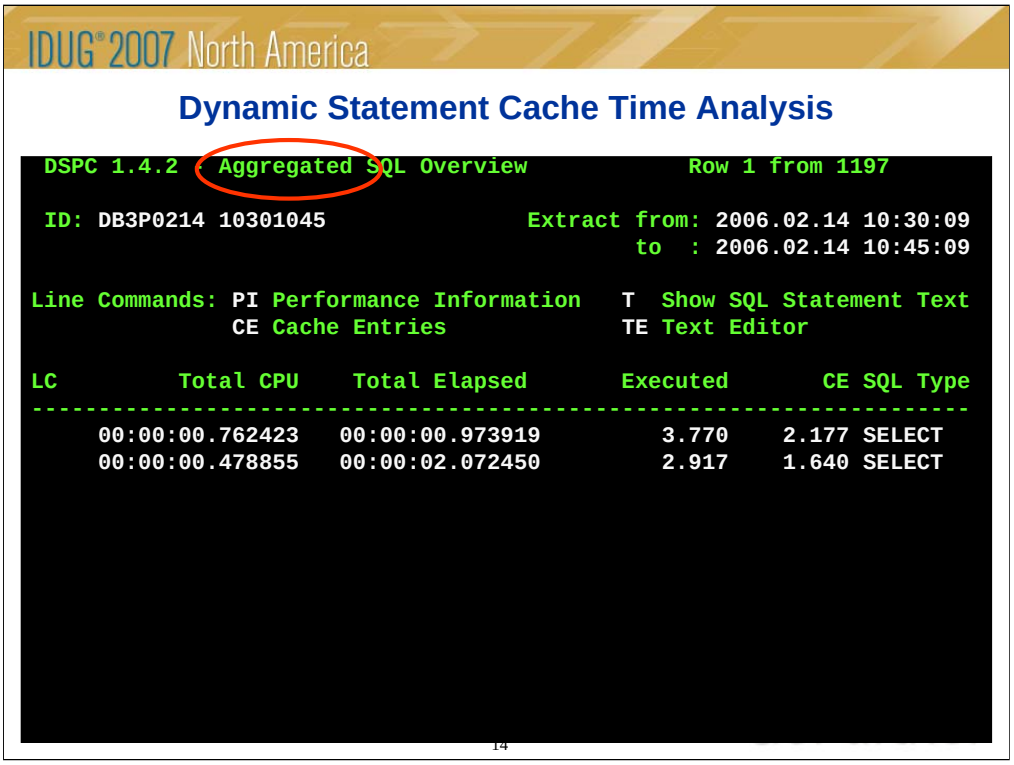
Measurement criterion A3: Efficient Use of Dynamic Statement Cache

$$\text{Query Execution / Cache Entry} < 2 \quad \& \quad \sum_{h} (\text{Query Execution}) > 10000$$

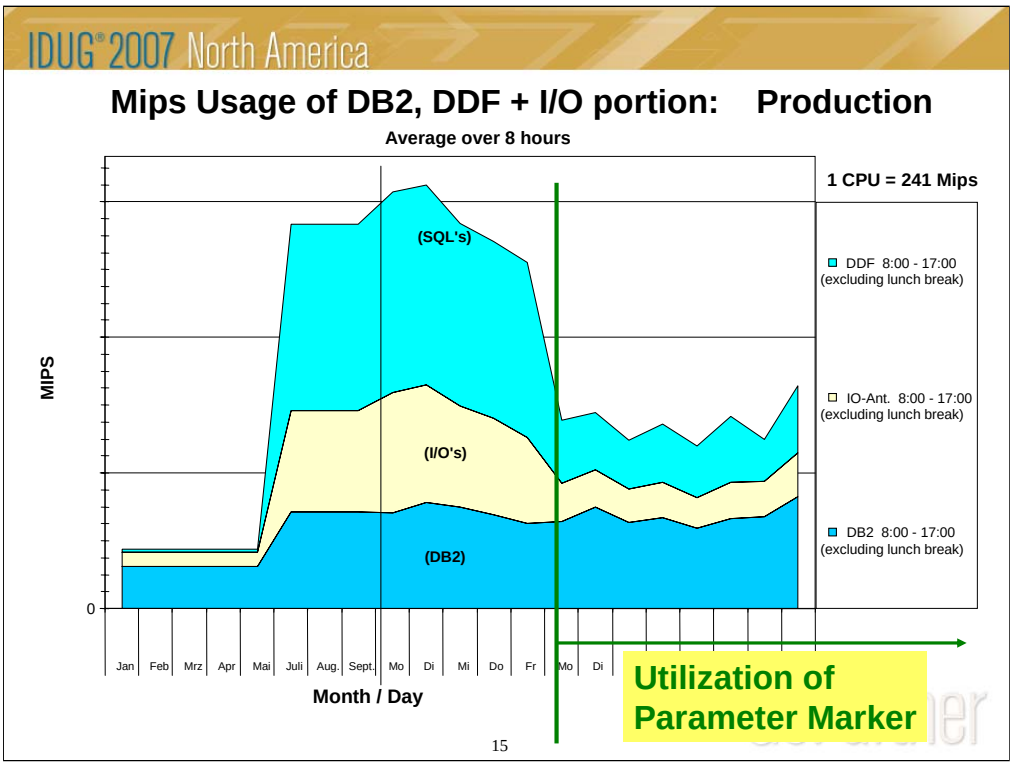
One of the largest problems in dynamic SQL tuning is the usage of parameter markers: If you don‘t use them, every single instance of a dynamic statement is considered to be a separate statement, and needs a complete bind process. And will be placed in the dynamic statement cache even if it will never be rereferenced.

On the other hand, if you use parameter markers for all kind of values, the optimizer can not benefit from the frequency values statistics.

In general, you should use parameter markers for all numbers and names which are really variable, such as client numbers, customer names, adresses, etc.



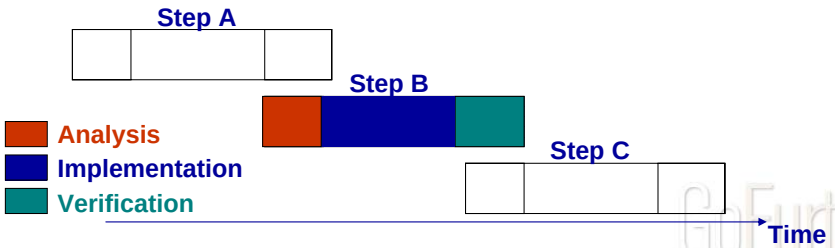
Parameter markers should be implemented to efficiently use the dynamic sql statement cache: This is the case for really variable values (like customer numbers or account numbers). *Constants* should be used for rather static values such as a status code, a flag or a currency code (columns with a low cardinality and/or a very skewed data distribution).



This slide shows the effect of introducing parameter markers in one of Swiss Mobiliar‘s key applications. The overall CPU consumption (not only the CPU consumption of dynamic SQL statements) was divided by two!

BMW for SQL Tuning: Step B

- ✓ Step A: Are there extremely critical SQL queries?
- ➢ **Step B: Is the workload being adequately supported by the database system?**
- • Step C: Are enough system resources (CPU, Memory, …) available?

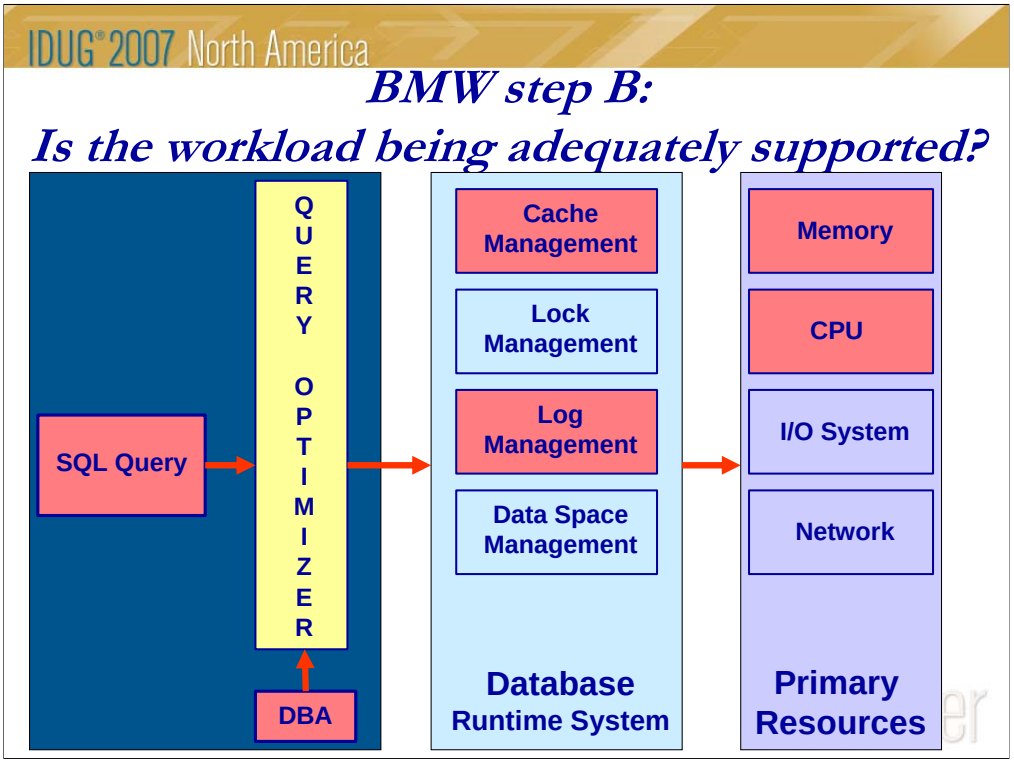


16

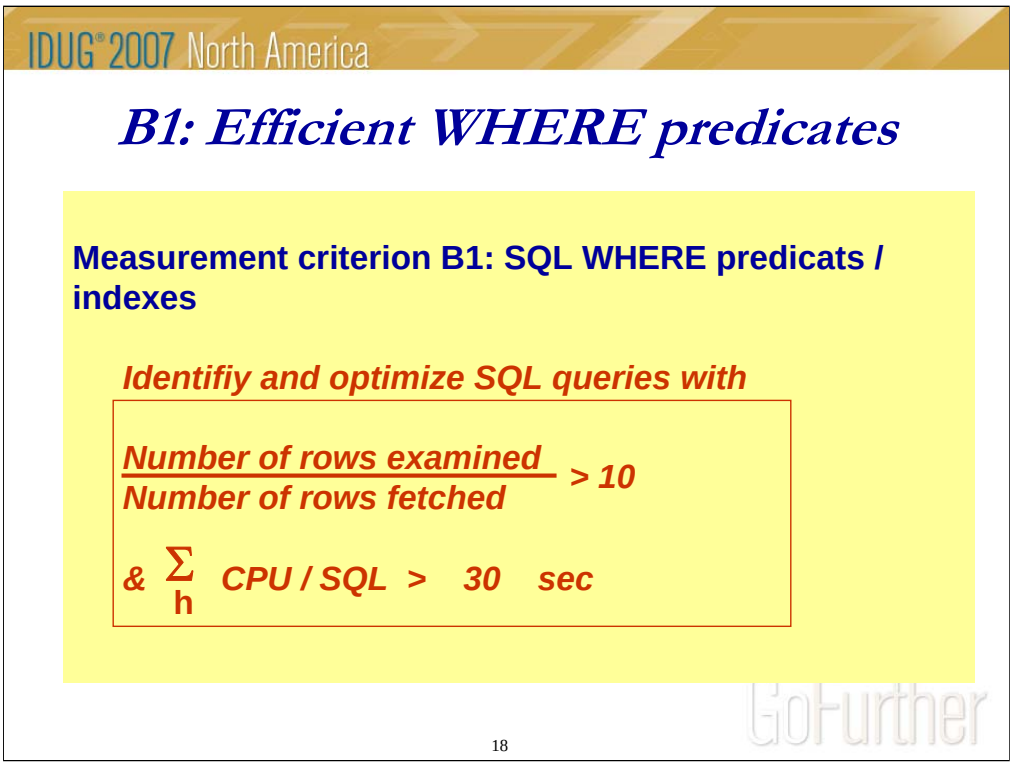
Now the first step has been done. The most critical queries are corrected. If they would still be in the system, there is a large probability that they would critically impact the results of the analysis of this second step – and leading to the wrong performance tuning actions.

If this methodology is used for routine monitoring, the first step should not identify any candidates.

Optimization opportunties detected during this second step do not require immediate (emergency) actions, but should be corrected until the next application release or system maintenance window. .



After tuning the most urgent SQL problems, there are still some areas coloured in red which require your attention. So we don‘t want to loose too much time and start to resolve these probmlems immediately.



This criterion lists SQL statements which need a certain (too large) amount of rows to be analyzed before finding a row which corresponds to the where predicates. This might have different reasons:

- no indexes available which cover the query‘s predicates
- stage 2 predicates
- old or unsufficient catalog statistics

etc.

Dynamic Statement Cache Time Analysis

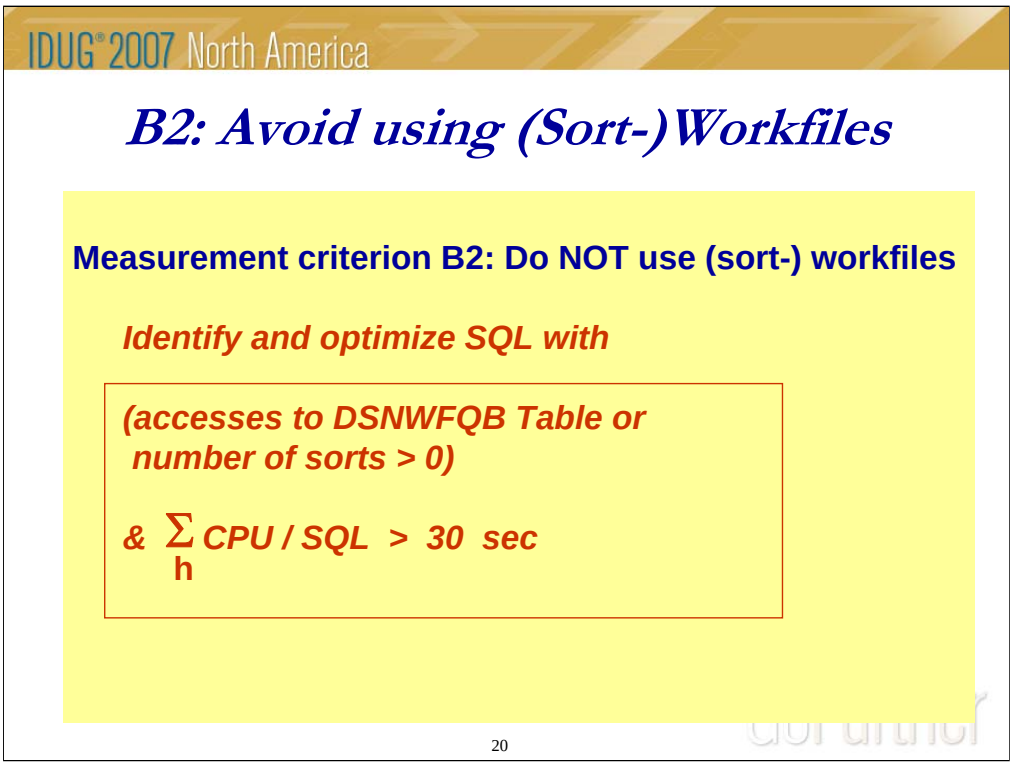
example of inefficient index usage

```
                SQL Statement Performance Information
COMMAND ===>                                              SCROLL ===> CSR
ID: DB3P0424 10301045           Extract from    : 2006.04.24 10:30:53
                                        to      : 2006.04.24 10:45:15
Aggr. ID : 1.857.158            in Cache since : 2006.04.22 09:03:18
SQL Type : SELECT               # Stmt execs   : 13
-------------------------------------------------------------------------
                                                  Total          Average
CPU Time ........................:    0:00:20.587893    0:00:01.583684
Elapsed Time .....................:    0:00:23.529180    0:00:01.809936
# Synchronous Buffer reads ......:              16
# Getpage Operations .............:         727.677           55.975
# Rows examined for Statement ...:       1.211.327           93.179
# Rows returned for Statement  ..:              39                3
# Sorts performed for Statement .:               0                0
# IDX Scans performed for Statement          1.709              131
# TS Scans performed for Statement               0                0

Wait Time Synchronous I/O .......:    0:00:00.356062    0:00:00.027389
```

19

These are typical values reported for queries with inefficient search attributes.



The DSNWFQB table is a table which you or your DBA colleagues probably never have defined. It represents the materialization of a query block during query execution (a *workfile* temporarily created by DB2), and has a great potential of causing performance problems.

B3: Less Overhead for Locking

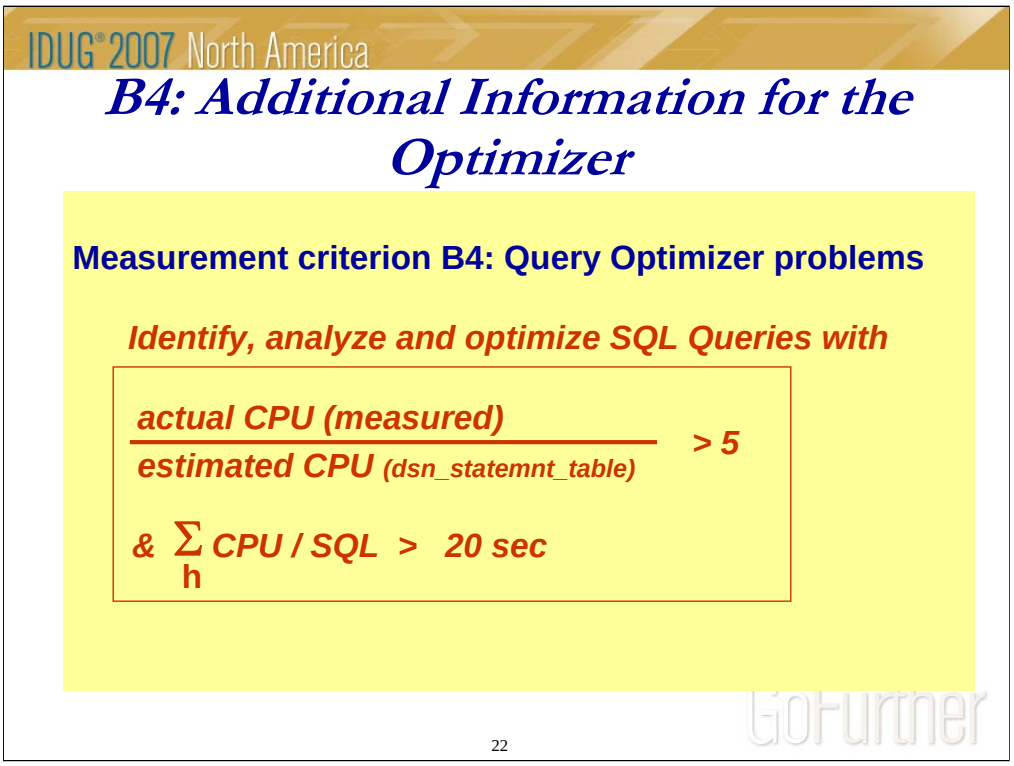
Measurement criterion B3: High degree of locking

Identify and optimize SQL Queries with

(Wait Time Lock / SQL > 0.01 sec or
Wait Time Lock / SQL > 5% of Elapsed Time or
Wait Time Global Locks > 5% of Elapsed Time)

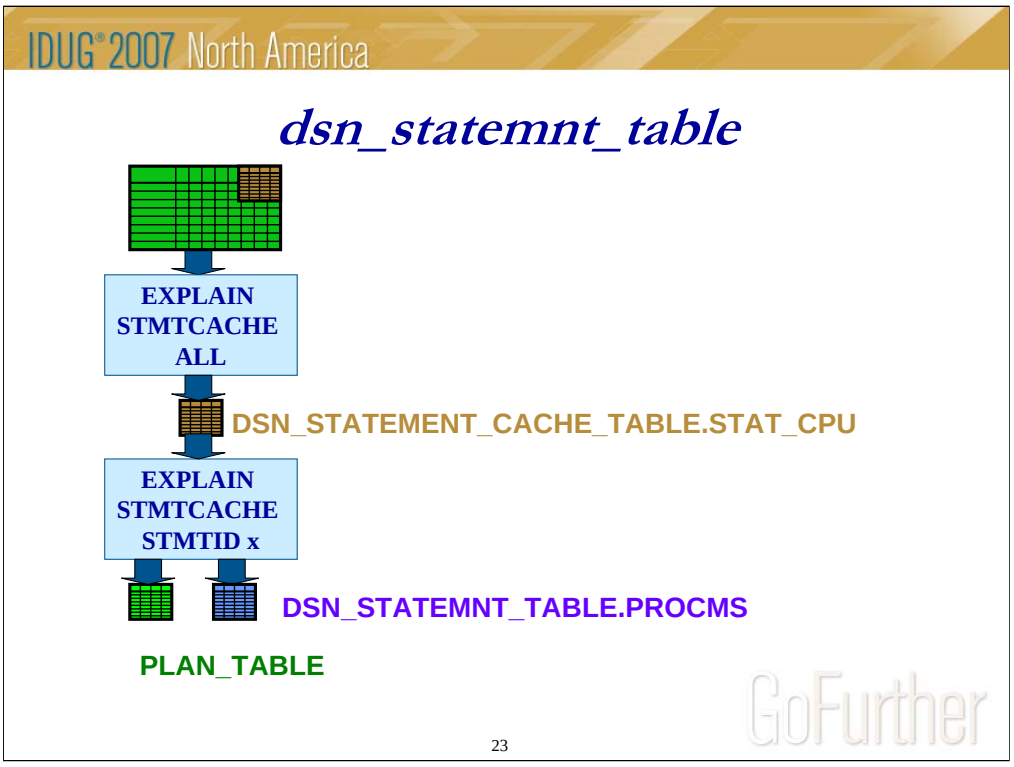
& $\sum_{h}$ Elapsed Time / SQL > 30 sec

Instead of looking at queries which cause deadlocks and timeouts, the queries reported here show a high amount of time spent in waiting for locks, but without necessarily reaching the timeout limits.

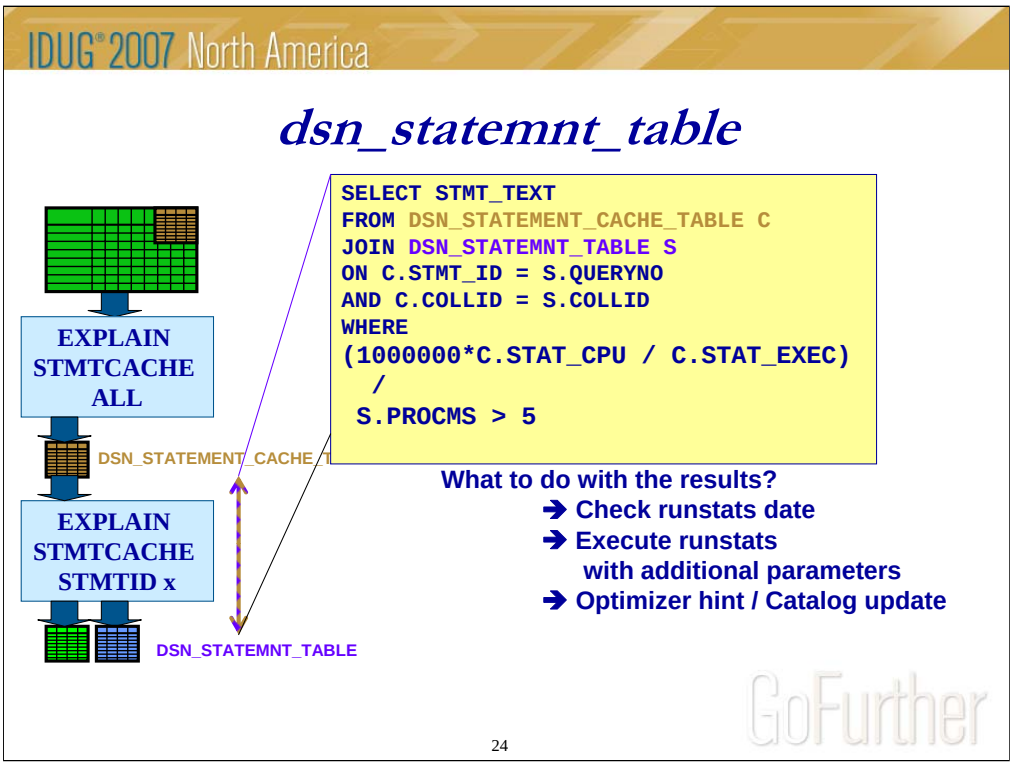


If the optimizer chooses a wrong access path, the final cpu measurements are very often far away from what it initially estimated.

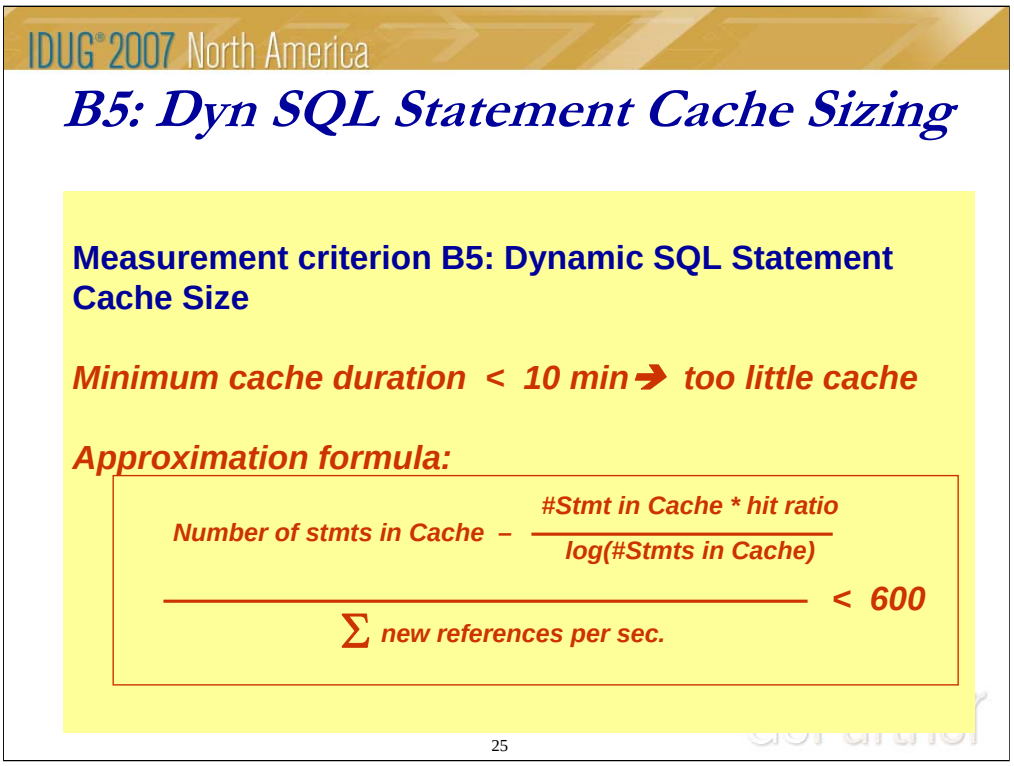
The ratio of measured compared to estimated CPU is a very good starting point to identify the queries which have inefficient access paths, often due to missing or wrong statistical information at the time the query was initially boud or prepared.



This is how to get the information to calculate the *measured vs. estimated cpu ratio*. The EXPLAIN STMTCACHE STMTID x statement does not recalculate the access path, but does insert the originally calculated access path information into the plan_table, and the original cpu estimate into the dsn_statement_table‘s PROCMS column.



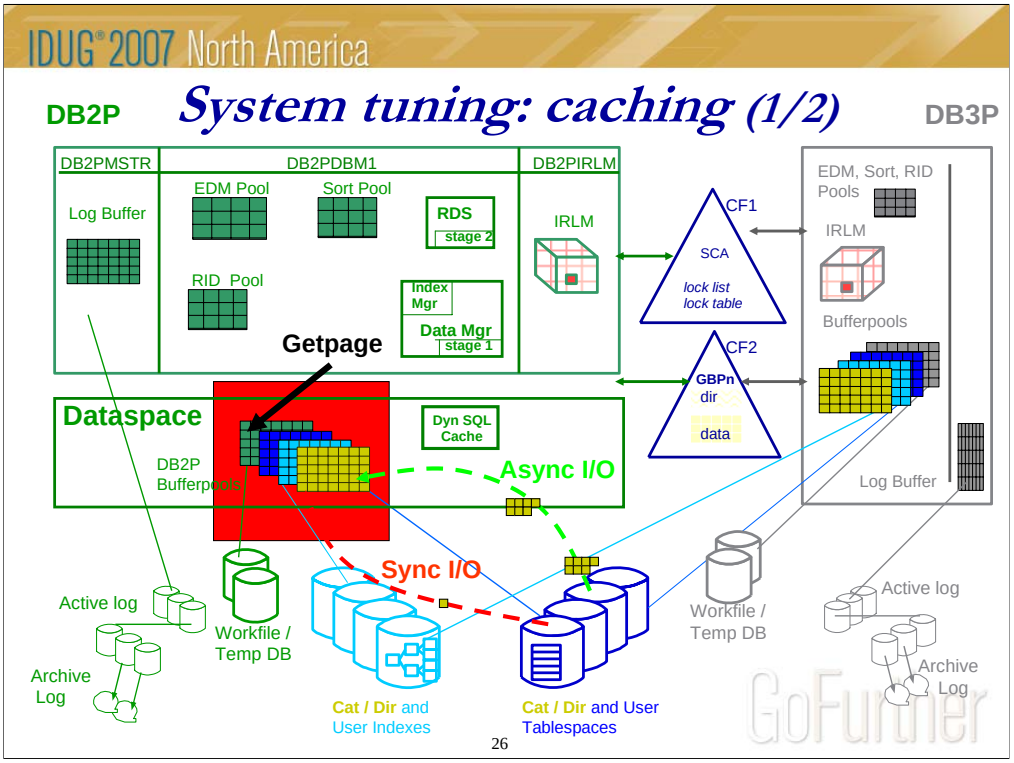
The intensive usage of dynamic SQL changes RUNSTATS: Now, this utility has a certain potential to increase a system's stability: The new statistics are immediately active and might change access paths (in most cases, they become more efficient). In rare cases, they might become much more inefficient. Therefore, the usage of RUNSTATS should be more careful.



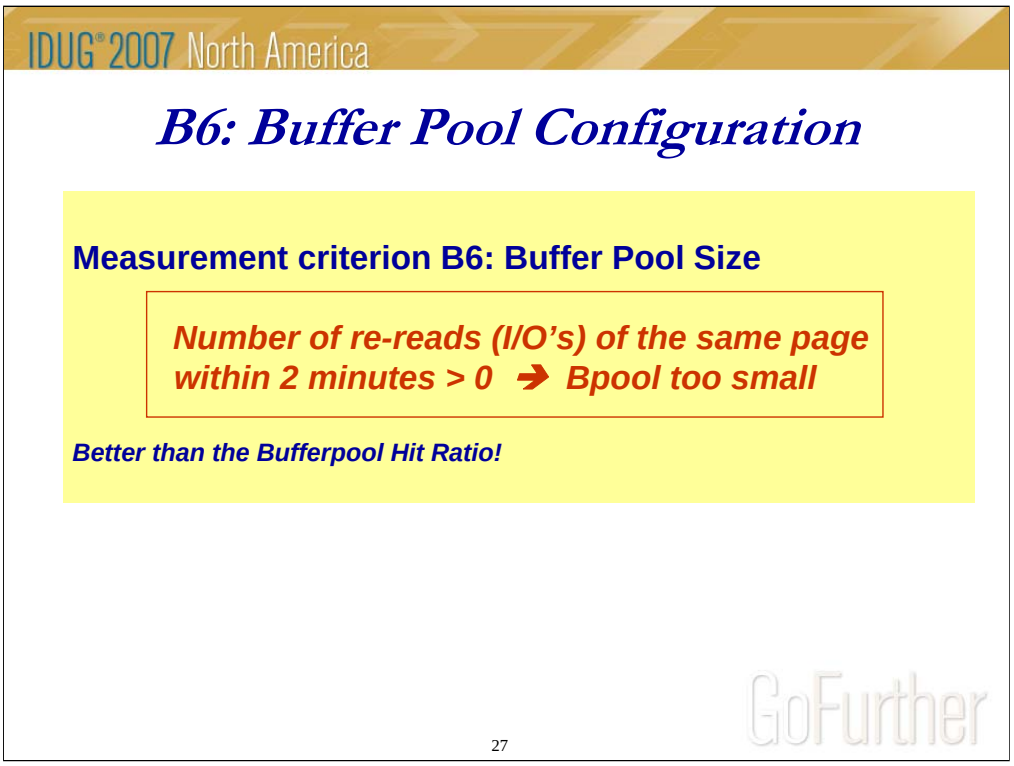
You might wonder how we found an upper limit of 10 minutes to be a good value to determine the dynamic statement cache‘s size. In fact we also do.

What we tested and measured was the impact of a larger dynamic statement cache (with a minimum cache residency time of up to 60 minutes). The benefit for the overall system was not much better than with a cache size according to the 10 minutes limit. However, with a smaller dynamic statement cache, the overall numbers and specifically the values measured for prepare-statements, were higher.

So we decided to have 10 minutes as our performance goal.



Once again, the ‚internal' view of where we are.



Bufferpool hit ratio is a very bad measurement criterion for bufferpool tuning: A single inefficient query (which retrieves for example the same few of a subquery a billion times) can lead to a bufferpool hit ratio of close to 100% - even with inefficiently organized bufferpools.
High bufferpool hit ratios can be a sign of good bufferpool tuning, but also a sign of inefficient queries being executed.
Another criterion (additionally or instead of 'No of. ReRead within 2 min > 0' is the minimal page residency time. If it is less than 2 min for a bufferpool the pool might be too small. The number of re-reads is more efficient (only really re-read pages are counted), whereas the minimal page residency time only identifies a potential for re-reads.

However, computing the number of re-reads involves executing an SQL performance trace (IFCID 101), whereas the minimal page residency time can be calculated using the same formula which was used for the dynamic statement cache. For FIFO type buffer pools, this formula is even simpler:

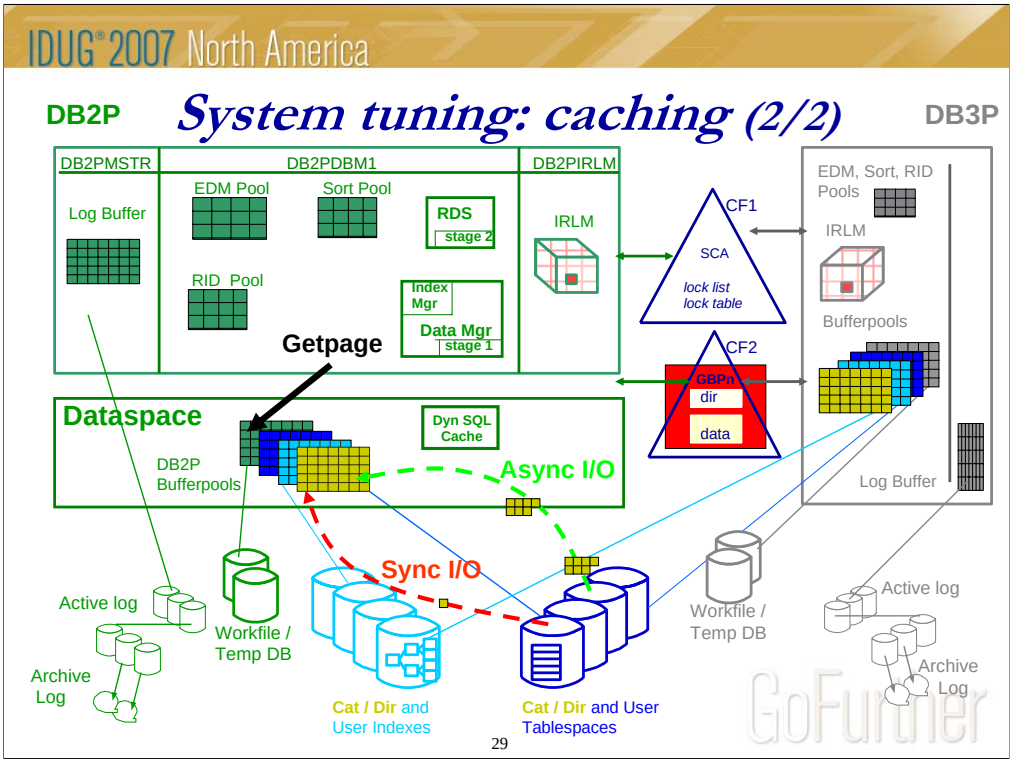
$$\frac{\text{Number of pages in Bpool} * \frac{(100-\text{vpseqt})}{100}}{\text{Sync I/O per second}} < 120$$

Buffer Pool Configuration DB2P

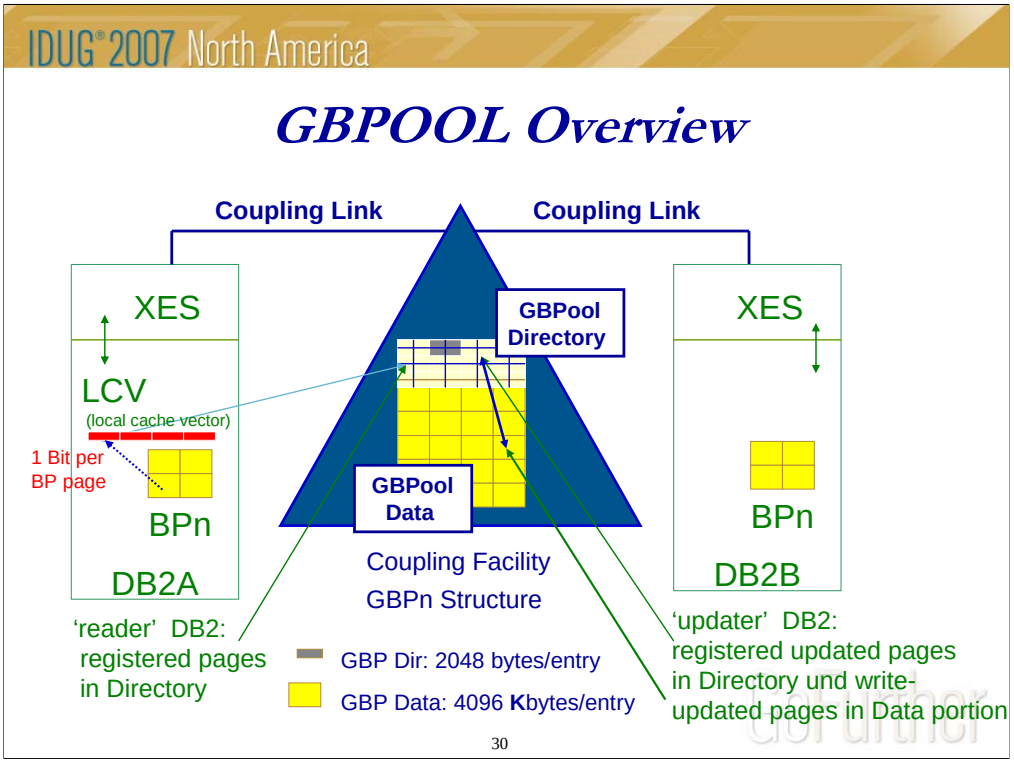
BP	Data type	Size	VPSEQT	DWT	VDWT	Method
0	Catalog	10K	80%	50%	10%	LRU
1	Data	1.0M	20%	0%	0%	LRU
2	Index	1.3M	20%	50%	10%	LRU
3	In-memory Heavy-read	40K	0%	50%	10%	FIFO
4	In-memory Heavy-upd	620K	0%	90%	90%	FIFO
5	Pipeline	5K	80%	80%	50%	FIFO
6	Workfile	20K	100%	70%	50%	LRU

Probably, you have your own bufferpool strategy. It it not my goal to sell my bufferpool tuning strategy. This is just for information how another company has defined their bufferpools.

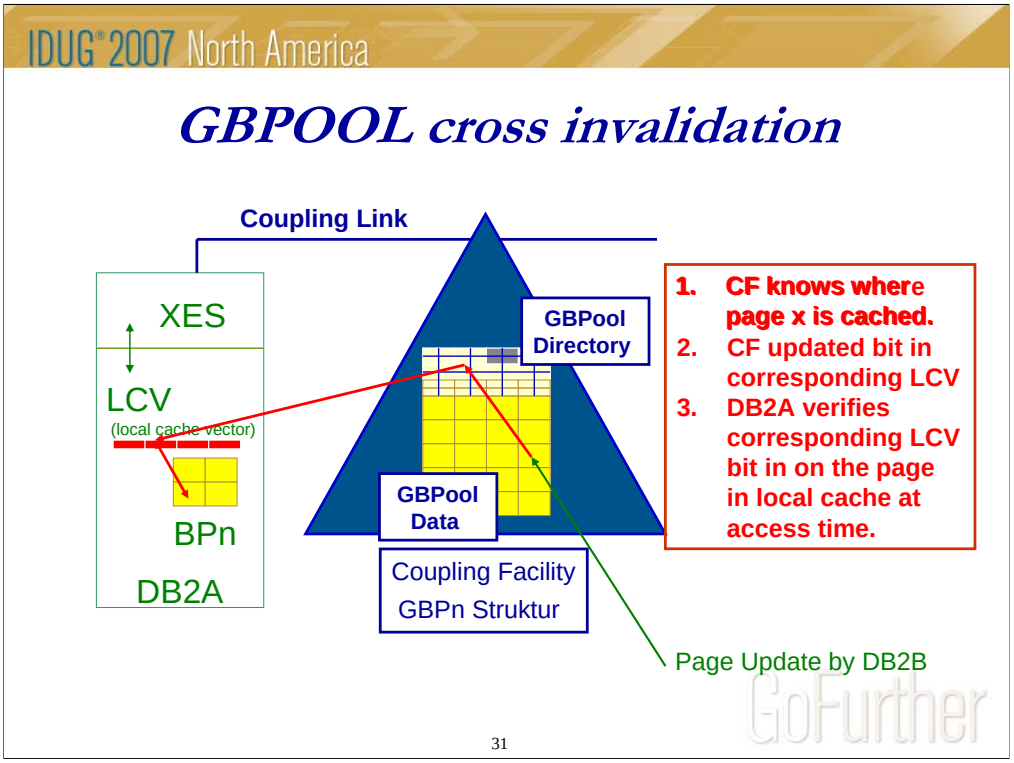
What is important is that you have a certain strategy, which is both easy to implement and easy to understand.



Now we look at the group buffer pool‘s point of view.

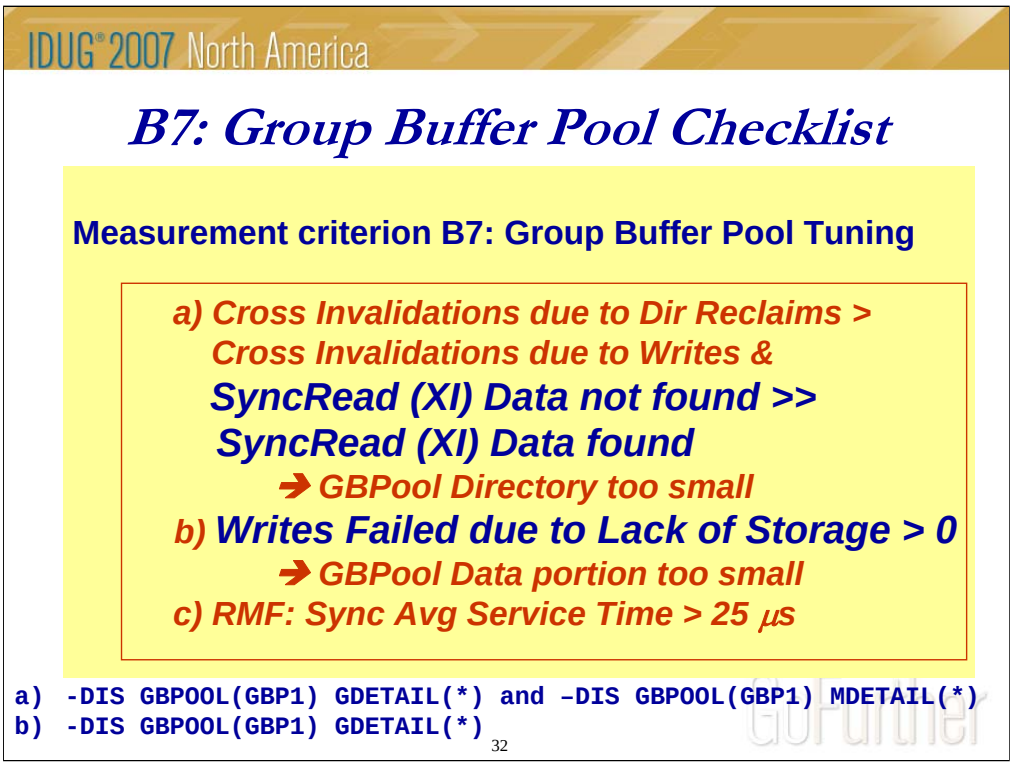


Group Bufferpools are used to make sure that all data sharing members get the most current state of a page. A positive side effect of this feature is the ability to avoid I/Os, which might already have been done by another data sharing member for the same page.



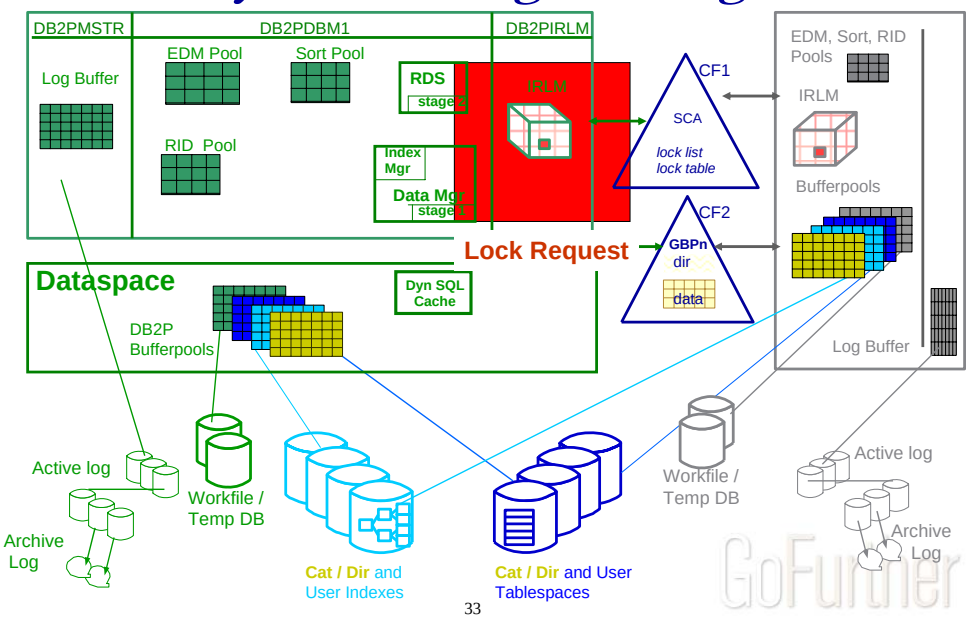
There are two events which lead to ‚cross invalidation‘:

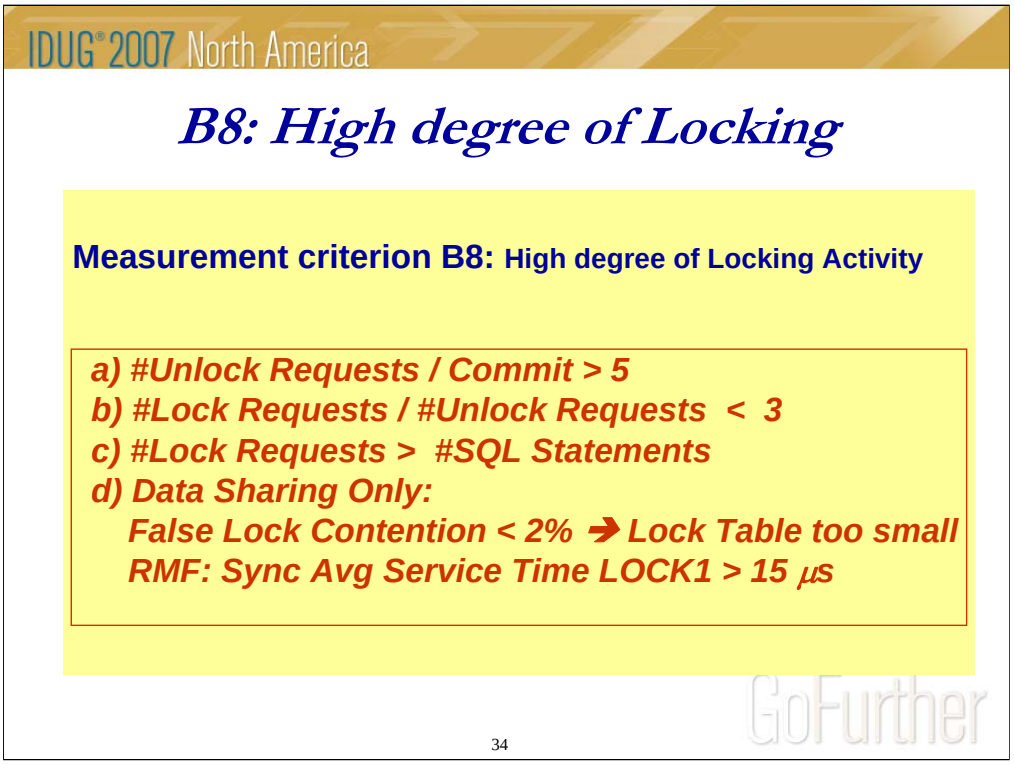
1. A page update in data sharing member DB2B cross invalidates the local buffer pool entry for the same page in DB2A.
2. If, during the Page Registration Process, no free slot in the directory is found, the oldest (LRU) directory entry will be deleted and the corresponding page in the local bufferpool will be invalidated (via the same mechanism used during the page-update process). If this page is later re-referenced in the local bufferpool, it needs an additional disk I/O.



These are the key values which we are happy to meet. Specifically for dynamic SQL workload for DB2 for z/OS V8, look at these values for GBP8K0 and GBP16K0, since these buffer pools contain now catalog data necessary for bind (and prepare) actions. If you see writes failed due to lack of storage in these pools, make them higher (10K should be ok), and recalculate elapsed times for bind and prepare operations of complex queries.

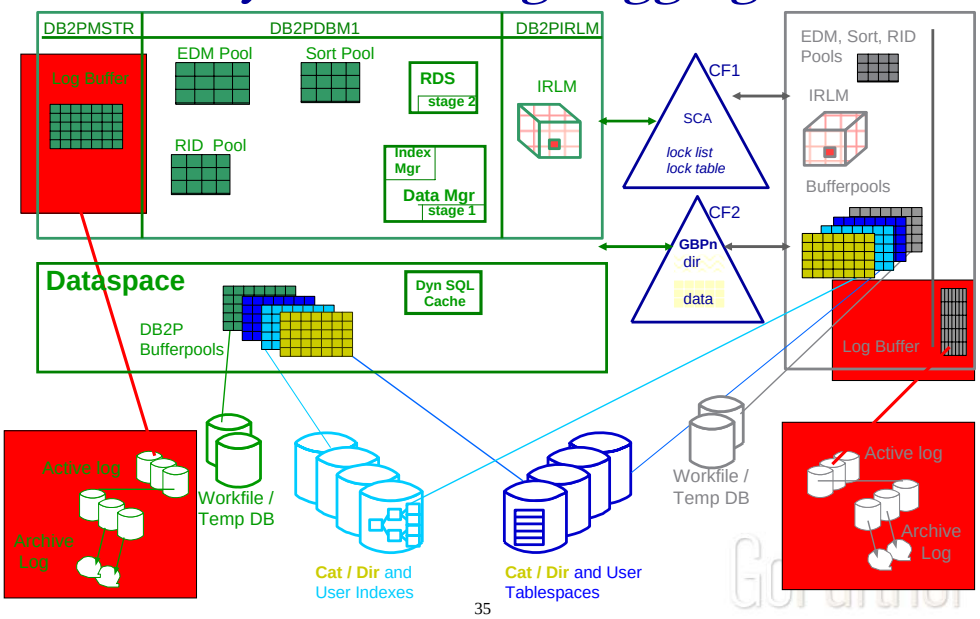
System tuning: locking





Unfortunatley, there is no knob to reduce locking or to resolve locking problems in general. Most important is the isolation level, and best practise is to have uncommitted read as your standard isolation level for remote applications – with exeptions at the statement level.

System tuning: logging



B9: Unnecessary Log Waits

Measurement criterion B9: Log Tuning

Wait for Active Log Write I/O > 1% of Elapsed Time *)

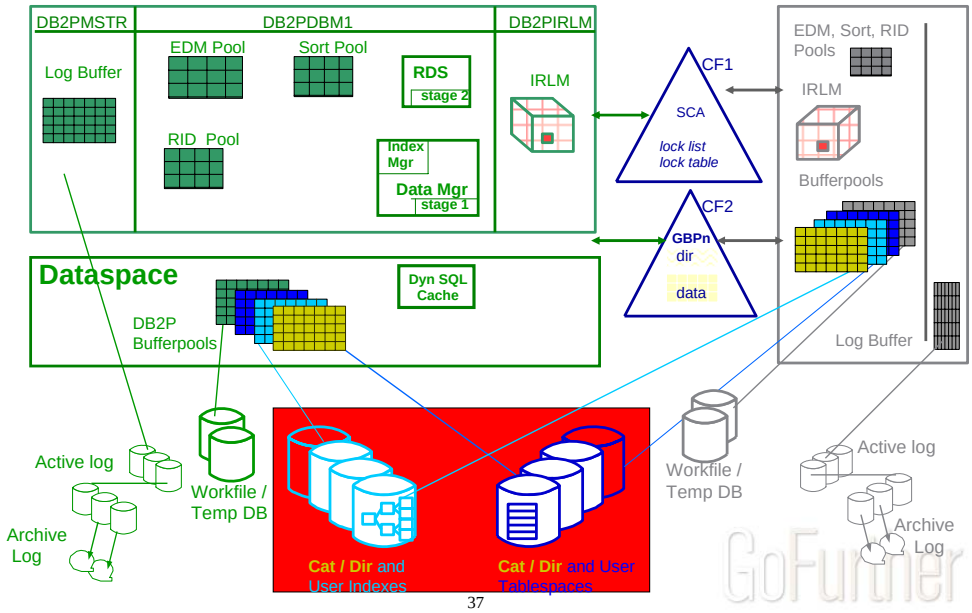
➔ enlarge the Output Log Buffer

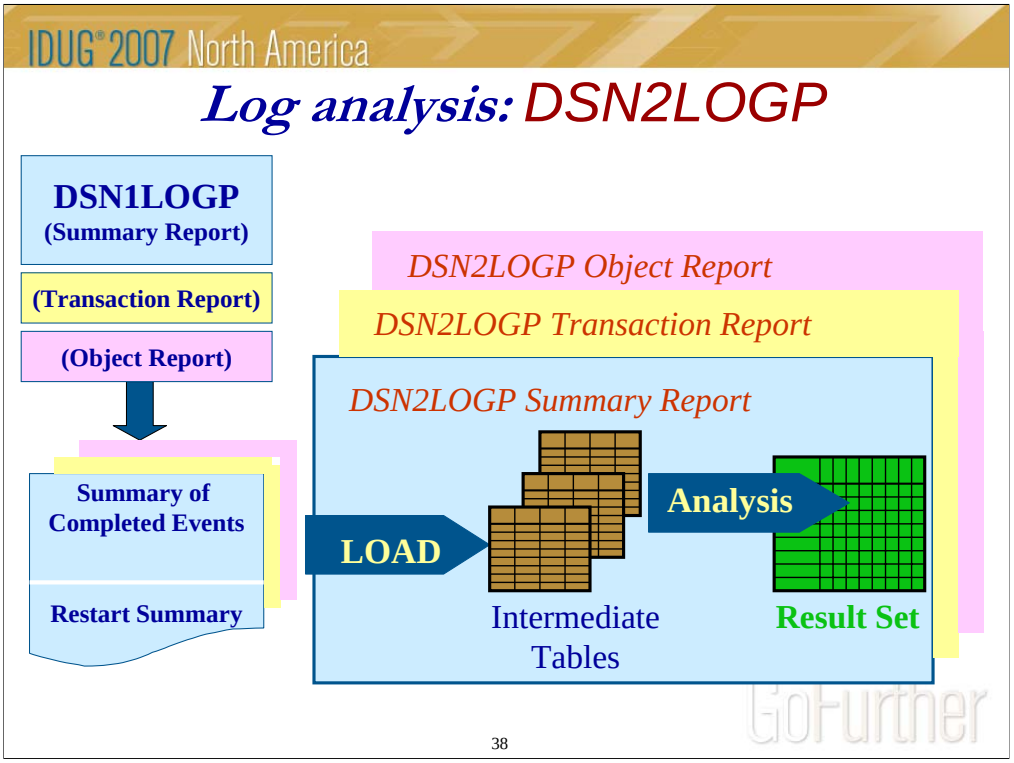
****) applies only if Log Speed <= 2 Mbyte / sec***

The log output buffer is situated in the DSNxMSTR address space, and not in the DSNxDBM1 address space. The good news is that there are no memory constraints, the bad news is that the additional memory which you require when increasing OUTBUFF zparm (which controls the output log buffer size) does not come for free, and is in competition with the memory demands for buffer pools, dynamic statement cache, etc.

In a nutshell: Make the output log buffer as large as necessary, but keep it as small as possible!

System tuning: Data Space Management





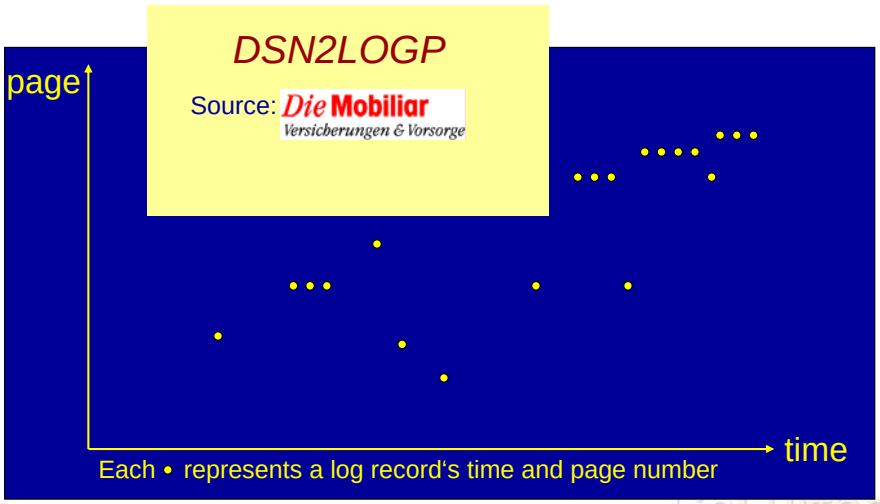
As DSN1LOGP‘s output is hard to interpret (especially if you want to do that using a systematic approach), we load the DSN1LOGP output into intermediate tables from which we calculate performance results, such as the page numbers accessed at given time points.

This straight forward approach to a graphical interface of a pageset‘s activity is sometimes of great help.

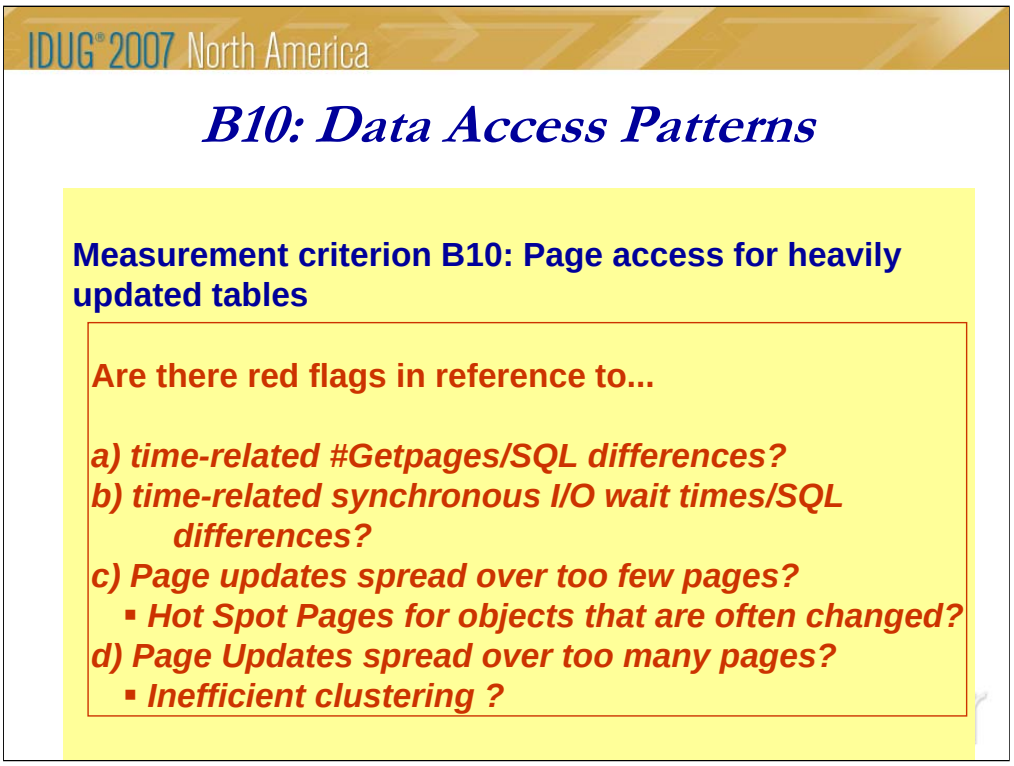
DSN2LOGP Object Report: Time View

Time	Page	Transaction URID	Plan
2006-01-11-15.18.28.236272	00007F33	142071F38D34	P72S30
2006-01-11-15.18.32.291728	0000864C	142072298BF9	P72S30
2006-01-11-15.18.47.692144	00000003	0BB37A14E000	P14501
2006-01-11-15.18.47.793520	00007F33	142072C7F689	P72S30

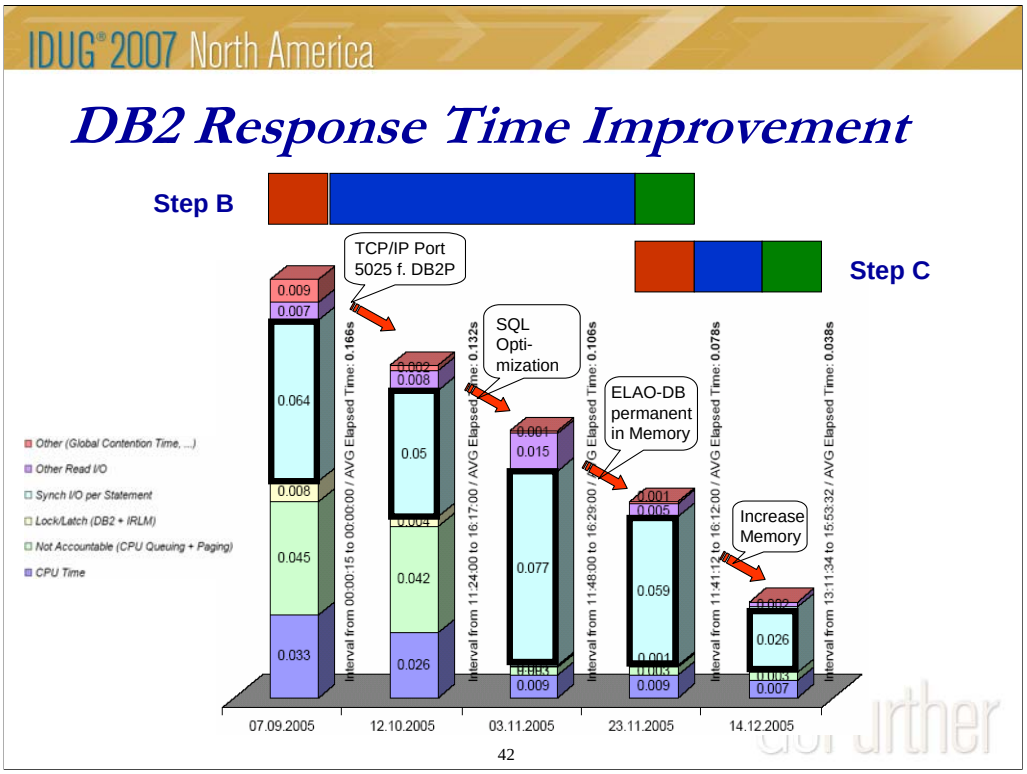
DSN2LOGP Object Report: Time View



There is no simple measurement criterion for cases such as this. Based on pattern detection, adequate optimization techniques might be applied, always bearing in mind the tradeoff between optimal locking behaviour and minimal I/Os.



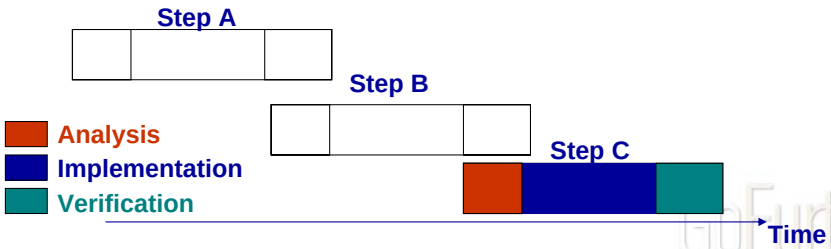
Some questions arising from DSN2LOGP object analysis.



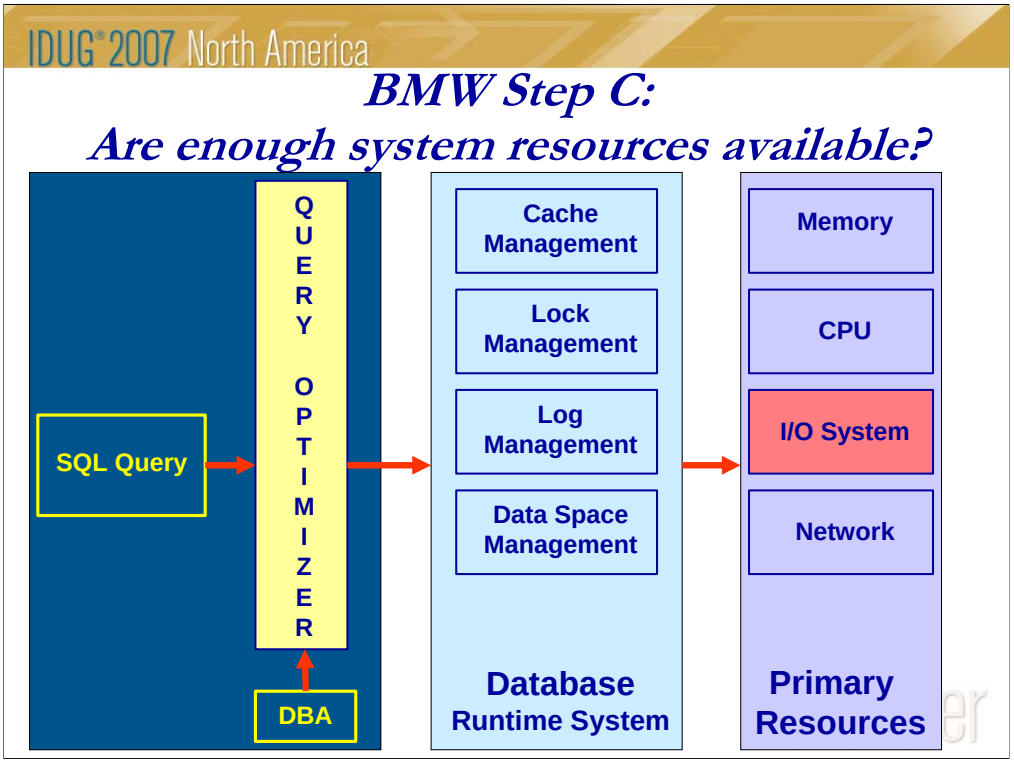
These measurements were produced by IBM during a review of our performance activities after a major application‘s rollout. The numbers identify average processing times of single SQL statements of this application.

BMW for SQL tuning: Step C

- ✓ Step A: Are there extremely critical SQL queries?
- ✓ Step B: Is the workload being adequately supported?
- ➢ **Step C: Are enough system resources (CPU, memory…) available?**



Now the second step of the methodology has been finished. Again – all the problems identified must be resolved before the third step can start..



Now it is time to concentrate on primary resources.

C1-C4: Check System Resources

Identify bottlenecks:

C1) Memory Check:

> ***Bufferpool re-read within 2min > 0, Paging ?***

C2) I/O Check:

> ***Average Sync I/O Suspension > 8msec***

C3) Network Check:

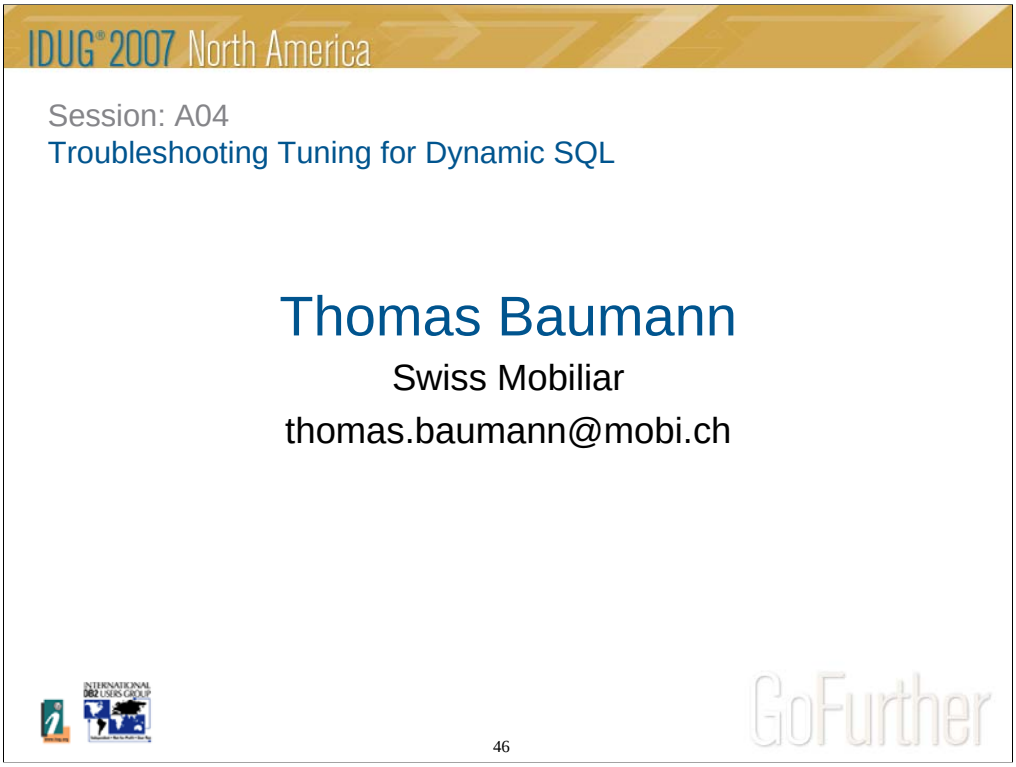
> ***End-to-end response time/SQL processing time > 2***

C4) CPU Check:

> ***'Wait for CPU' / SQL processing time > 0.5***

45

Some simple resource checks.



Thomas Baumann has been a database performance and availability professional for many years. At Swiss Mobiliar, he is, as a data architect, responsible for DB2 and is engaged in the analysis and integration of advances in the database technology with existing and new Swiss Mobiliar applications. He holds a master degree from Swiss Federal Institute of Technology (ETH) Zürich in computer science and probability theory. Thomas has also lectured on data management at the Zurich University for Applied Sciences and has conducted various workshops and delivered presentations worldwide. He is the author of the *Advanced Recovery Functions* seminar series and a member of the IDUG hall of fame, a small but distinguished group of DB2 professionals. You can reach him at thomas.baumann@mobi.ch.