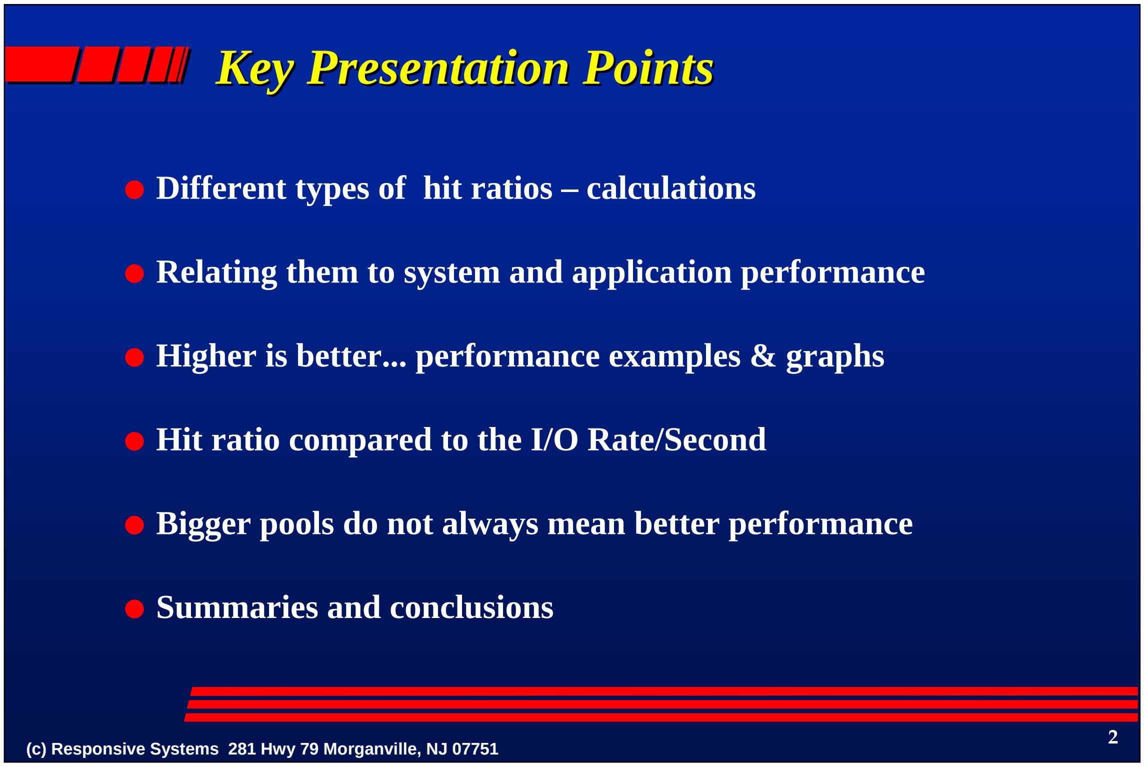
The Buffer Pool Hit Ratio is Dead !!

Joel Goldstein
Responsive Systems

(732) 972-1261
Email: joel@responsivesystems.com

Web Site
www.responsivesystems.com



History is important – you have to know where you came from, and where you are, to be able to move forward. We need to understand what hit ratios mean relative to performance – and then compare them to the I/O rate to see the difference. This presentation will illustrate that an I/O rate is meaningful and measurable, and that a hit ratio is not necessarily relevant to performance.

With 64bit memory, and Gigabytes of memory, is it still worth tuning, or should you just throw memory at the pools? Data shows that more memory often does not improve performance unless you can make the pool(s) larger than the data – and this still can't happen in most systems.

Background

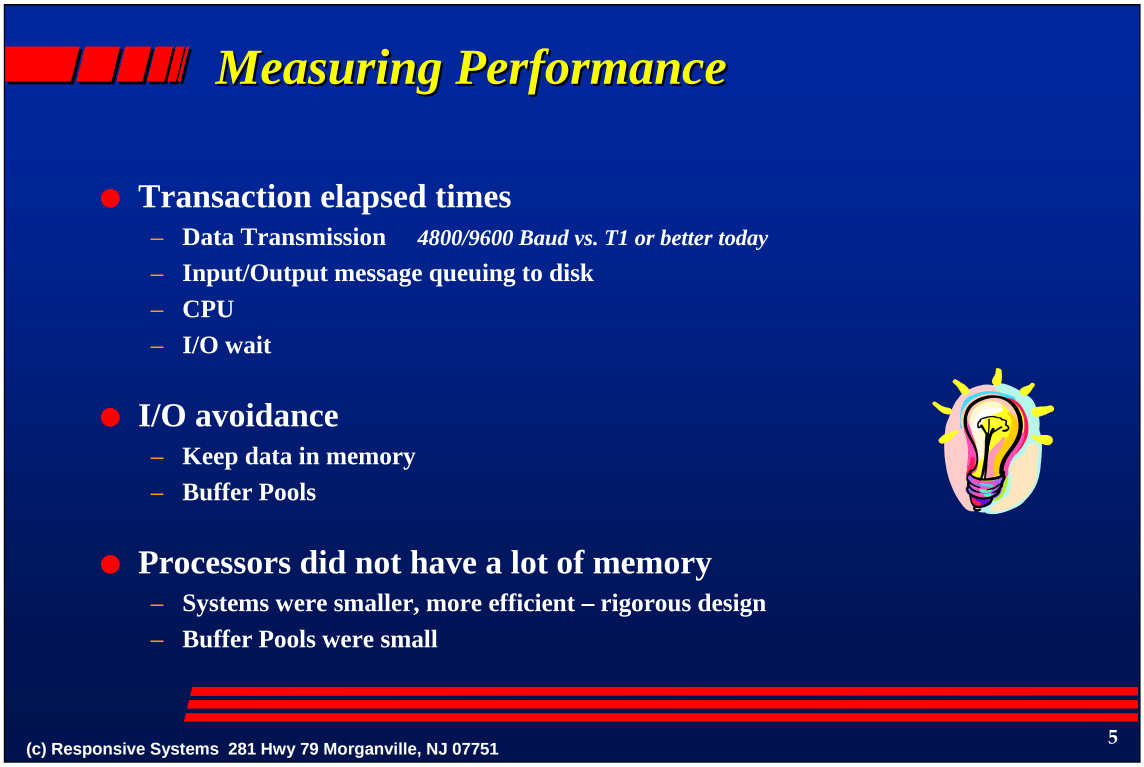
- Database Management Systems
- Physical I/O – or keep Data in memory
  - Memory is fast, I/O is slow
- Both were expensive
  - While memory is much less expensive every year, and CACHE memory is even less than mainframe memory, DASD is really cheap now
- Online systems
  - Performance became important
  - Users were waiting for a response

I/O has always been a performance problem and limiting factor for large applications. This has not changed today, and the goal of DB2 is to eliminate/reduce I/Os by keeping data in memory. As we look ahead to DB2 V8, it will allow up to a TB of memory specification for the buffer pools. Of course, this depends upon your processor having this much or more available.

Access to data

- All access was random, in theory
  - Of course you could still read the whole database
- Processing was Record oriented, not Set oriented
  - Hierarchical
  - Network
- There was no anticipatory reading of data (prefetch)

When all access was (is) random, a hit ratio had some relevance to performance – but was (and is) not measurable.



Many of the major factors of transaction response time have changed over the years. In the early 70's, it was common to see data transmission times for (basically) text messages of 1600 bytes take 3-6 Secs. I/O times took more than an order of magnitude longer than the averages on well running dasd subsystems today. Yet, there are many poorly performing dasd subsystems in the world today.

On the processor CPU side, the average desktop PC is approaches the mainframe power of a decade past.

The same perspectives exist for memory. Most desktop machines today start at 256meg.

Current mainframes can have 64 Gig of memory, and this number will increase rapidly now that we have 64bit operating systems.

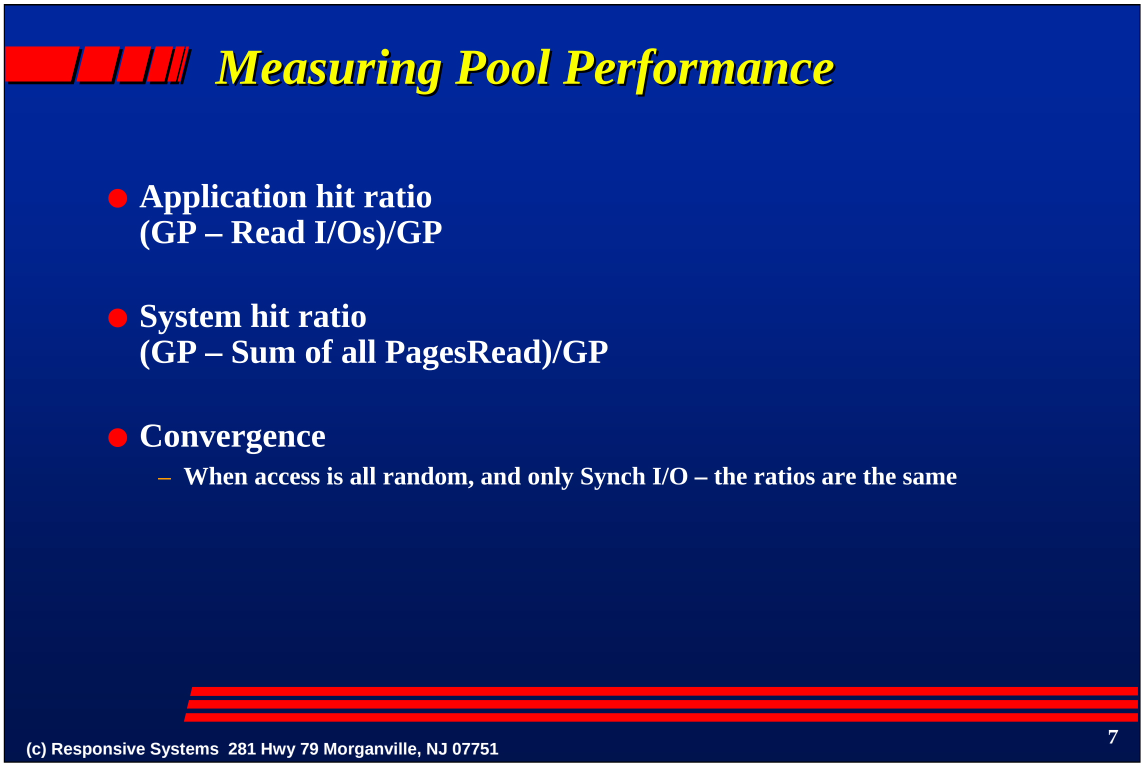
We have always filled (and often exceeded) available memory – and this won't change within this decade.

Measuring DB2 Pool Performance

- **The first measurement at the system level**
  - Hit ratio
- **Application hit ratio**
  - Ignores any pages/blocks prefetched in advance
  - Is only useful to measure application delay
  - Useless as a system tuning metric
- **System hit ratio**
  - Measures pool efficiency by factoring in pages/blocks prefetched in advance

When all access was random, and there was no concept of prefetching multiple blocks of data, determined by the DBMS software.

So measurements were simpler. Of course, if we knew that data would be accessed sequentially, we could buffer some data in memory outside a program using a bufno=x parameter on the JCL – but this still did not help data accessed by a DBMS.



The initial purpose of the first Hit Ratio we used for DB2 effectively measured only application delay, and did not really address the activity taking place in the buffer pools. Then we came to the System Hit Ratio that factored in the effect of pages read into the pool by prefetch reads.

When access is all random, so there is no sequential prefetch, and dynamic prefetch is not being used by the buffer manager, then the application and system hit ratios will be identical.

Measuring Pool Performance

- **Divergence**
  - **When access is sequential with heavy prefetch**
    - **The application hit ratio is always high, 95-98%**
      - **One I/O is 32 pages (97% hit)**
    - **The system hit ratio is always low**
      - **A complete scan, hit ratio is Zero**
  - **Dynamic prefetch reduces the system hit ratio, *and it may be negative***

Let's take the case of a pool where all objects are accessed using sequential prefetch. The application hit ratio will always be high, at least 97%, because 32 pages are read with a single I/O. A prefetch "request" can read anywhere from 0 to 32 pages, because the buffer manager knows which (if any pages) are already in the pool.

So – if all 32 pages are read, the application hit ratio is 97%; however, the system hit ratio is ZERO.

Additionally, it is quite common that many pages read by dynamic prefetch are never actually accessed by the application, so

No getpage requests are ever issued for them. This drives the system hit ratio down, and it can actually have a negative number.

We will see from data later in this presentation that pool performance is not linear. Doubling the pool size does not double the hit ratio, or cut the I/O rate in half. We will always reach a point of diminishing returns, when adding buffers to a pool.

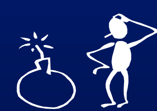
Now, I realize that some of the above items seem contradictory, and will explain them in more detail during the presentation.

The data shown in future slides will also illustrate all the above points.

As you Start to Tune Pool Performance

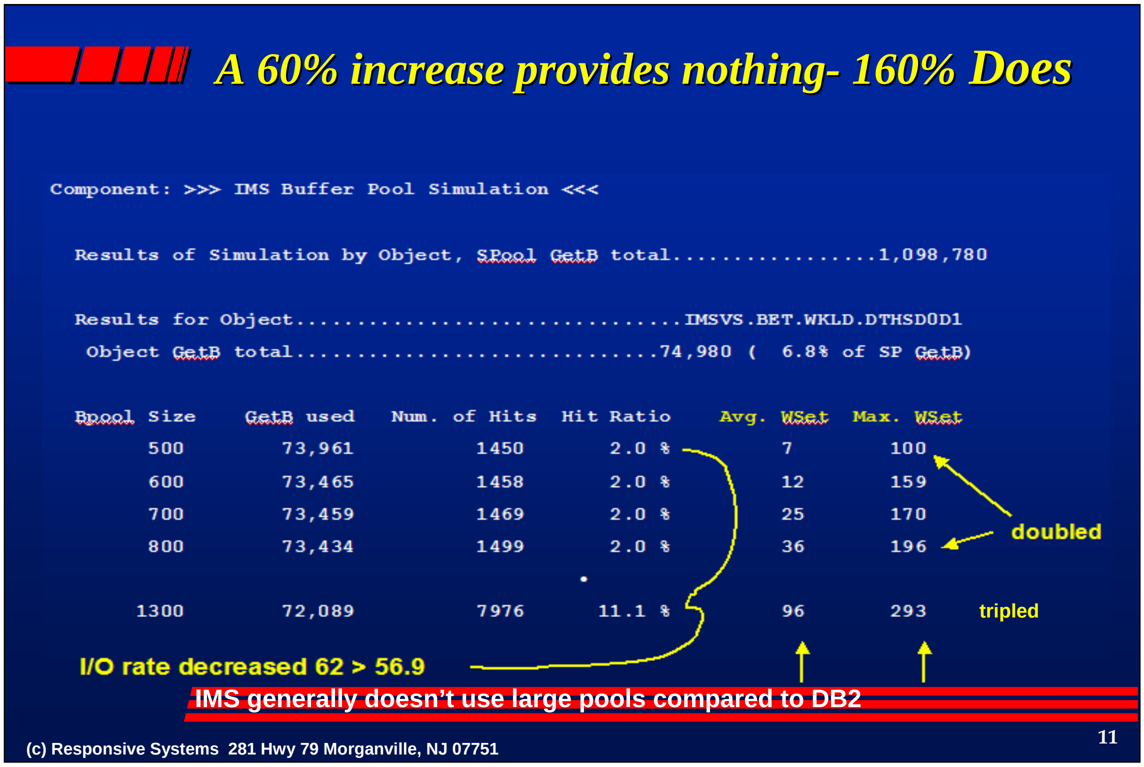
- You will find many interesting application performance problems that your online monitors may not be highlighting
  - Objects that are used in a very different manner than the application designers expected
  - Tablespaces with lots of sequential scan
  - Indexes with sequential scan

Burn those CPU engines…..



As start to analyze your pool performance, and the performance of the objects in each pool, you often find objects that are not being accessed as you expect they should be. Perhaps an Index with heavy scan, or a tablespace that is monopolizing a pool – and this can be either sequential, or a very large random object.

The key to performance, is grouping objects based on access type, and working set size.



Here I'm using an IMS example to show that even large percentage increases may not show any improvement until you pass a certain "critical" threshold for the amount of data that can remain in a pool. IMS systems do not use the amount of buffer pool sizes that we commonly use for DB2.

Another IMS example

VSAM Buffer SubPool simulation Results

Pool Name.............................TST1

Pool Type.............................VSAM

SubPool Buffer Size.................16,384

Lower Block Size.....................8,192

Number of Objects accessed..............13

Total number of Get Block...........150,169

SPool Size	GetB used	Num. of Hits	Hit Ratio	Elapsed Time
50	150,119	91	0.0 %	00:08:15
		. Missing lines .		
370	149,799	751	0.5 %	00:08:12
450	149,692	45,356	30.3 %	00:08:11
530	149,577	59,991	40.1 %	00:08:10
610	149,462	74,279	49.7 %	00:08:10

SPool Size	Blocks Read
50	303.1 /S
	. Missing lines .
370	302.9 /S
450	212.5 /S
530	182.6 /S
610	153.4 /S

No improvement until the pool is 9 times the starting size

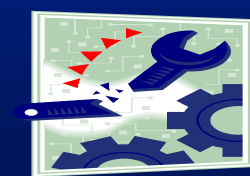
Here again, there is no real performance improvement until the number of buffers is increased by a factor of 9.

Increasing from 370 to 450 buffers reduced the I/O rate by 90/Sec. However, subsequent increases save 30 I/Os/Sec,

then 29 I/Os/Sec, so the reduction starts to taper off….

Pool Performance

- **Eliminating I/O reduces CPU costs**
  - **The CPU cost for an I/O does not change when you improve DASD performance**
- **Throughput and Productivity Improvements**
- **Eliminating I/O**
  - **Usually will not reduce the processor “busy rate”**
  - **It may increase the processor “busy rate”**
    - » ***But this is good***



The huge amounts of cache memory on today’s dasd control units are essentially an external extension of the DB2 buffer pools, provided up to 512 Gig of memory on some models.

However, the CPU cost of I/O remains the same whether the data is in the cache, or has to be read from the disk subsystem.

The only difference is the elapsed time. Finding the data in cache can provide a response in 1 Ms, but a cache miss often

takes more than 20 Ms.

Pool Performance

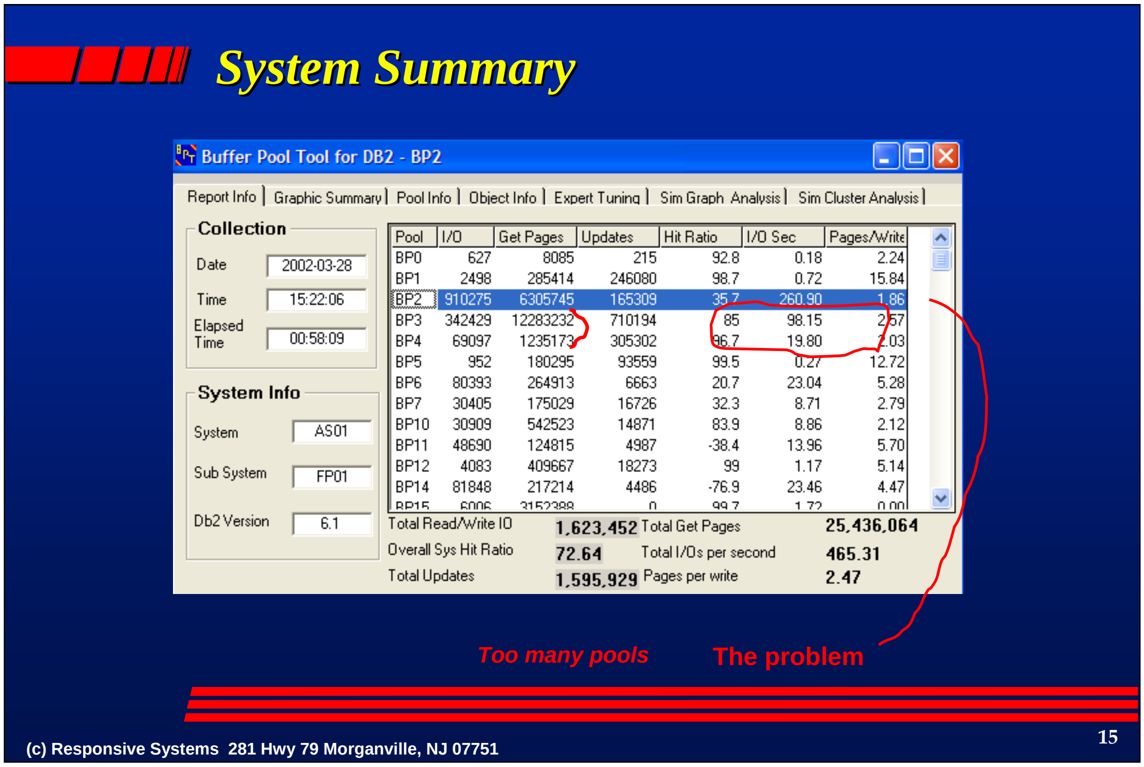
- **Larger pools – more memory**
  - **Do not always improve performance**
  - **May increase system paging and hurt performance**
    - **Might look better statistically, but response time can be worse**

- **Memory is a *system* resource**
  - **DB2 is not the *only* sub-system on your processor**
  - **You will run out of memory on a 64 bit machine too…**

Just throwing more memory at pools does not always provide better performance, and I will show you data to illustrate this.

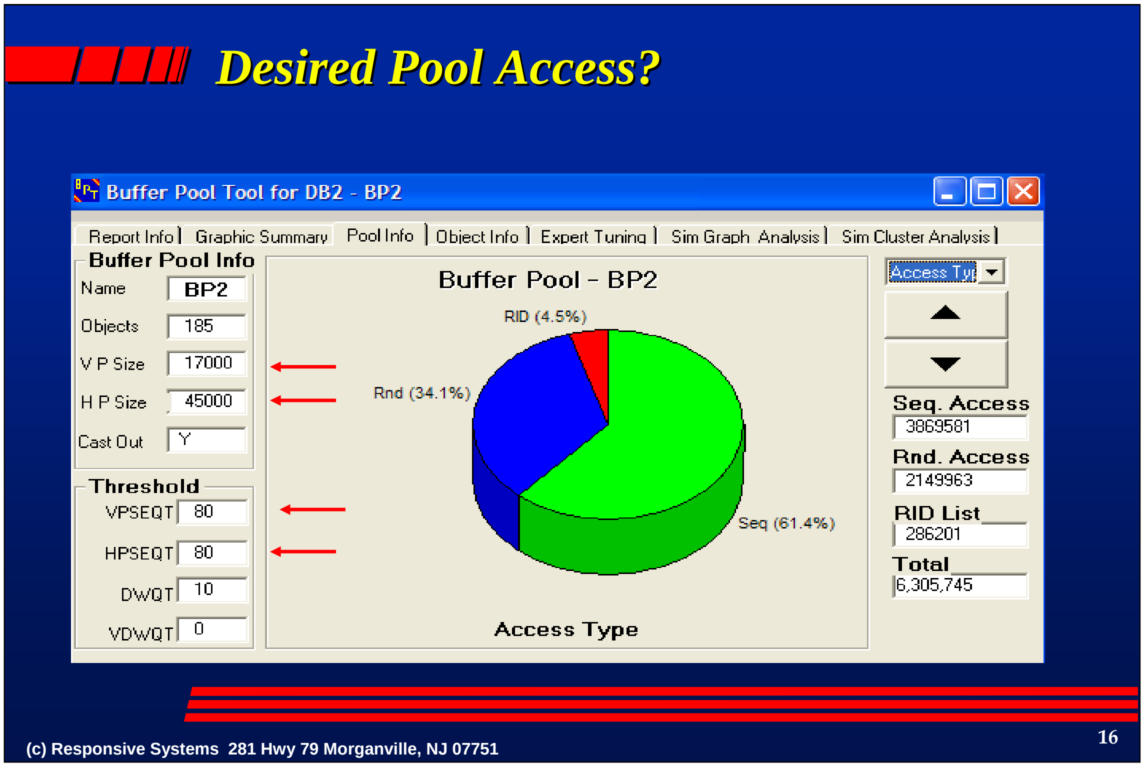
Over allocating overall memory resources may initially look like you are getting better performance because the I/O rate drops, but

if the system paging rate increases too much, overall response time will be worse.



Use the Eyeball method of problem analysis…. What’s a big value compared to everything else?

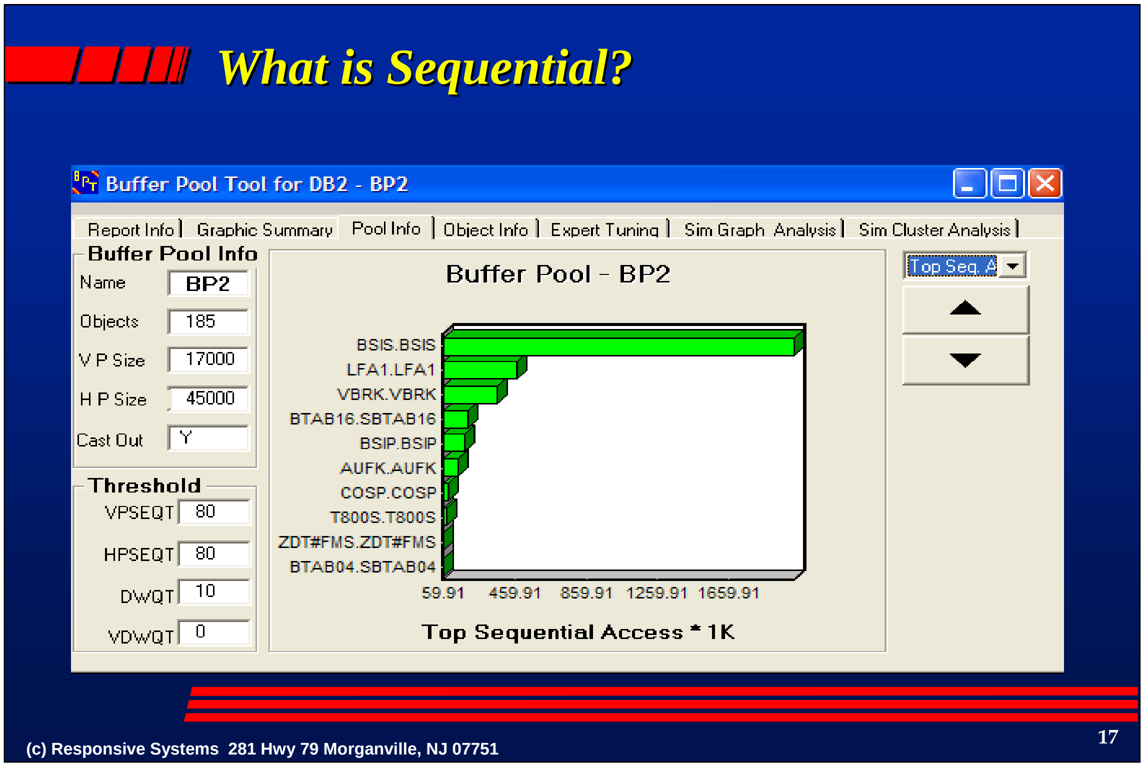
Note that BP3, that has a high 85% hit ratio, higher than BP10, shows a high I/O rate.



What is the “desired” access we want to see for this pool? Is this what we expected?

The pool has a large number of buffers, yet it has a very high I/O rate.

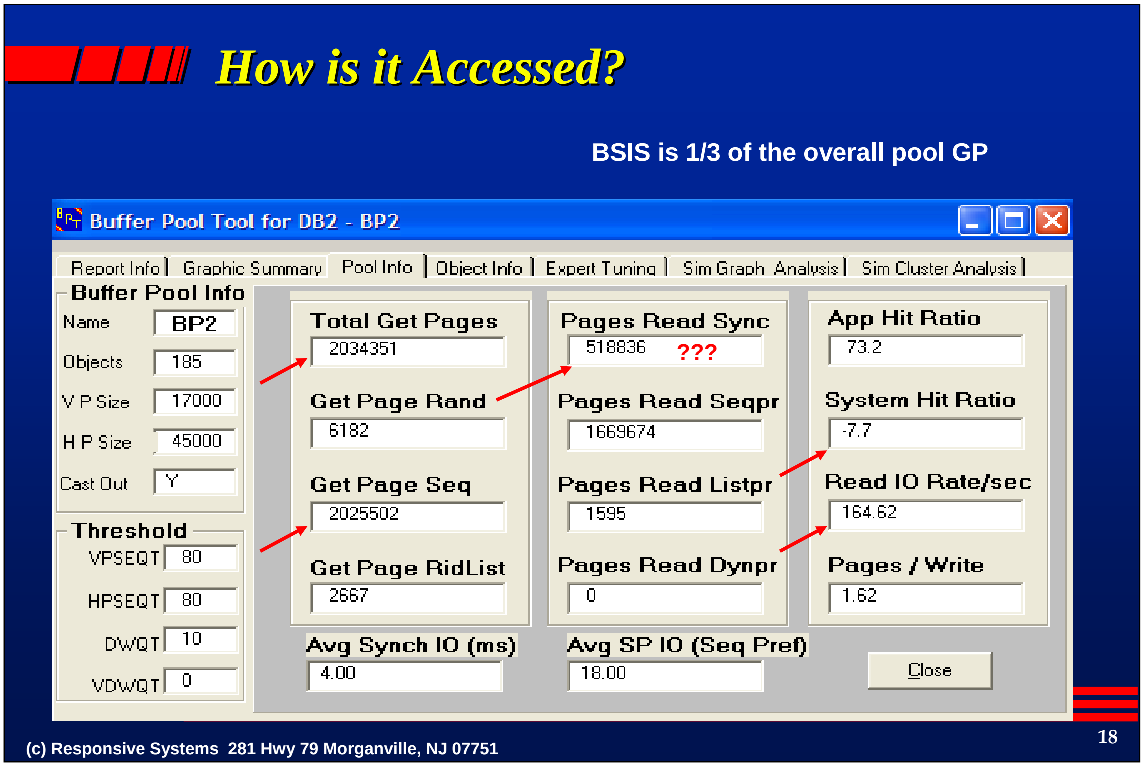
What is causing the I/O rate to be so high, and what can we do about it?



Since we saw very large amounts of SP, we want to see which objects are monopolizing the pool with this type of access.

Lowering the vpseqt and hpseqt will not reduce the amount of SP activity, but will prevent it from forcing as many random

Pages out of the pool.



Object BSIS has, by far, the greatest amount of SP activity. Now what else can we find out about it?

This object is also causing 148.7 I/Os/Sec using Synch I/O. Why are we seeing this?

The number of SP I/Os is about 10% of the number of Synch I/Os.

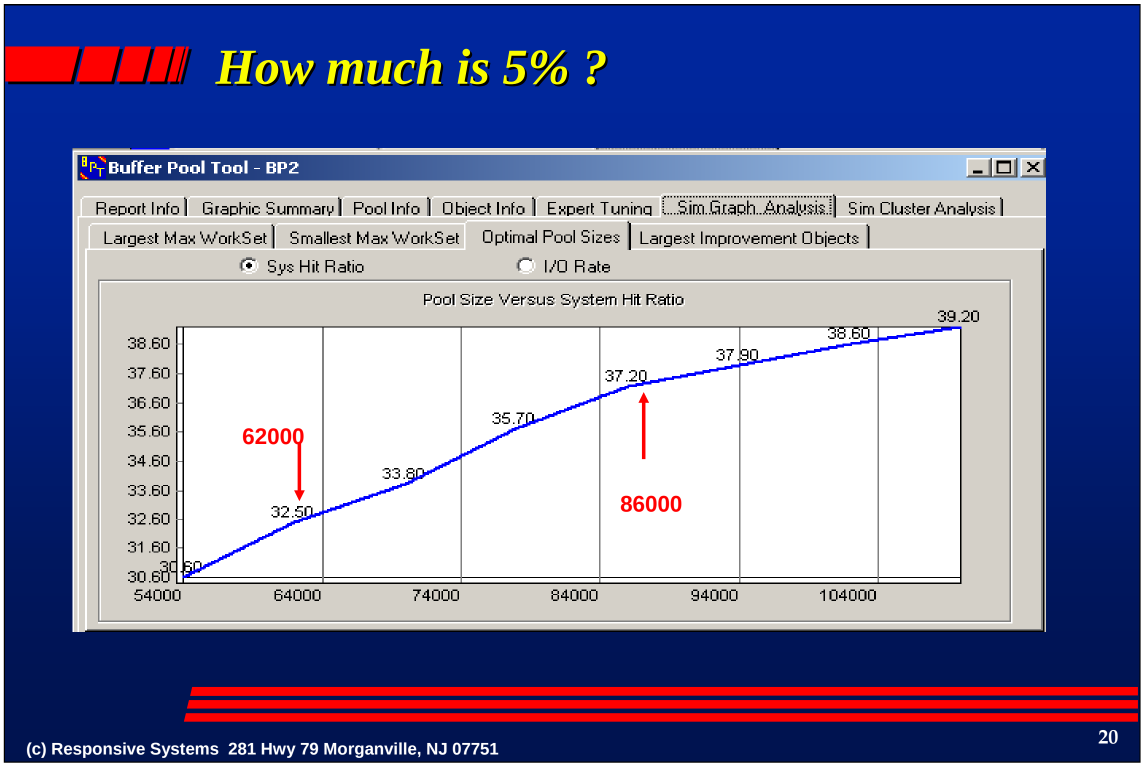
How can we Eliminate I/Os?

- **Give the pool more buffers – memory**
  - **How much will we need?**
    - **How much can we gain?**
  - **Is it available on the system?**
- **Move objects into different pools**
- **Create a useful Index on the Object**

There are three options to reduce the heavy I/O activity against BP2.

Creating a useful index on the object will provide a large reduction in the number of pages referenced, and the

greatest overall performance improvement – and cpu reduction. Unfortunately, this may not always be possible.

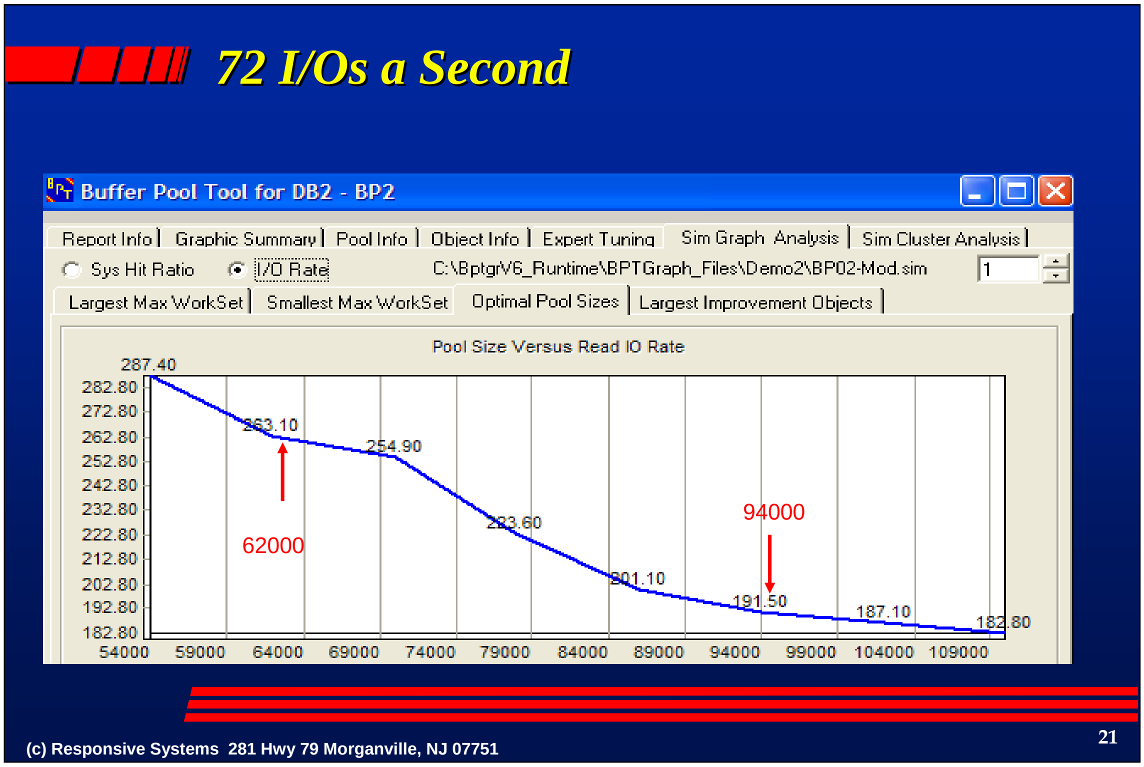


The current pool has 62,000 buffers in total. What can we gain by throwing more memory at it?

If we look at the Hit Ratio, an additional 24,000 buffers (96 Meg) gets us 5%, but the slope of the curve flattens after that, so even more memory provides diminishing returns.

Just what does 5% really mean? Can we convert this into an elapsed time saving, or CPU saving?

The answer is – **no**.

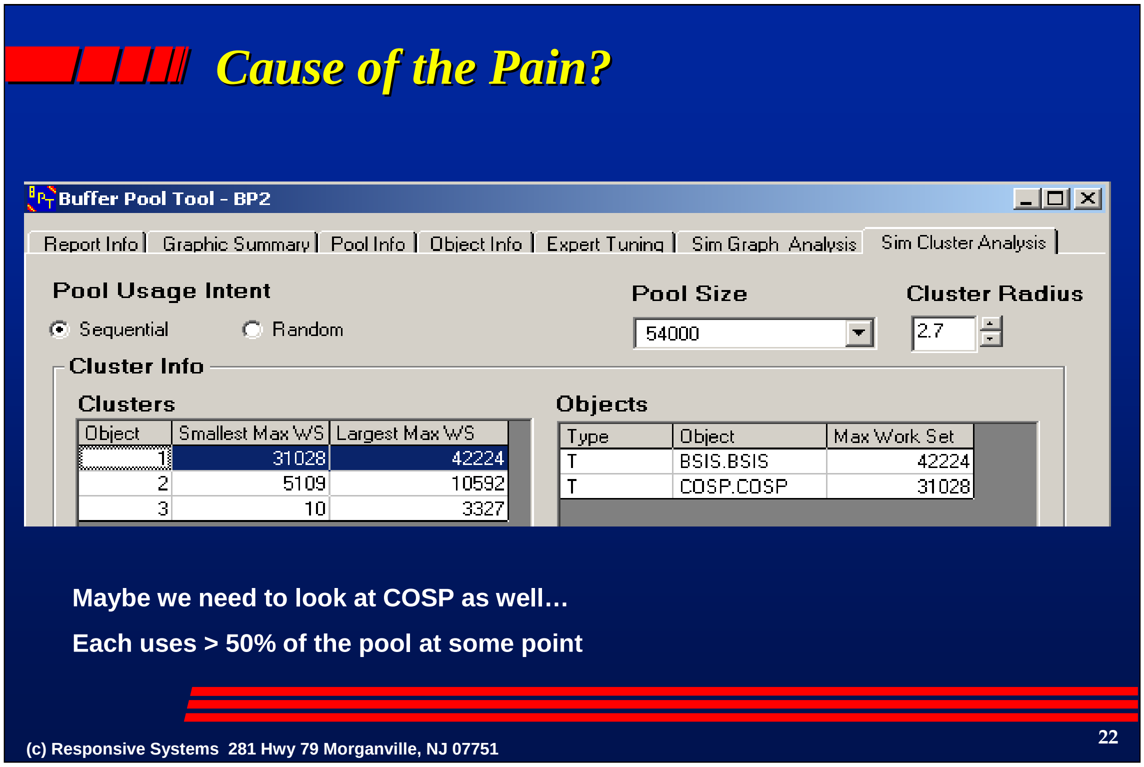


Looking at the I/O side, we can see the saving of I/Os/Sec.

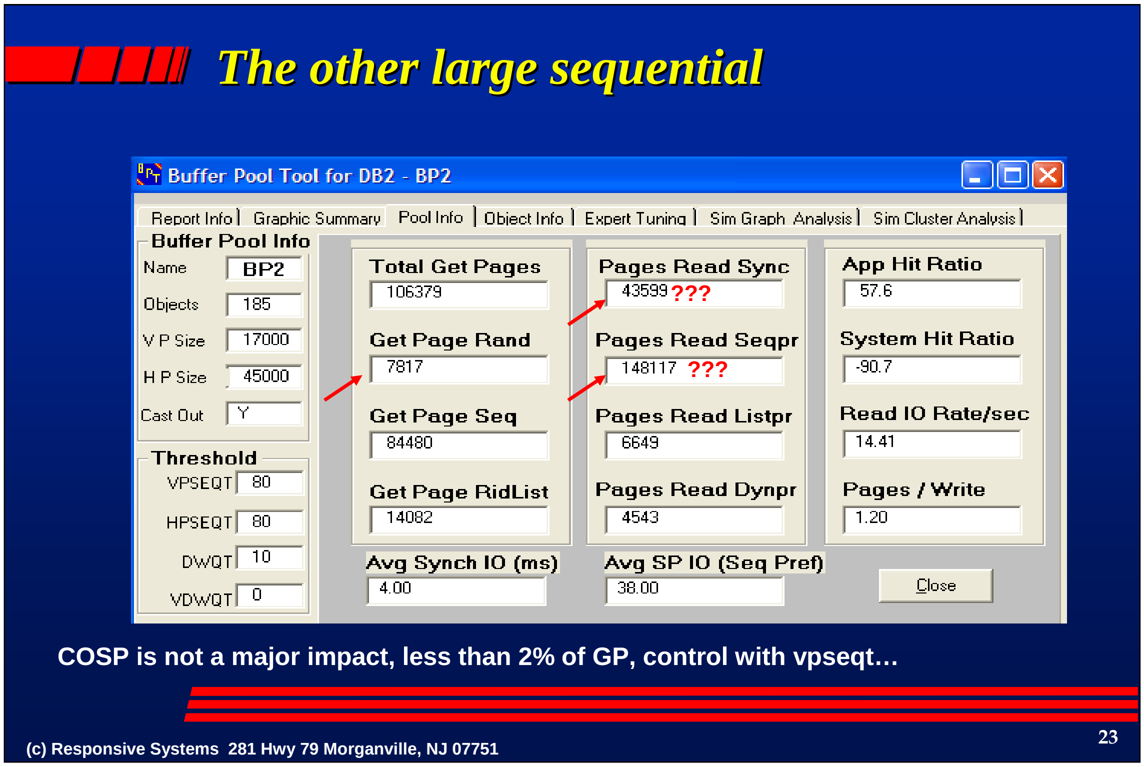
Since the duration of the data we were looking at was 58 Mins and 9 Sec, this is 3,489 Sec, times 72 I/Os = 251,208 I/Os.

Object BSIS showed an average Synch I/O time of 4 Ms, and this is most of where our saving would be.

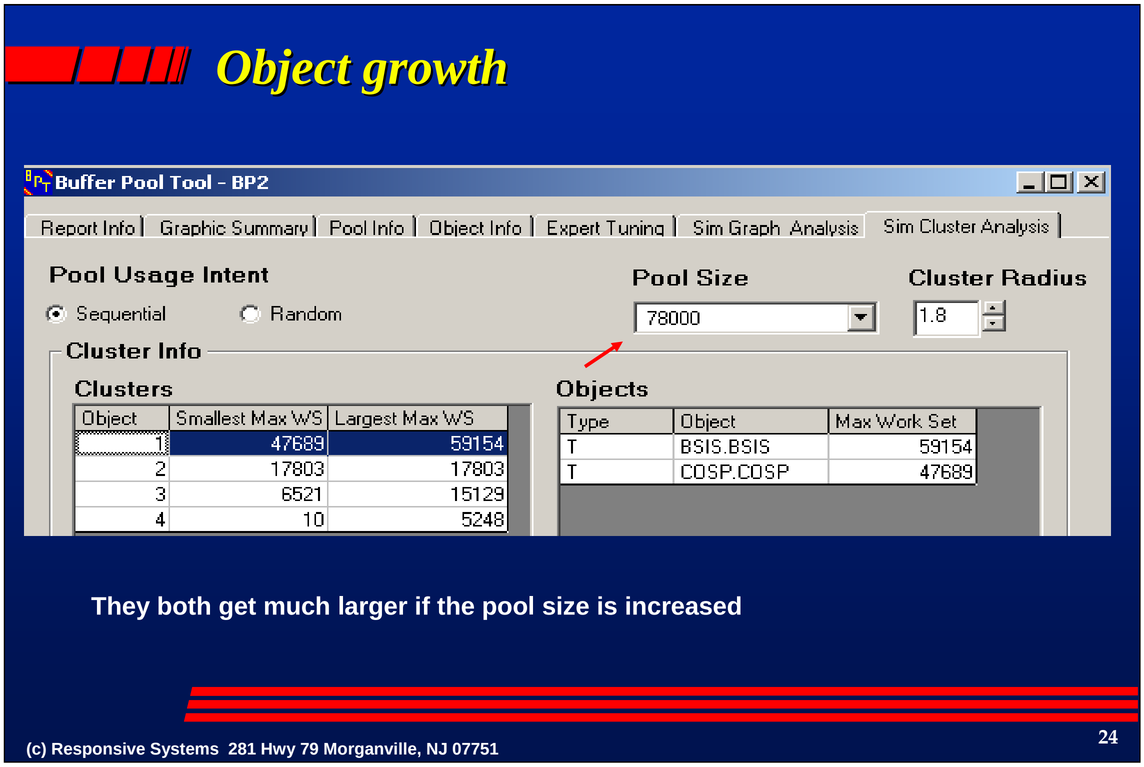
So if this might be representative of of a batch jobstream, this will reduce the elapsed time by almost 17 Minutes.



We can't stop with BSIS, let's look at the number two object as well.

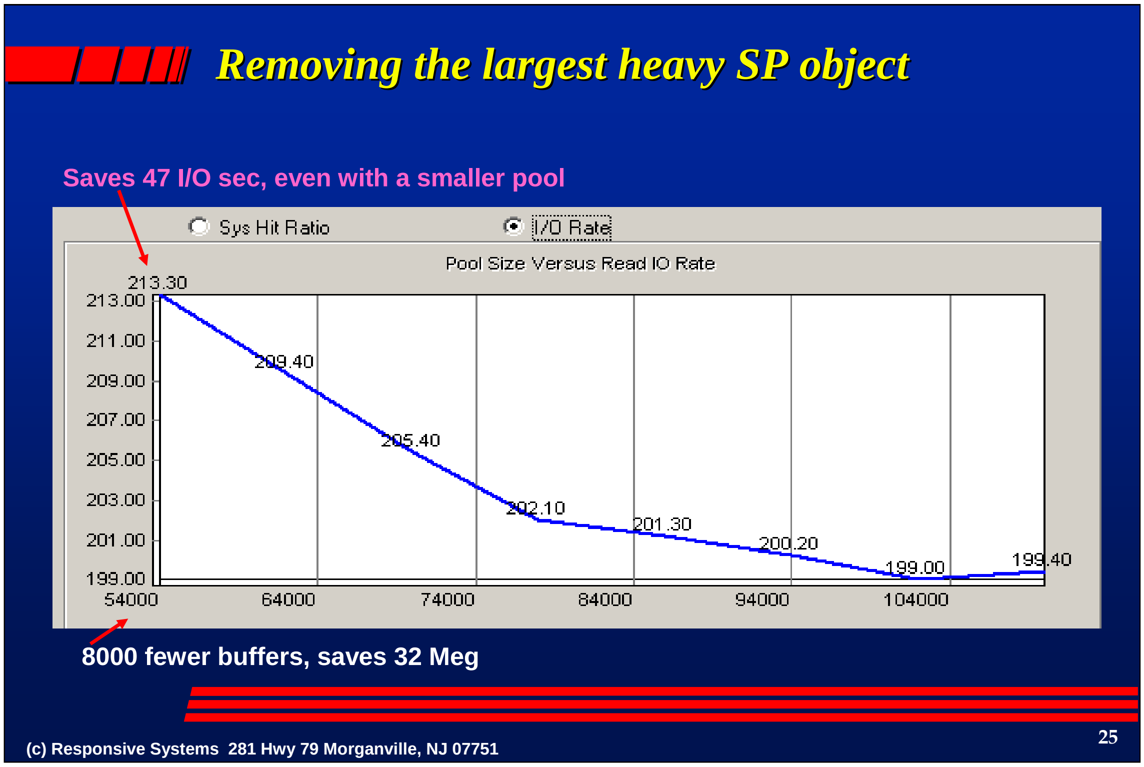


Again we see mostly SP, but a high number of Synch I/Os.
But, we issued 106K getpage requests and read in 198K pages.

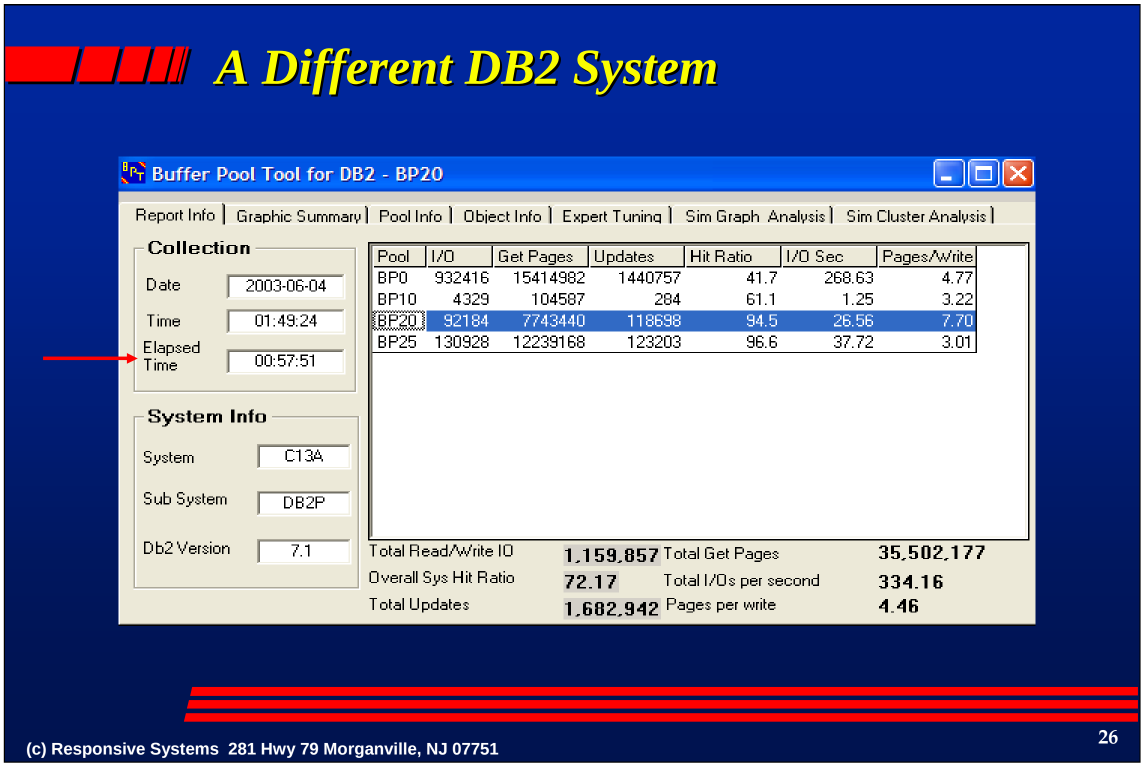


Increasing the number of buffers in the pool from 54,000 to 78,000, and the objects still monopolize the pool.

So just giving the pool a lot more memory will not help performance a lot, since these objects will continue to monopolize the pool resources.



Removing object BSIS can save 42 I/Os per second against the other mostly random objects, while saving 32 meg of memory by reducing the pool size by 8,000 buffers (if there are system memory constraints). Object BSIS can be placed into another pool with sequential objects, or a new pool for sequential access. Objects that are heavily scanned all the time do not need large memory allocations (unless you can make the pool larger than the entire object).

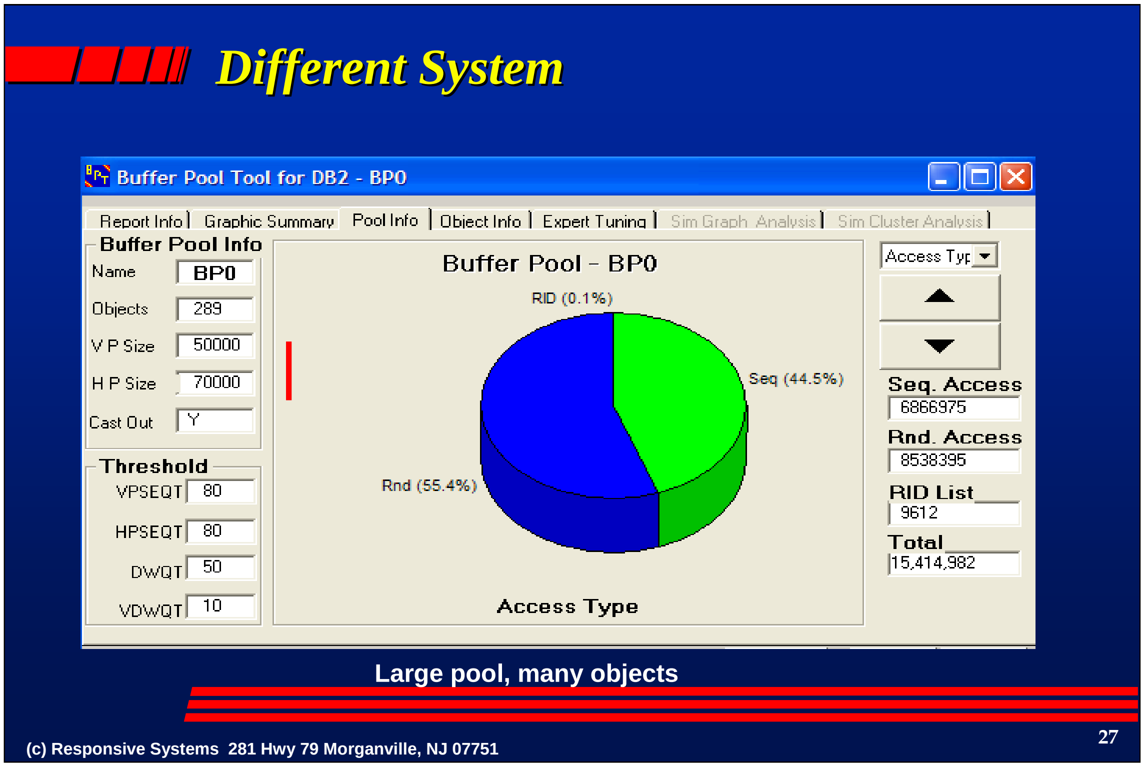


Let's look at data from a different DB2 subsystem.

This system is not using enough pools, and has not separated the objects out of BP0.

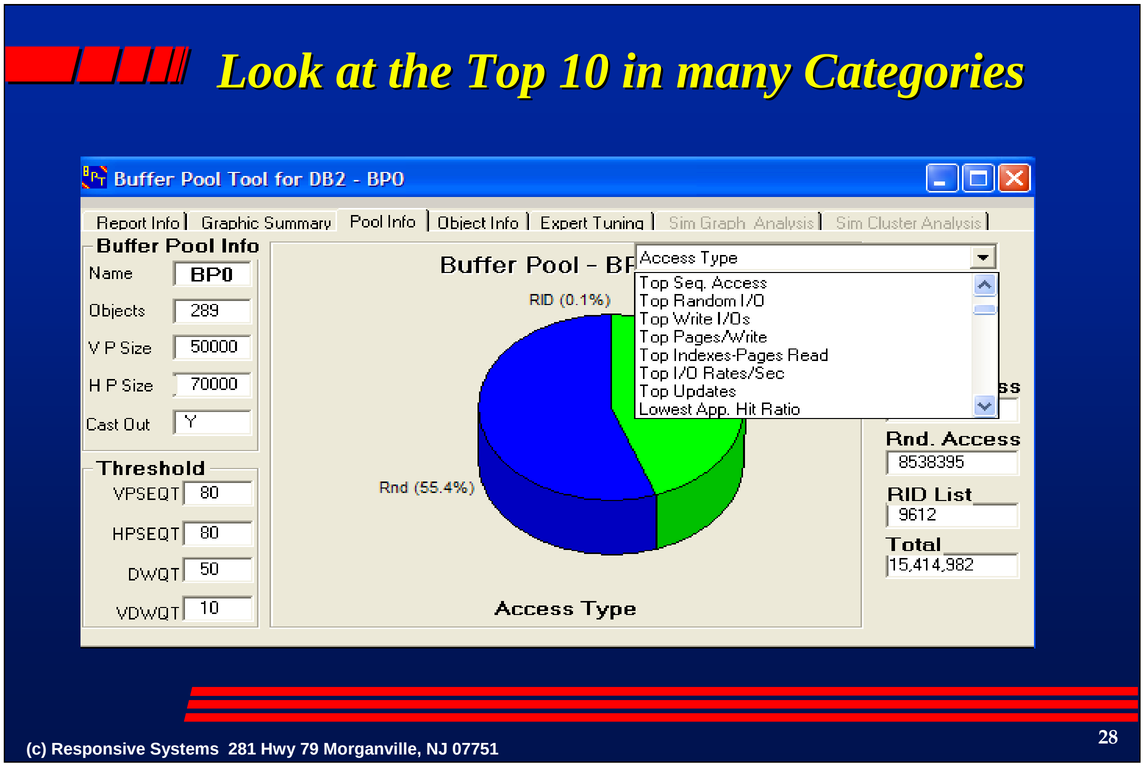
BP0 is generating 80% of the entire system I/O.

This is probably part of an early morning batch processing scenario.

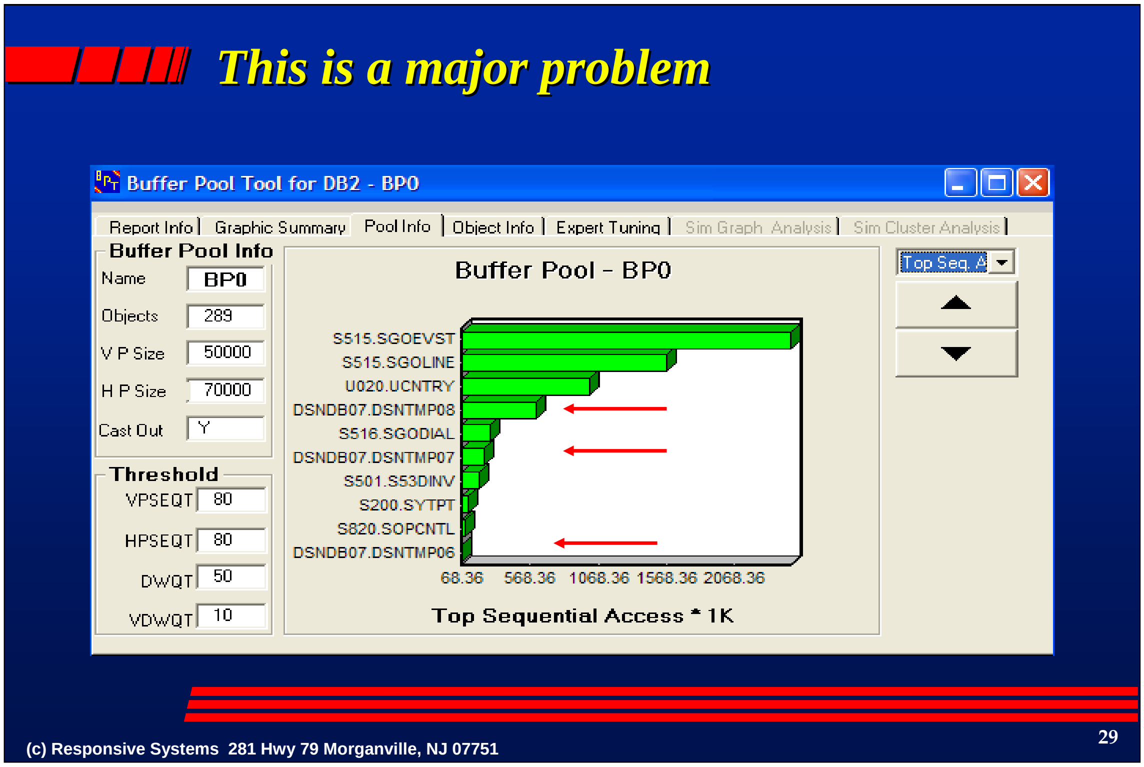


BP0 has a large number of buffers allocated, in an attempt to reduce I/O and help performance.

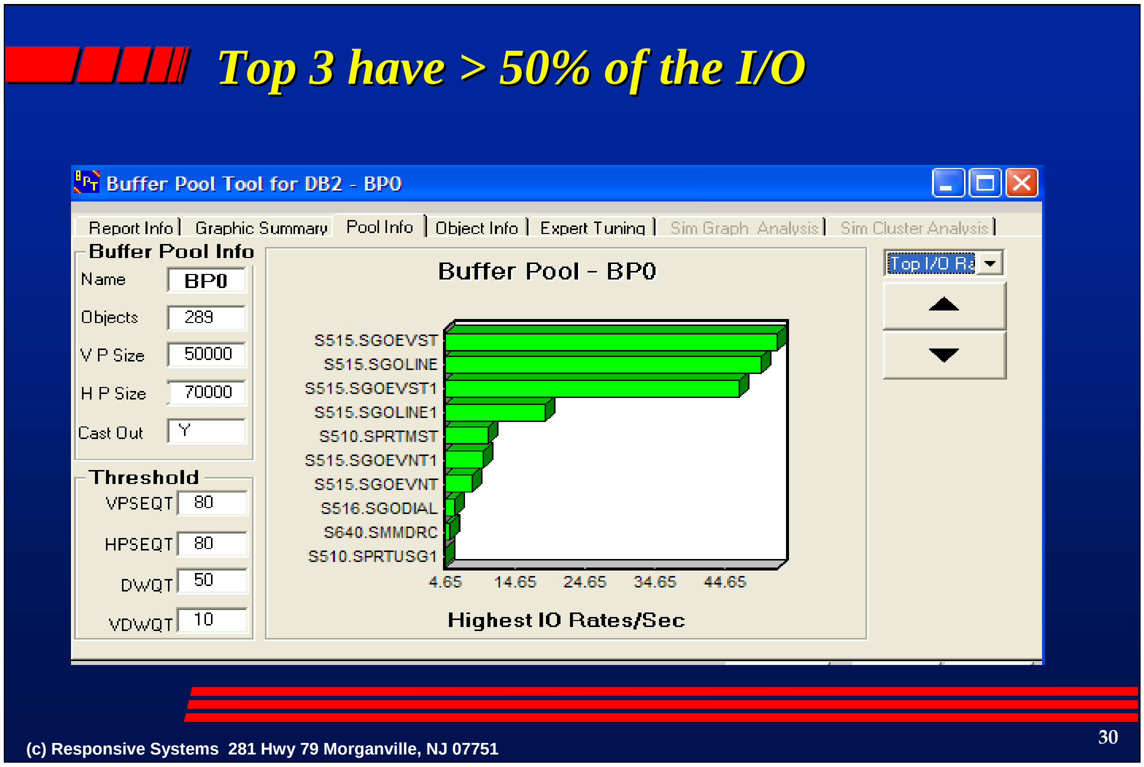
A lot of the objects in the system (289) are in BP0.



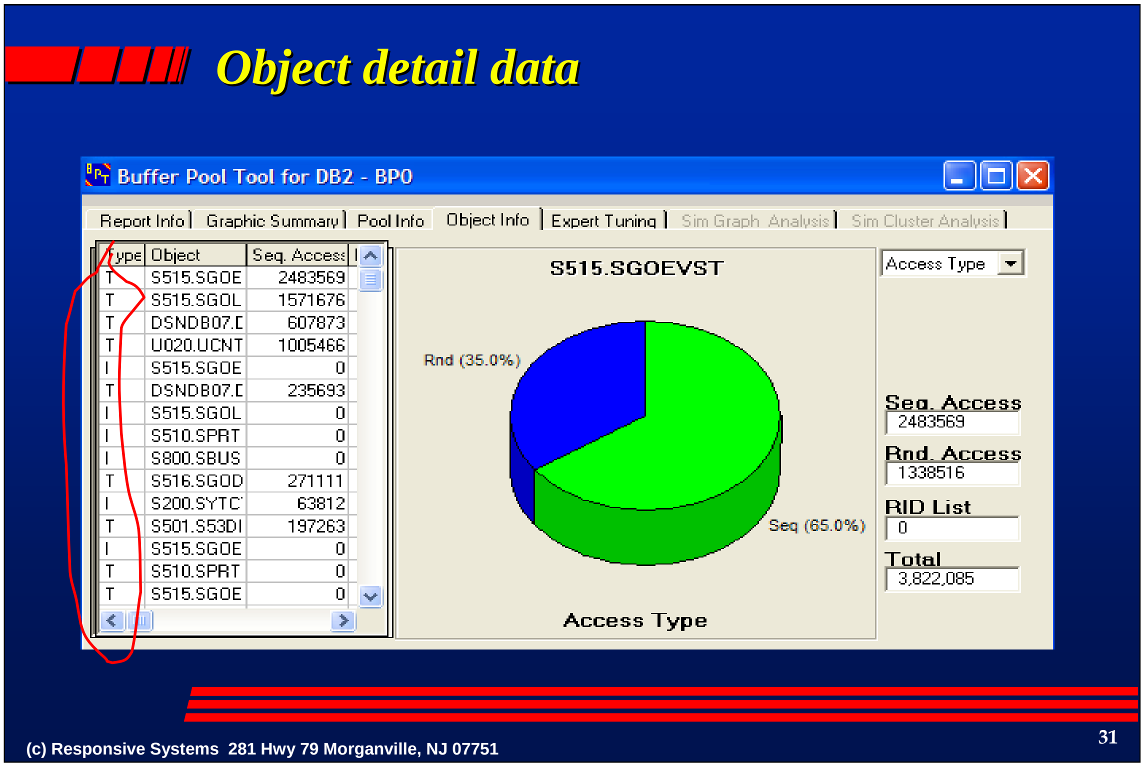
45% of the I/O is sequential, so the objects we are most concerned with, are those with the most sequential access – that are hurting the randomly accessed objects. We also want to look at the objects with the highest I/O rates,



Hey guys, we need to get back to tuning basics….. Separate the Sort objects into their own pool.

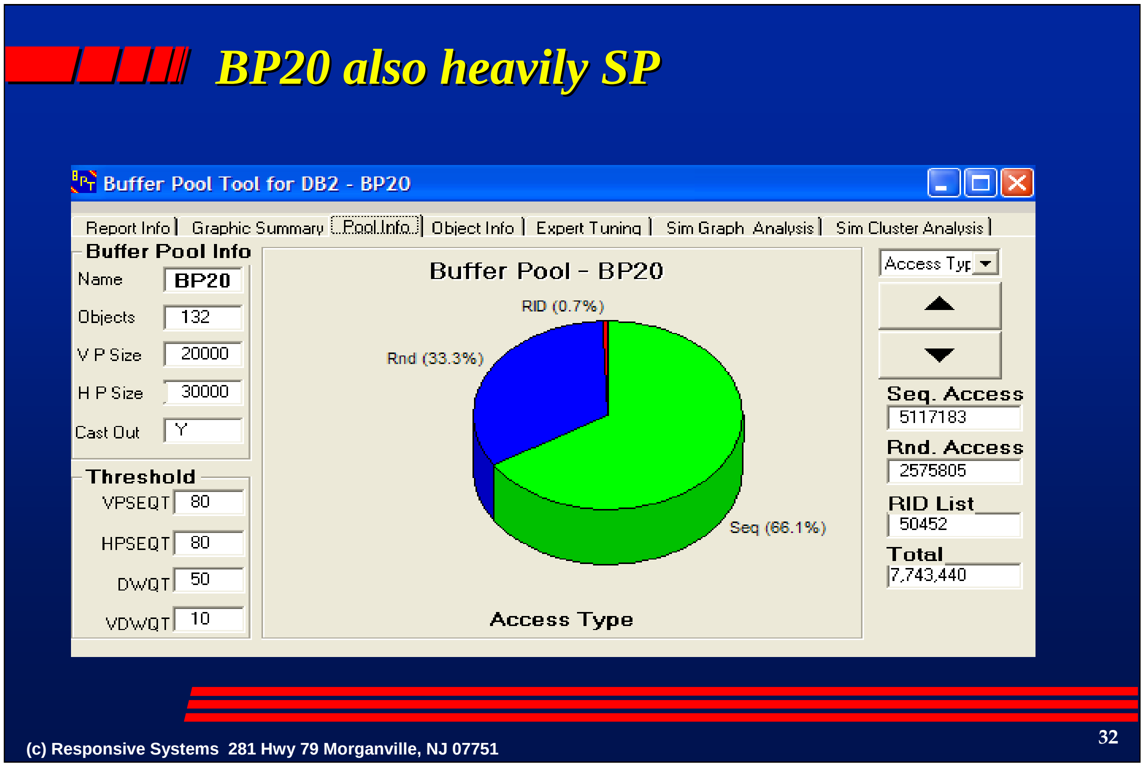


The top two sequentially accessed objects, are also the top two in the I/O rate/second category. This is not always the case in every system.

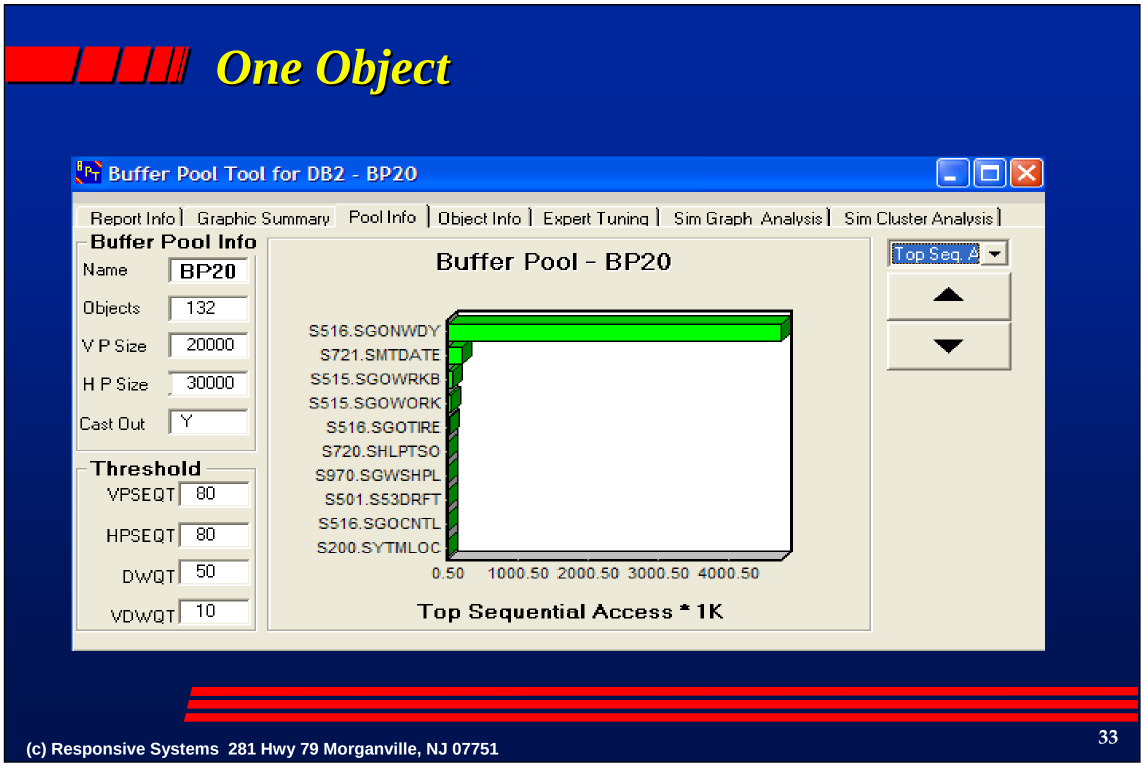


We also have most of the indexes mixed in BP0 with the tablespaces, and the sort objects.

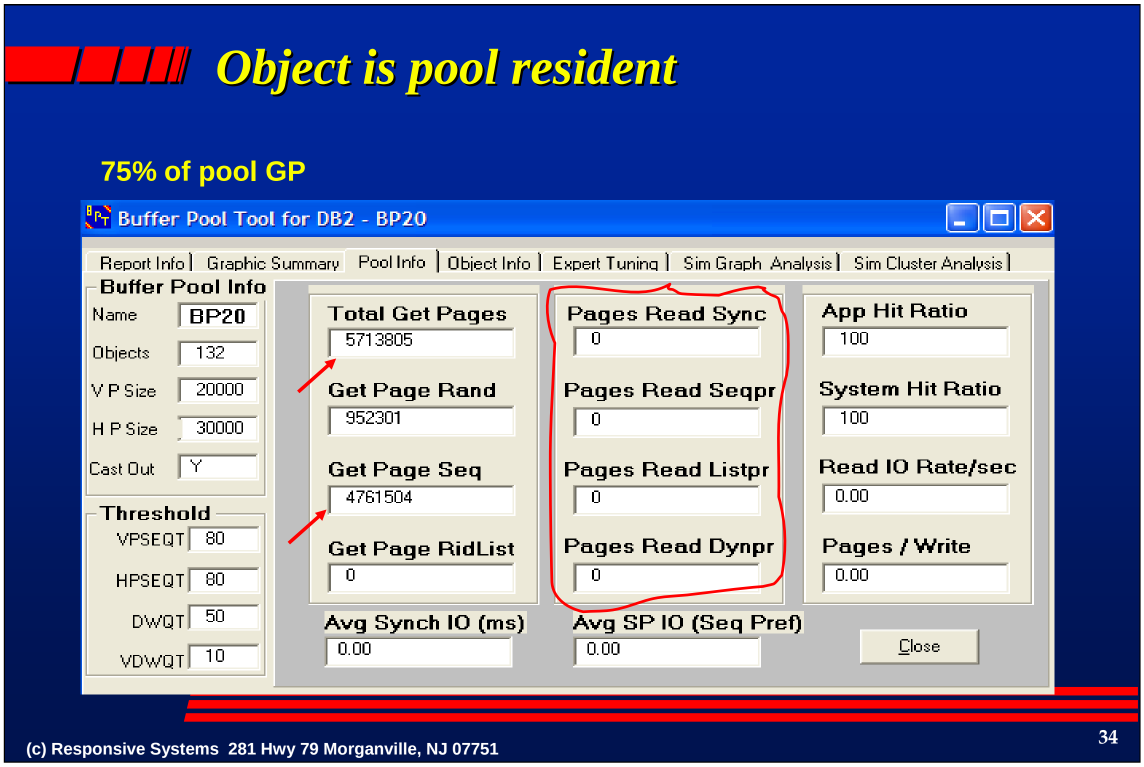
Developing some useful indexes on the top two objects will provide the highest benefit for reducing both I/O and CPU.



BP20 is also large, and heavily sequential, with the thresholds set too high.



One object is causing almost all the SP activity.



This object has 5.7 million getpage requests, and 4.7 million of them are sequential.

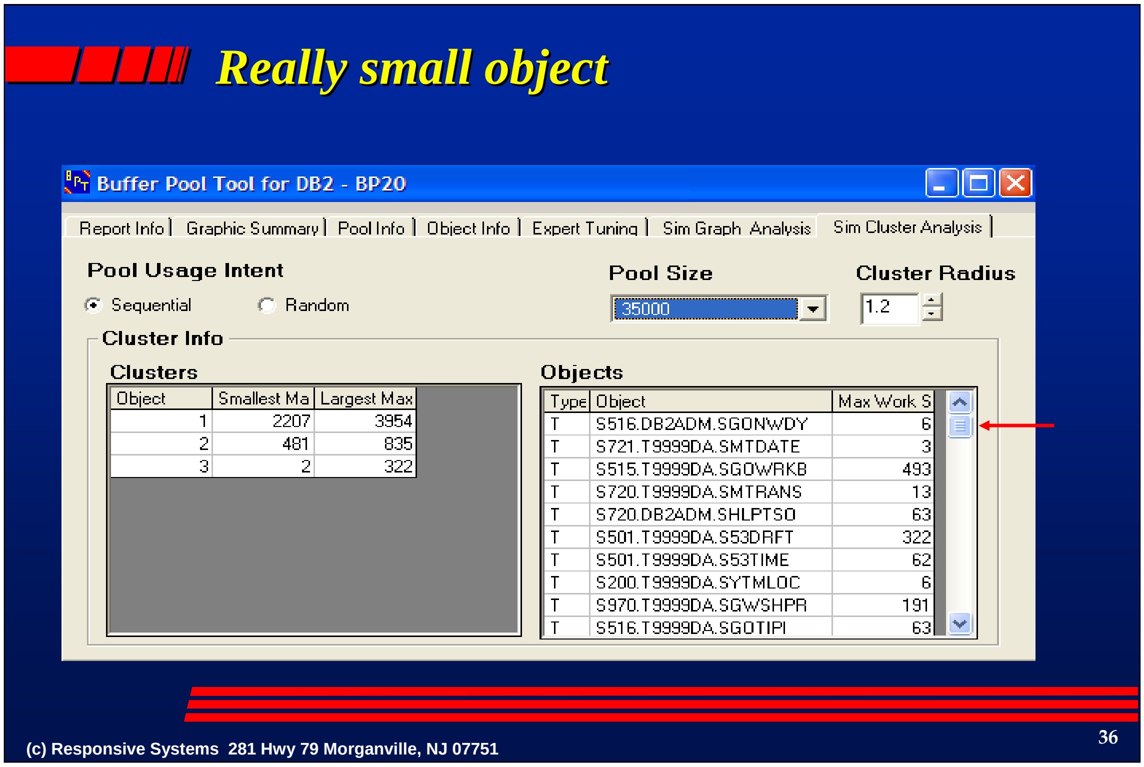
The object is pool resident, with no I/O activity.

This is a real CPU burner!!!

First thought for this problem

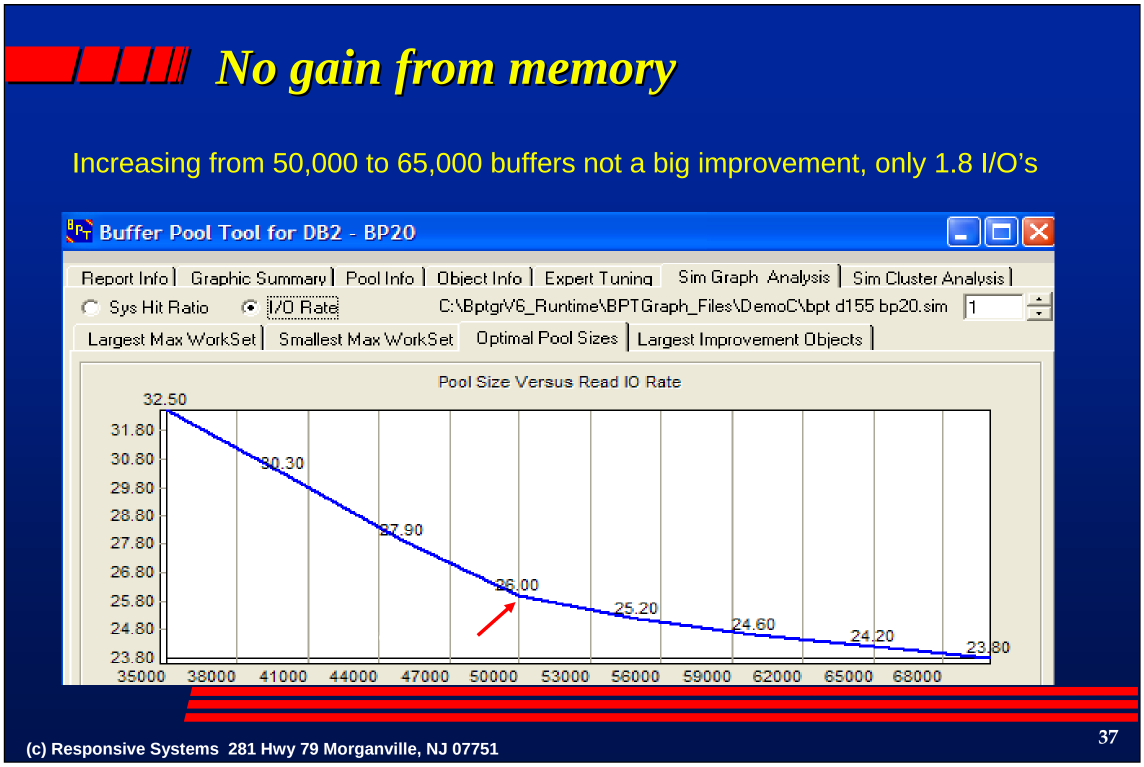
- **Add an Index the application can use to eliminate the scan activity**
- ***A first guess may not always be right….***
- **Let's look at some more information about this object**

Let's dig a bit deeper into the available data before jumping to easy conclusions.



The object has only 6 pages that are getting all that access and scan activity. But, since it is so small, it is not impacting the overall pool performance and I/O rate. It ***is having a huge impact on CPU cycles*** because of the number of pages scanned.

Now, if you noticed, the data was collected starting at 1:49 am… an obvious batch cycle period. So a good possibility, is that a batch program is repeatedly accessing this object for every transaction/process it executes.

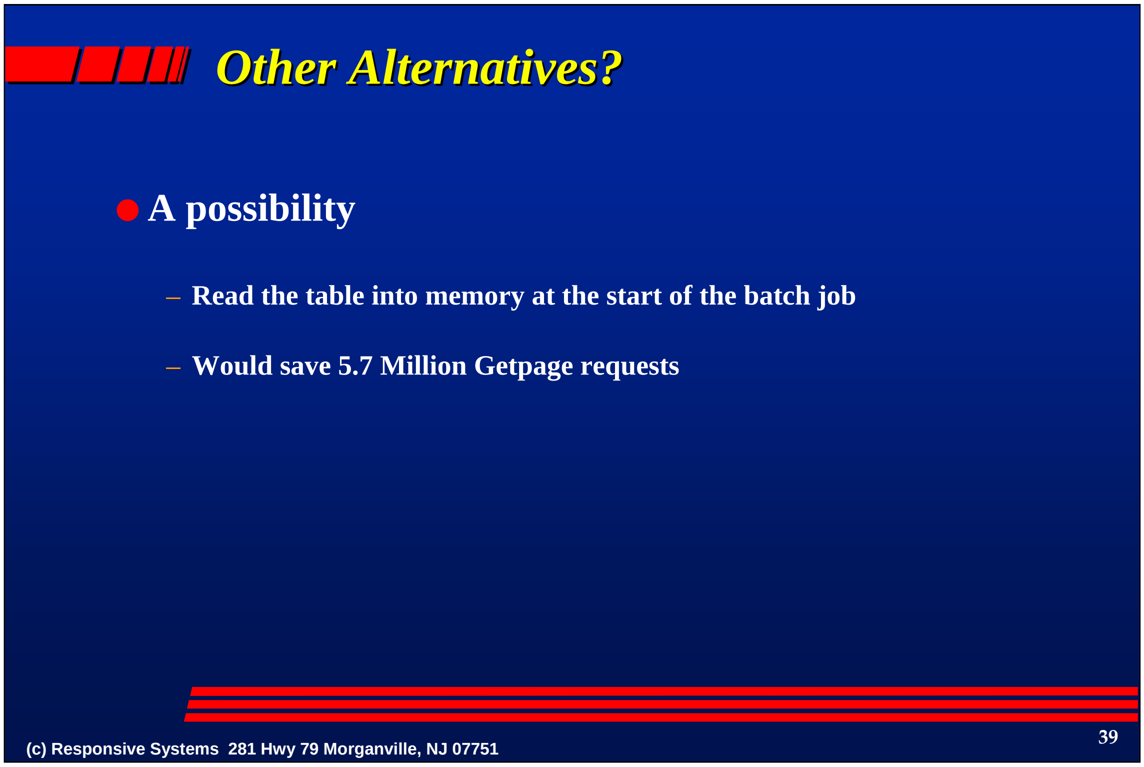


The pool currently has 50,000 buffers. If we added 15,000 buffers, 60 megabytes of memory, this would only save 1.8 I/Os a Sec. Not a worthwhile usage of memory. That very heavily scanned object is not having any real impact on the I/O rate, because it only occupies 6 pages.

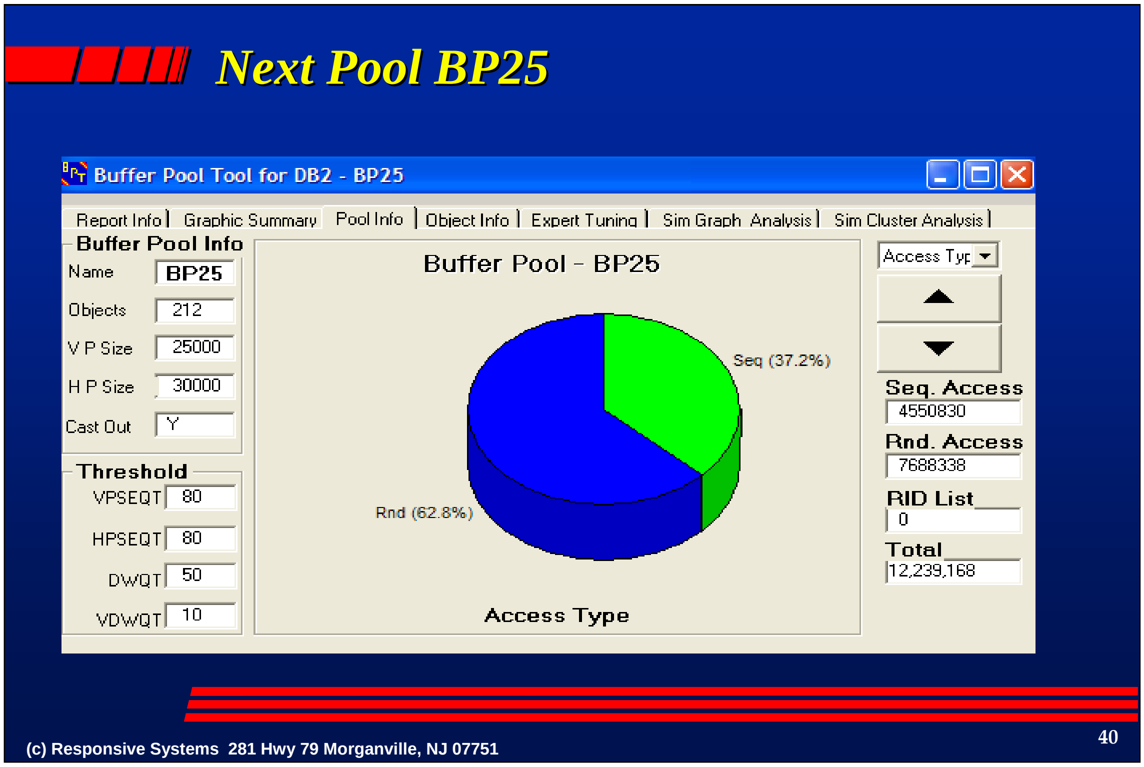
Gain from an Index?

- **If it would go directly to the desired page**
  - **It would reduce the data access Getpages by 5/6**
    - **But it would add an Index access**
      - **Maybe Index only access?**
- **Saving….**
  - **3,950,000 Getpages for Index only access**
  - **But saving only 3,135,000 Getpages if data access required**
- **Adding another Index will add some overhead to other processes**

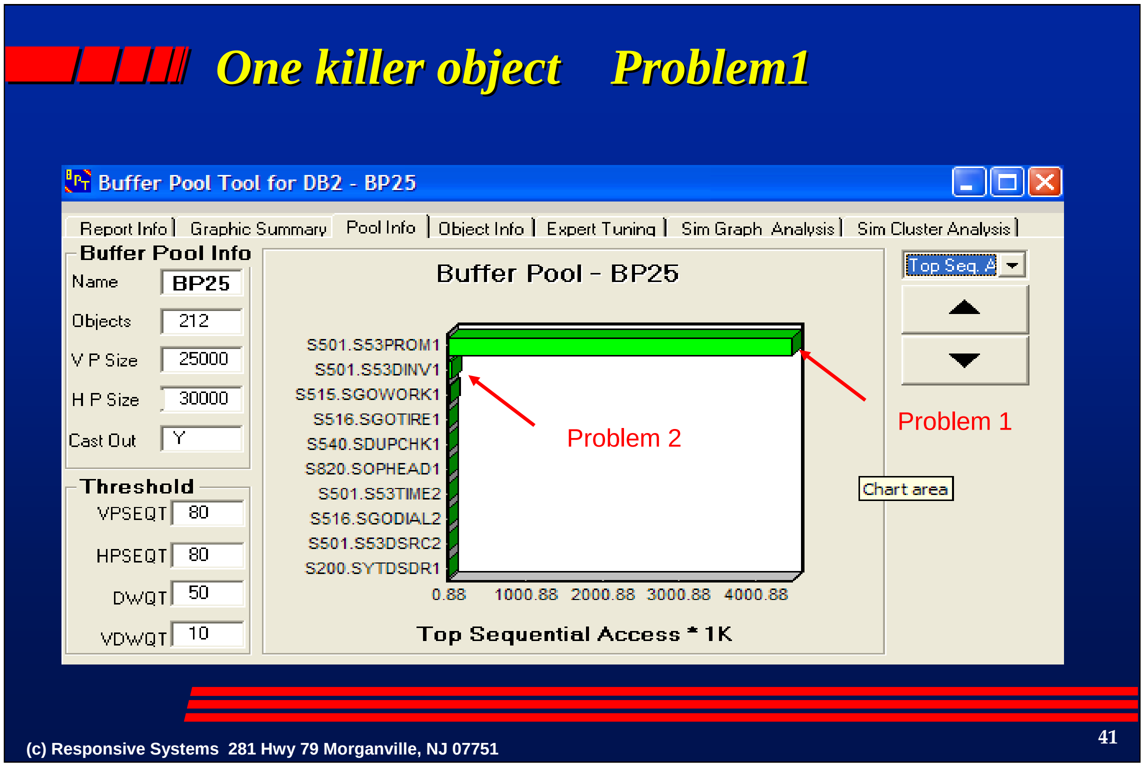
Adding another Index would save more than 3 milliion Getpage requests, but might add some overhead to other processes if they update the data.



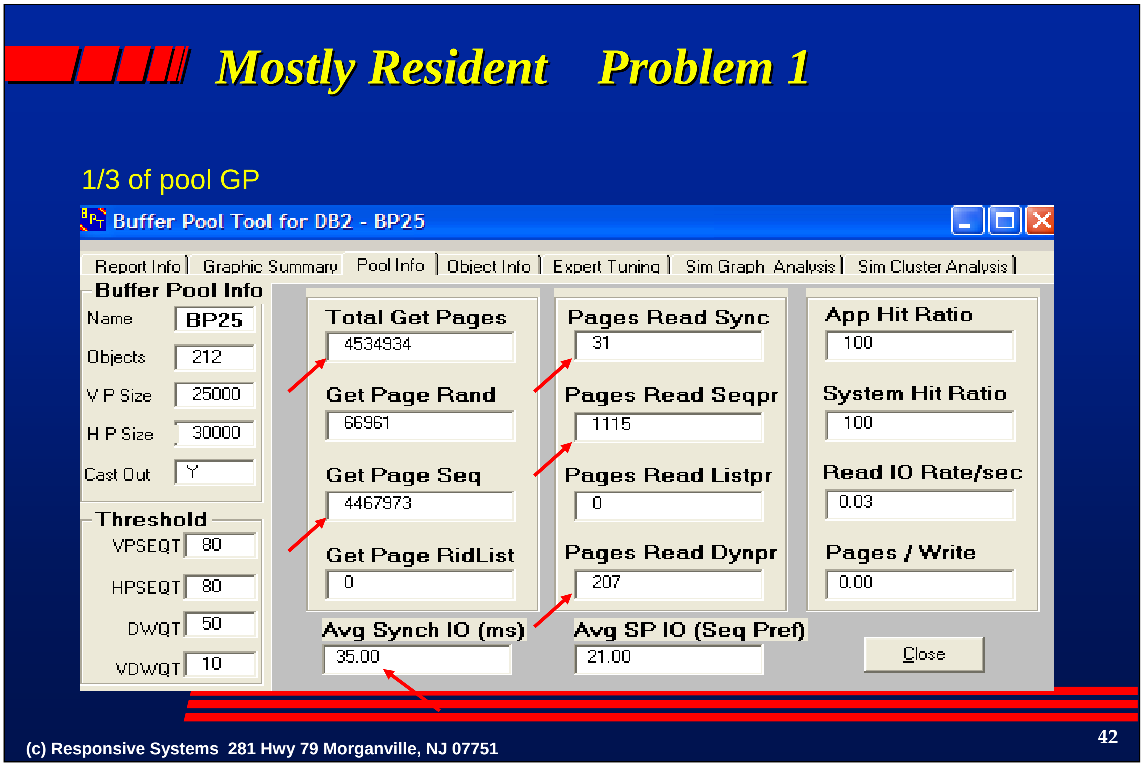
Would eliminate a huge amount of Getpage activity – but might not be feasible if any other processes update the table.



Again we see a pool with a lot of buffers, and having a lot of sequential access. The sequential thresholds should be lowered to favor the randomly accessed objects.

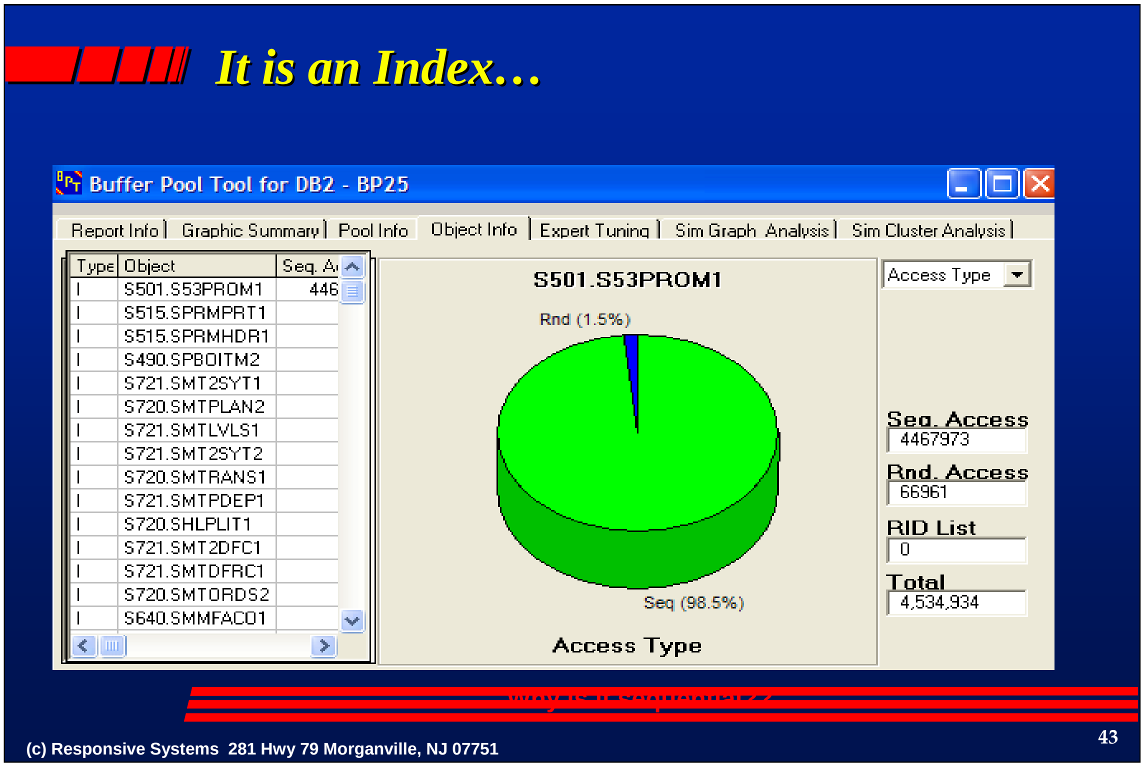


Aside from the obvious problem object PROM1, DINV1 will also show some interesting problem information.



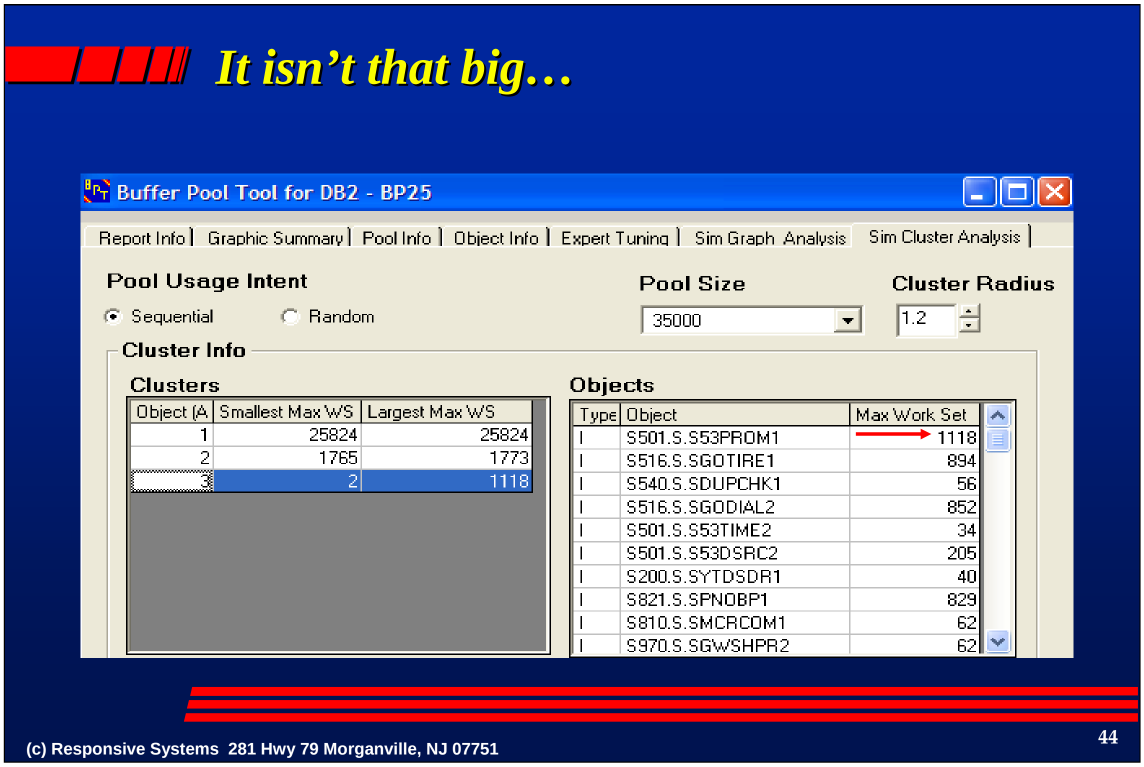
Prom1 has 1/3 of the pool getpage activity, and that's 4.5 million getpages – and almost all sequential activity.

When synch I/Os are necessary, there is a cache miss and poor response time.



PROM1 is an Index….. So the huge scan activity indicates either poor design, or an sql coding problem. This is an opportunity to achieve a large cpu saving from reduced getpage activity.

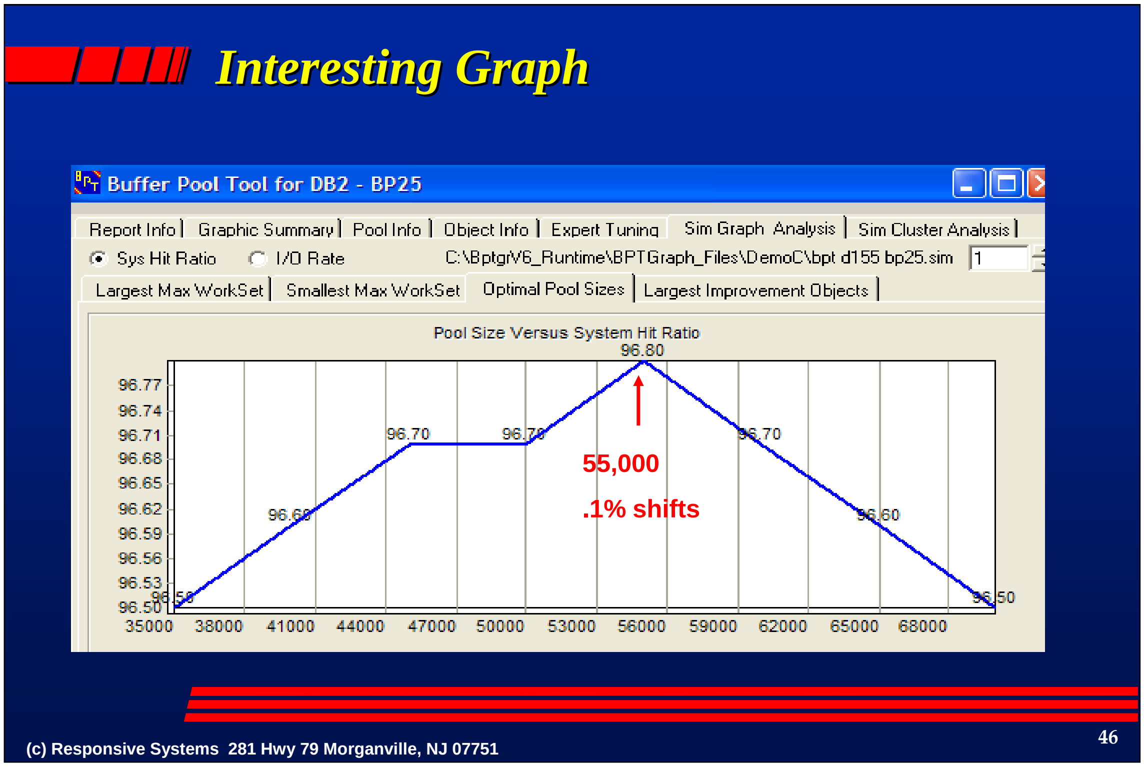
The object is mostly pool resident as we will see later too, from the wkset size.



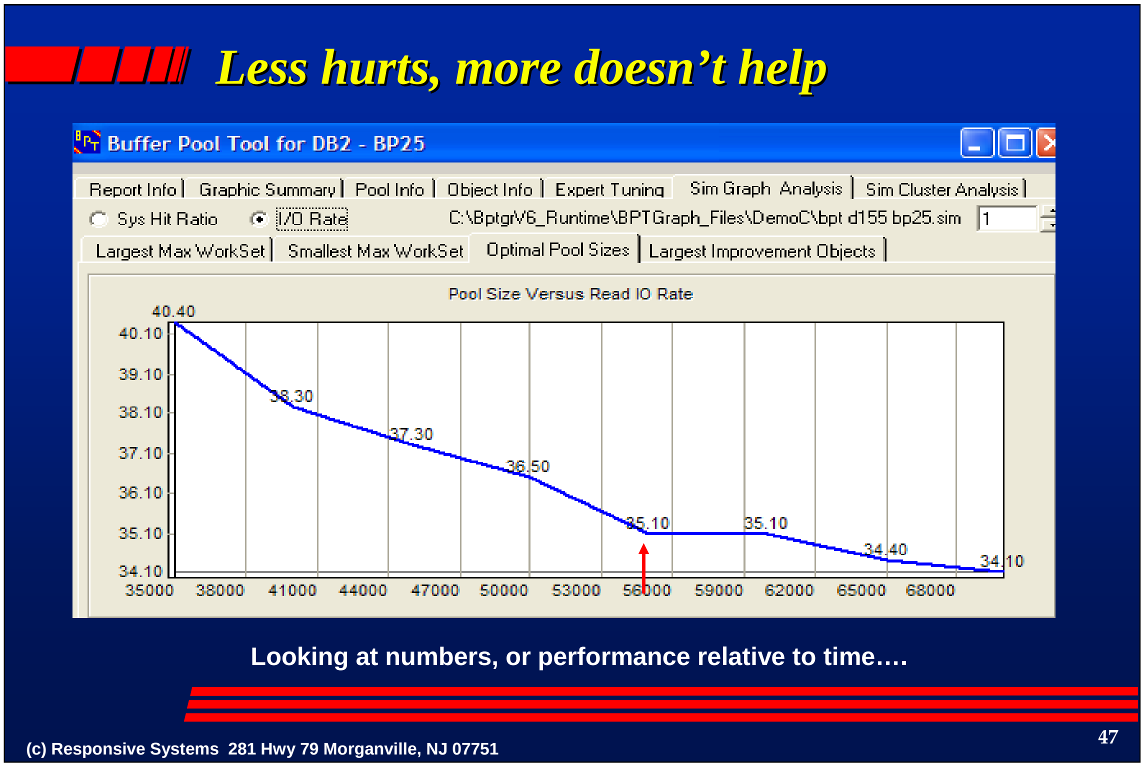
The working set (wkset) size of PROM1 only 1118 pages in a pool of 35,000 buffers.

Why are we scanning an Index?

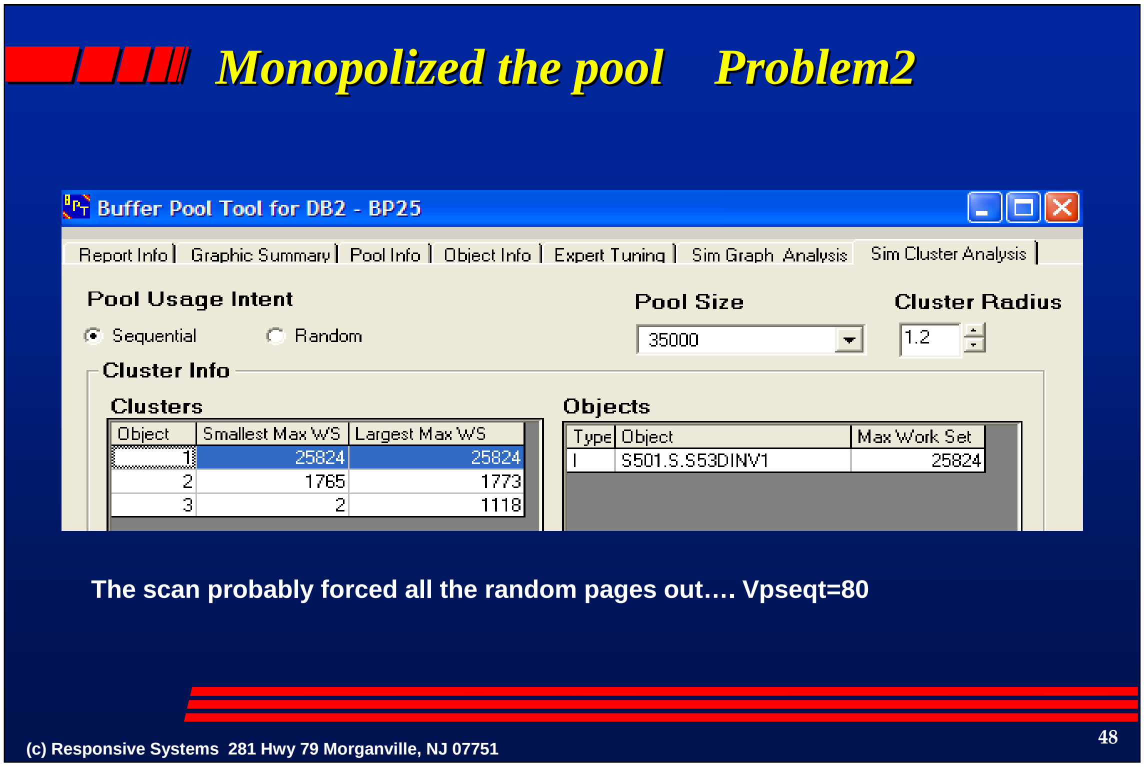
- Poor/improper Index design
- SQL coding problems
- Fix the application……



The pool currently has 55,000 buffers. So, even though the hit ratio is not the best metric to look at, it does show that reducing memory hurts performance, while increasing it does not help.

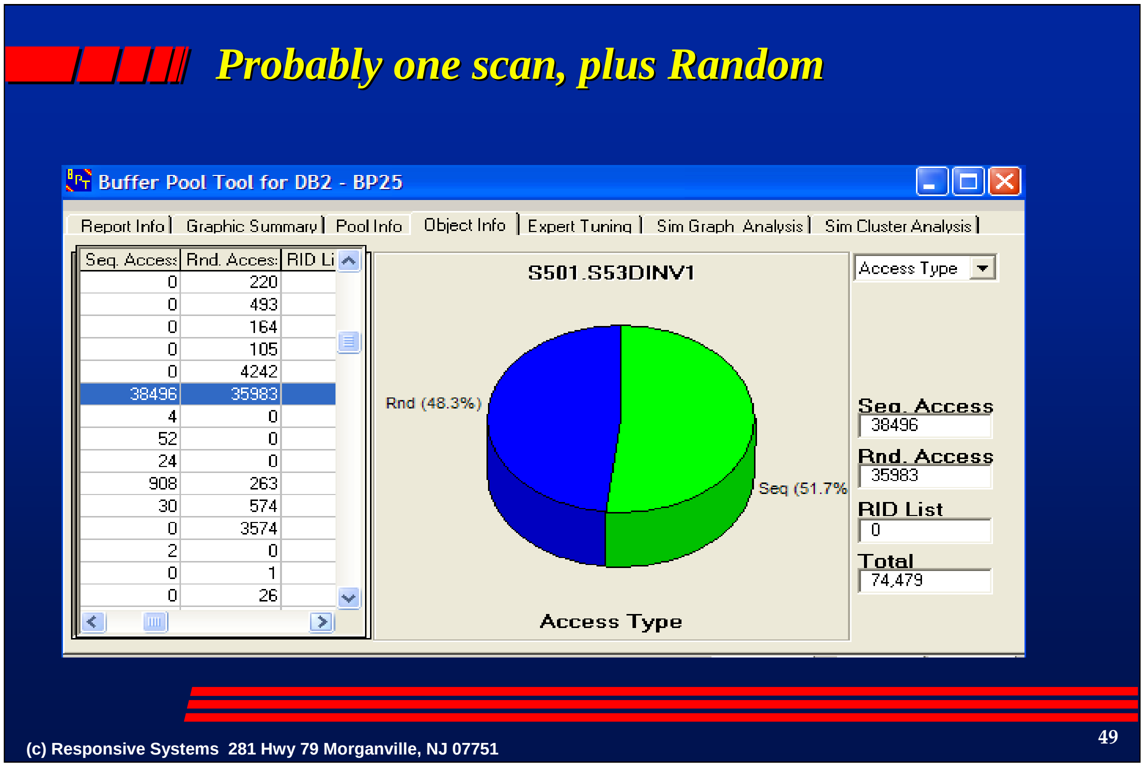


The I/O rate is much more indicative of the performance impact of reducing or adding memory from the current 55,000 buffers.



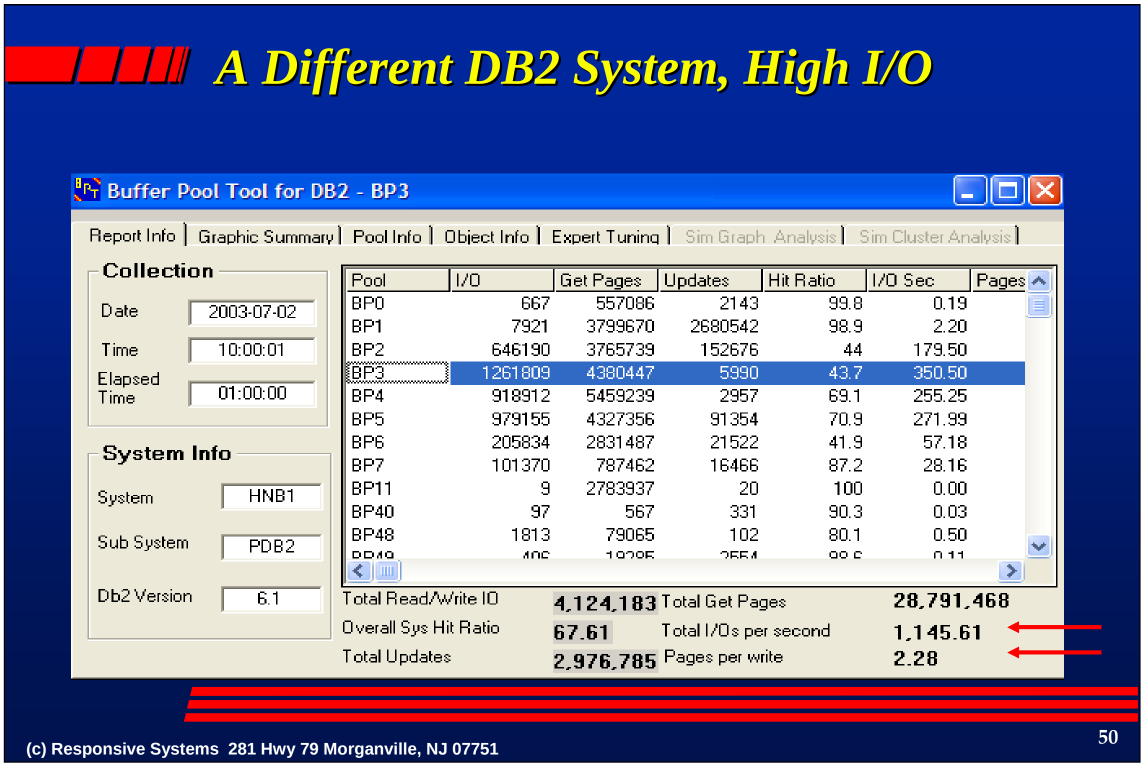
Even though the activity against DINV1 is much lower than PROM1, at some point it monopolized the entire pool, and forced out pages of PROM1 and all the other random object index pages.

So, reducing the vpseqt to 25% or less will maintain the performance of the other objects in the pool, and reduce the overall I/O rate in this pool.

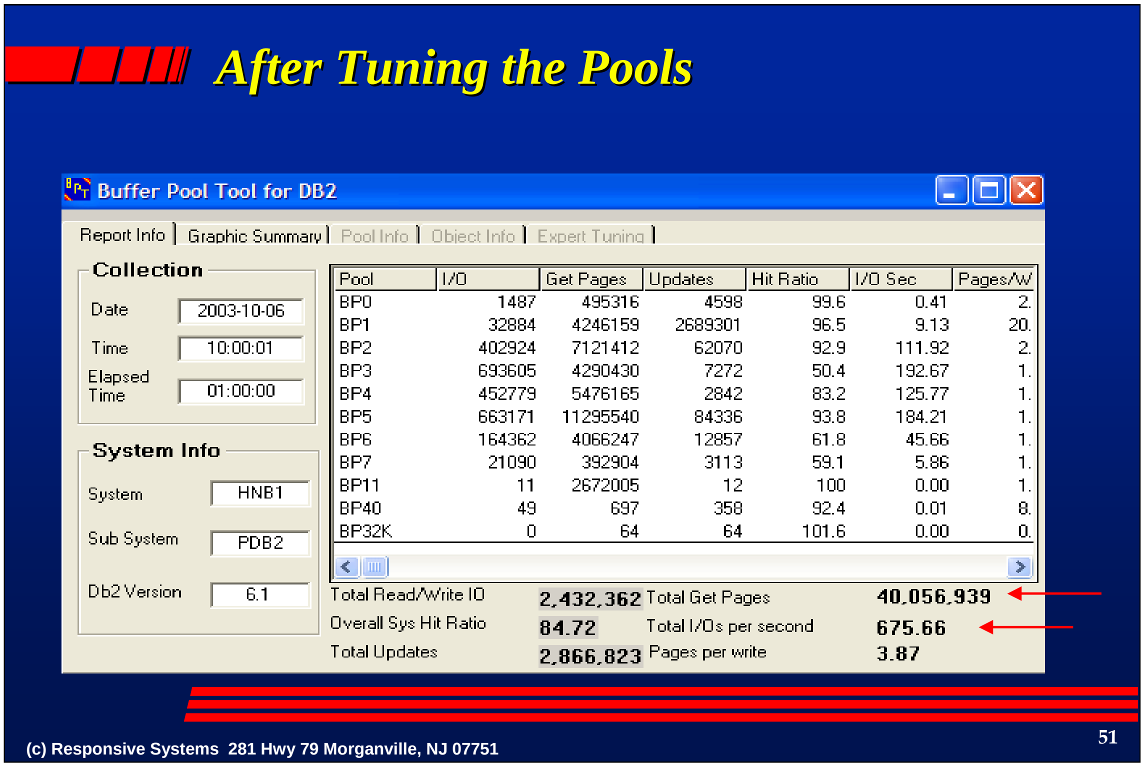


Now if we continue to look at the activity against DINV1, we see that the access is almost evenly split between random and sequential.

So – the access was mostly random, and had one, or possibly a couple of scans. However, it was the scans that hurt the performance of everything else in the pool.

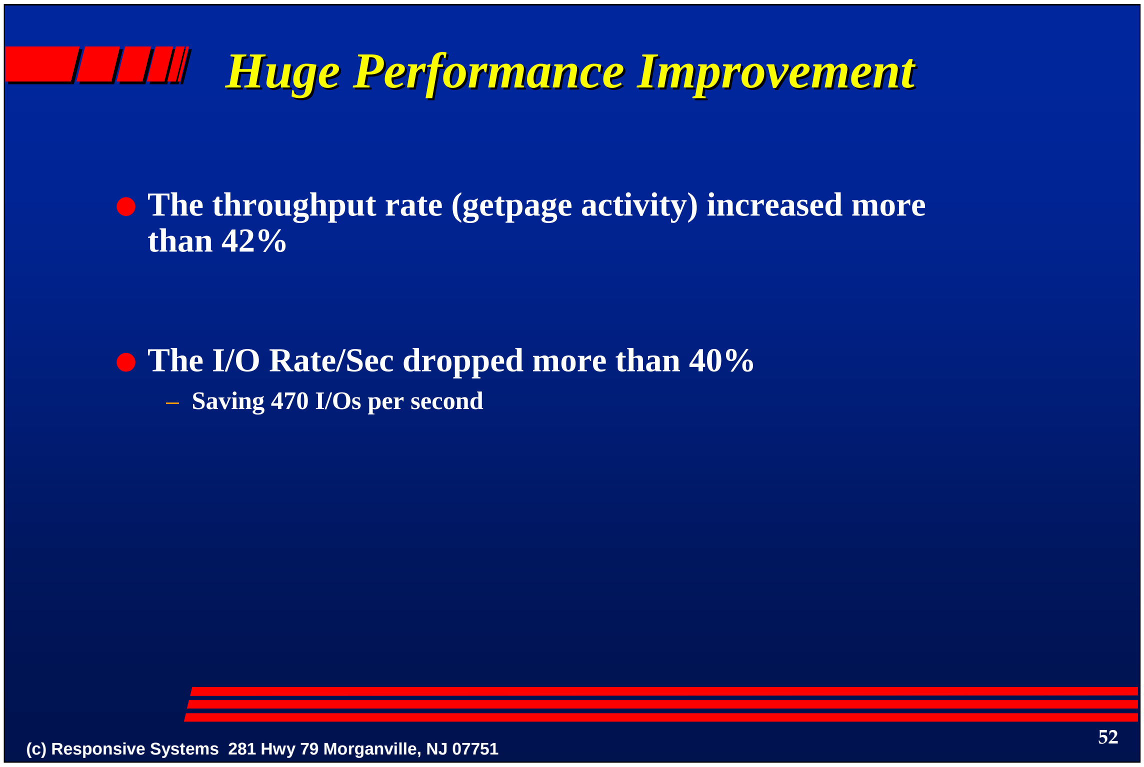


Here is a system with a much higher overall I/O rate.



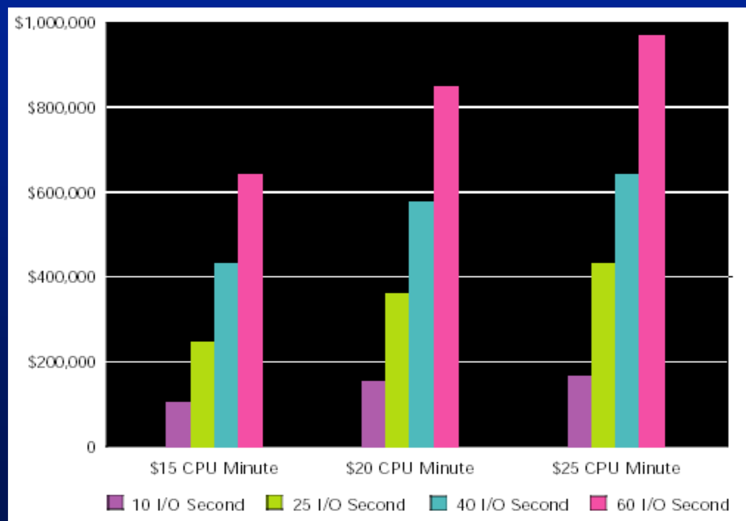
Here is the same system 3 months later at the same timeframe, the number of pools has been reduced, the getpage activity has increased about 42%, and the I/O rate has dropped by 470 per second.

The greatly reduced I/O allows more useful work to get through the system, reflected by the increased Getpage activity.



This is easily a six figure cost saving from I/O elimination.

Cost Savings from I/O Elimination



9 Minutes of Data, extreme I/O rate

Too many pools

Pool	I/O	Get Pages	Updates	Hit Ratio	RIO/Sec	WIO/Sec	Pages/Write
BP0	296	31154	914	99.2	0.37	0.17	4.06
BP1	2	368	92	100	0.00	0.00	9.00
BP2	242	3165253	2128023	100	0.44	0.00	0.00
BP3	34136	2567512	83108	95.6	59.14	3.38	13.31
BP4	39451	2283803	46286	97.8	70.58	1.68	8.29
BP5	307	3534	103	94.9	0.27	0.29	1.03
BP6	21470	326944	26341	90.5	32.35	6.98	3.20
BP7	21	50653	63	100	0.03	0.01	1.00
BP8	26	129057	22	100	0.05	0.00	1.00
BP11	236600	1045550	312	77.4	433.14	0.19	3.45
BP13	147	904	503	72.6	0.16	0.11	8.18
BP14	52490	1053955	72590	64.8	93.73	2.41	19.02
BP23	587005	4630481	379764	85.1	765.37	309.73	1.74
BP24	436740	1764411	349297	77.8	544.21	255.68	1.99
BP26	44350	496102	26334	85.8	73.12	8.11	3.26
BP33	88205	1832650	162935	65.2	150.07	11.48	7.00
BP34	19073	210266	676	-6.1	33.07	1.87	1.06
BP43	603915	4349025	91943	34.3	1,077.14	28.94	3.57
BP44	480966	6849343	95790	89.2	860.65	20.24	2.97

Total Read/Write IO	2,645,442	Total Get Pages	30,790,965
Overall Sys Hit Ratio	79.18	Total I/Os per second	4,845.13
Total Updates	3,465,096	Pages per write	2.22

15.75 Minutes of Data even higher I/O rate Different System

Collection

Date	2004-03-01
Time	10:05:30
Elapsed Time	00:15:48

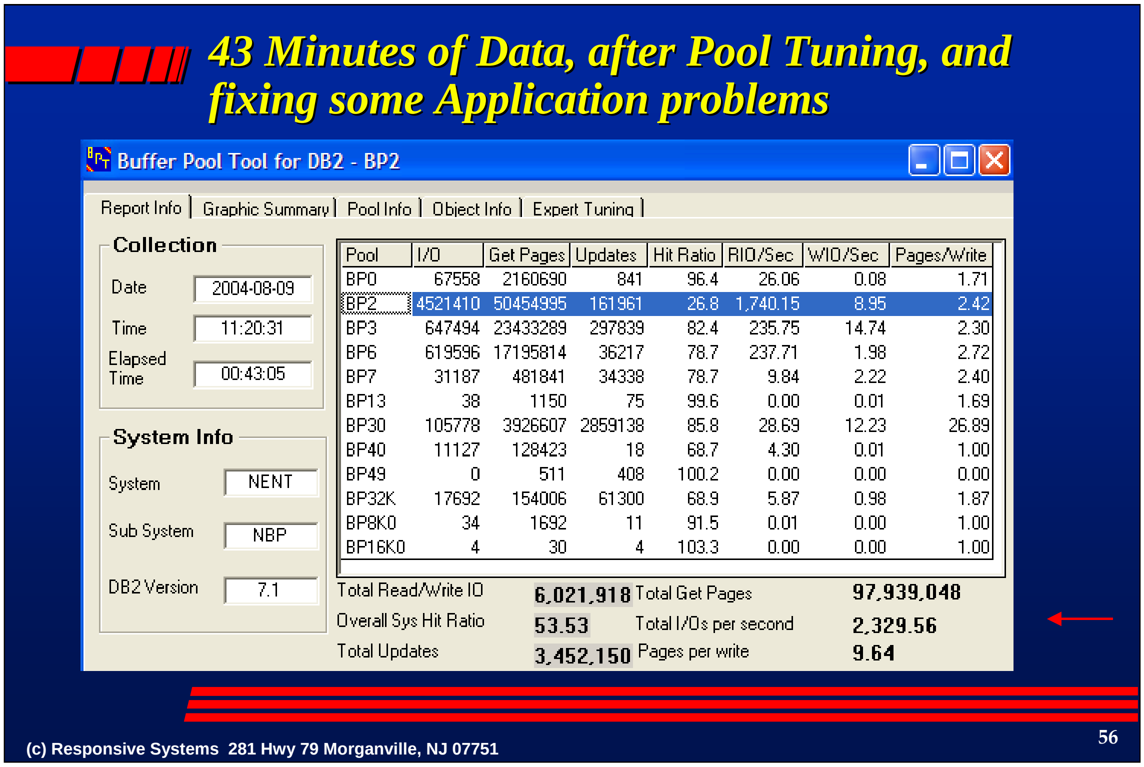
System Info

System	NENT
Sub System	NBP
DB2 Version	7.1

Pool	I/O	Get Pages	Updates	Hit Ratio	RIO/Sec	WIO/Sec	Pages/Write
BP0	86914	792976	60	77	91.65	0.03	1.88
BP2	2989001	20443958	76353	-18.6	3,122.22	30.74	1.77
BP3	1304487	16649309	112401	59.4	1,321.93	54.11	1.42
BP4	836	39610	9288	96.4	0.81	0.07	5.74
BP6	619697	1602799	7192	-63.1	651.08	2.61	1.60
BP7	41327	262998	23924	68.4	37.85	5.74	2.06
BP13	39	222	104	88.3	0.03	0.01	3.75
BP30	5701	1245951	213501	98	3.45	2.57	22.46
BP49	0	500	300	100.2	0.00	0.00	0.00
BP32K	10560	82445	19188	84.7	9.29	1.85	2.04

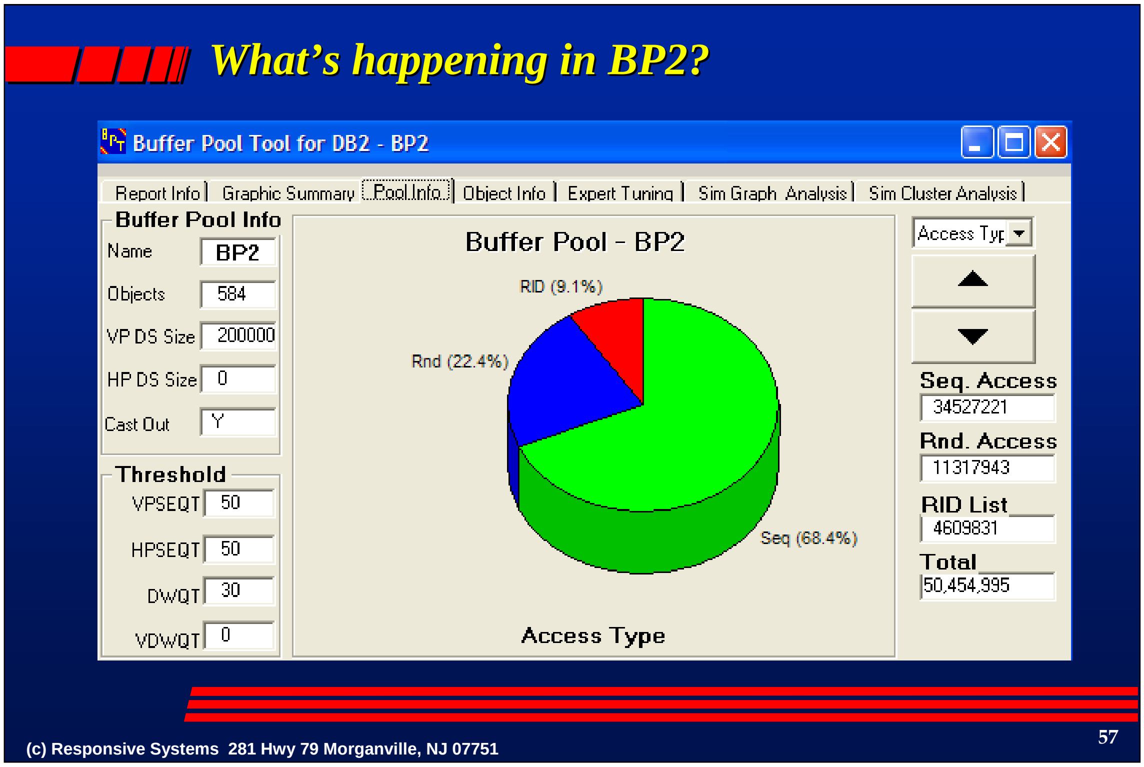
Total Read/Write IO	5,058,562	Total Get Pages	41,120,768
Overall Sys Hit Ratio	17.50	Total I/Os per second	5,336.04
Total Updates	462,311	Pages per write	2.14

A new high for an I/O Rate/Sec. This is a standard AM workload.

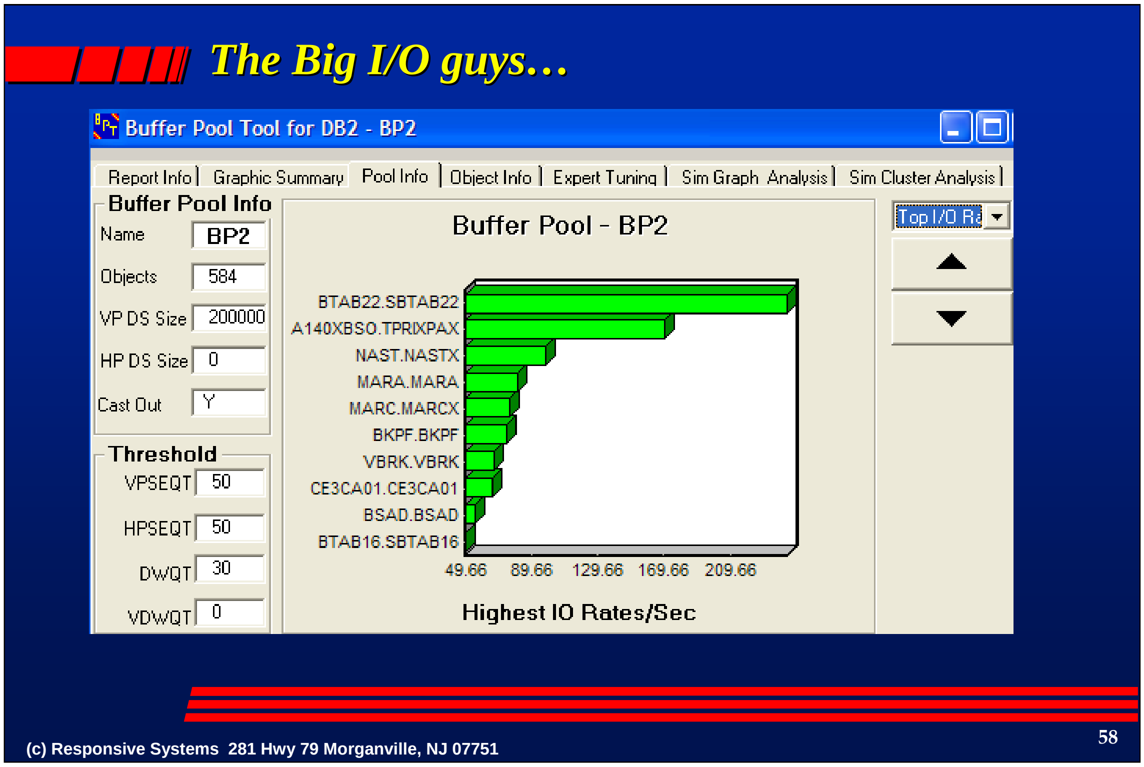


Pool	I/O	Get Pages	Updates	Hit Ratio	RIO/Sec	WIO/Sec	Pages/Write
BP0	67558	2160690	841	96.4	26.06	0.08	1.71
BP2	4521410	50454995	161961	26.8	1,740.15	8.95	2.42
BP3	647494	23433289	297839	82.4	235.75	14.74	2.30
BP6	619596	17195814	36217	78.7	237.71	1.98	2.72
BP7	31187	481841	34338	78.7	9.84	2.22	2.40
BP13	38	1150	75	99.6	0.00	0.01	1.69
BP30	105778	3926607	2859138	85.8	28.69	12.23	26.89
BP40	11127	128423	18	68.7	4.30	0.01	1.00
BP49	0	511	408	100.2	0.00	0.00	0.00
BP32K	17692	154006	61300	68.9	5.87	0.98	1.87
BP8K0	34	1692	11	91.5	0.01	0.00	1.00
BP16K0	4	30	4	103.3	0.00	0.00	1.00

After tuning the pools, and fixing some application performance problems.

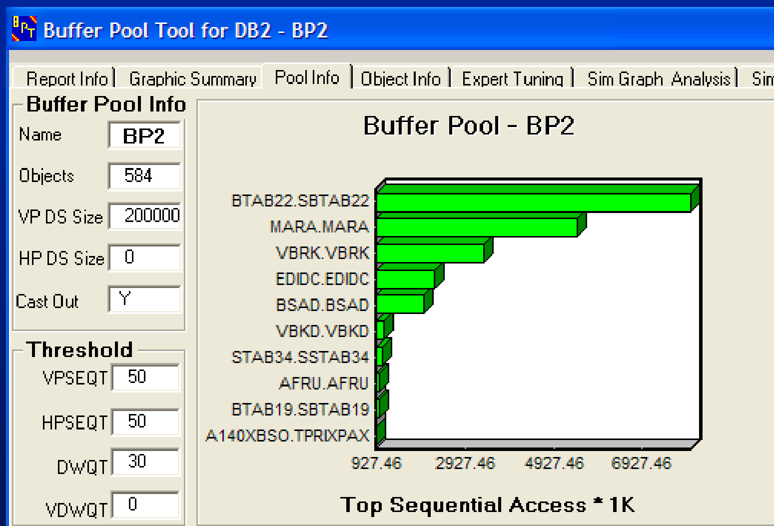


BP2 has the highest I/O rate, and the access is heavily sequential.
Always look at the big problem areas first.

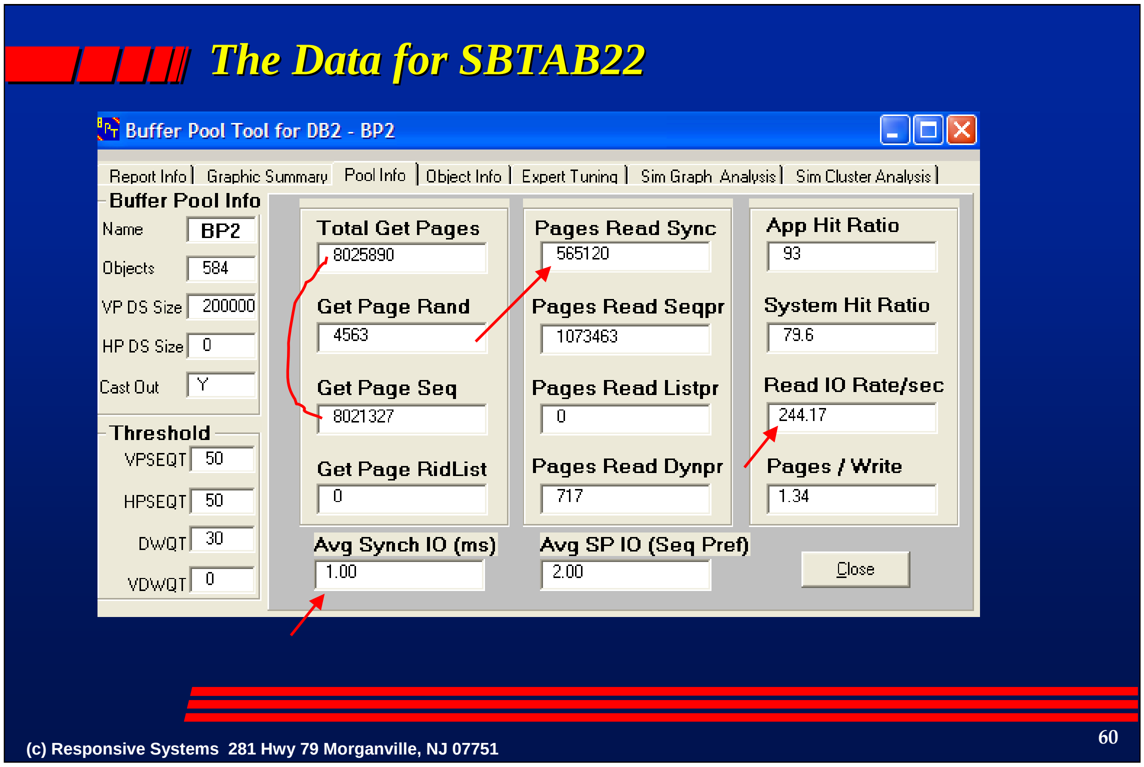

The Big I/O guys…
Buffer Pool Tool for DB2 - BP2
Report Info
Graphic Summary
Pool Info
Object Info
Expert Tuning
Sim Graph Analysis
Sim Cluster Analysis
Buffer Pool Info
Name BP2
Objects 584
VP DS Size 200000
HP DS Size 0
Cast Out Y
Threshold
VPSEQT 50
HPSEQT 50
DWQT 30
VDWQT 0
Buffer Pool - BP2
BTAB22.SBTAB22
A140XBSO.TPRIXPAX
NAST.NASTX
MARA.MARA
MARC.MARCX
BKPF.BKPF
VBRK.VBRK
CE3CA01.CE3CA01
BSAD.BSAD
BTAB16.SBTAB16
49.66
89.66
129.66
169.66
209.66
Highest IO Rates/Sec
(c) Responsive Systems 281 Hwy 79 Morganville, NJ 07751
58

The bad guys….

The Heavy Sequential Objects



The *really bad* guys…. So let's take a lower level look at SBTAB22



We have several interesting things here:

A: almost all the getpage requests are sequential

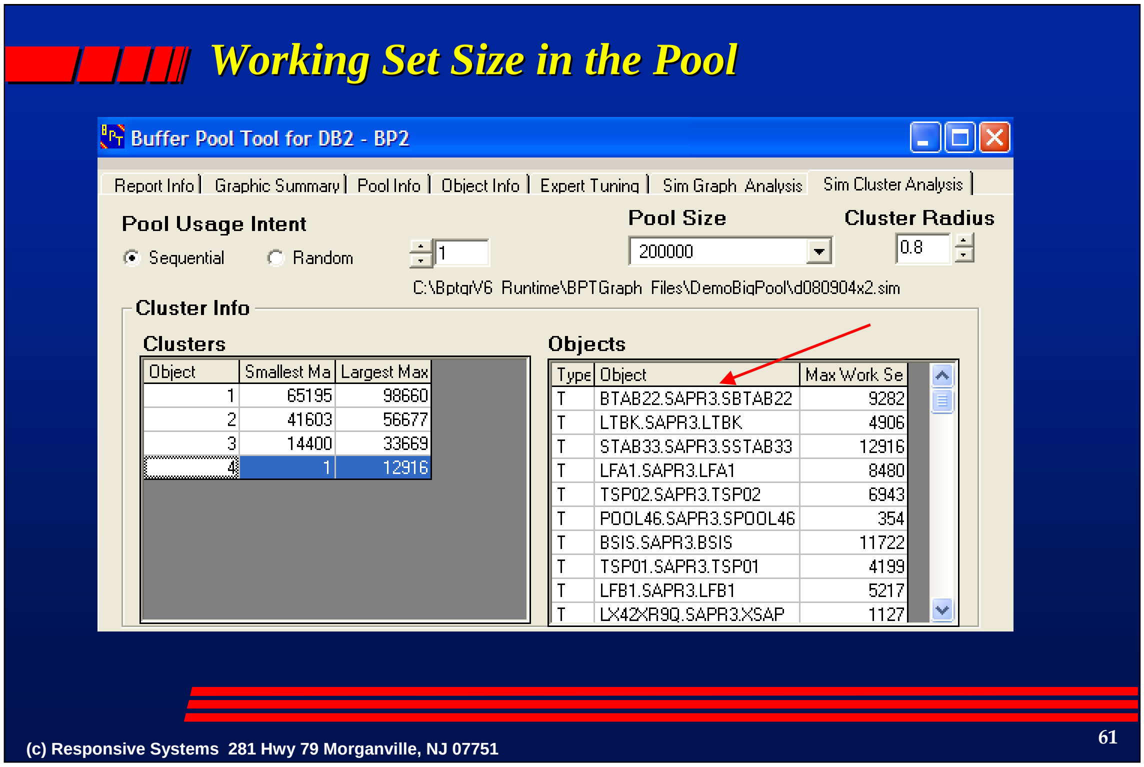
B: high synch I/O activity

C: high I/O rate

D: vpseqt of 50% can allow up to 100,000 pages for one sequential scan

Analysis: 1/8 of the requested pages are read into the pool with prefetch, but sequential data falls off the LRU queues before the application can read the data – and the pages are read back in using Synch I/O. The 1 Ms avg Synch I/O time shows that the requested pages are all found in the cache of the dasd control unit.

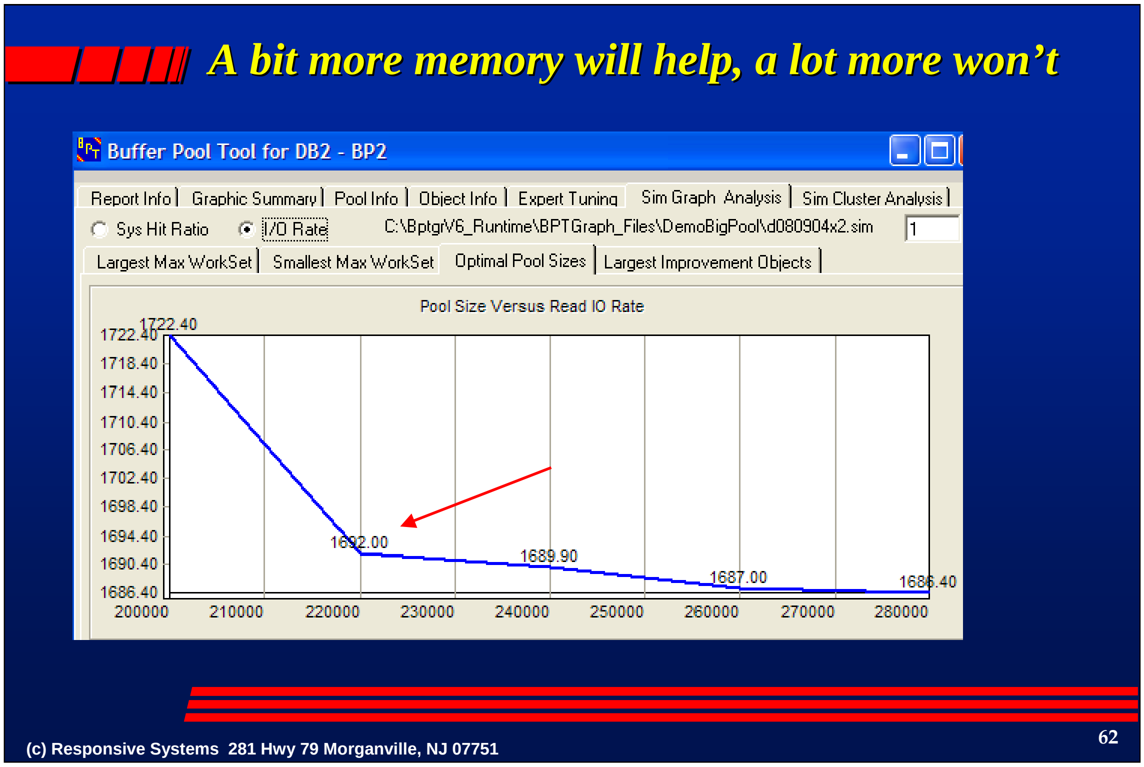
So – if it can live at the dasd cache level, maybe a larger pool will be able to keep it in memory?



Since the Wkset is not really big compared to other sequential objects, there are probably many scans/accesses of the object, and it uses less than 5% of the pool.

Analysis: It's probably accessed a lot by a batch job, that does not have a high enough priority to get the pages read in by prefetch, and they have to be re-read with Synch I/O a second time when the actual getpage is issued.

So – again, will a larger pool help?



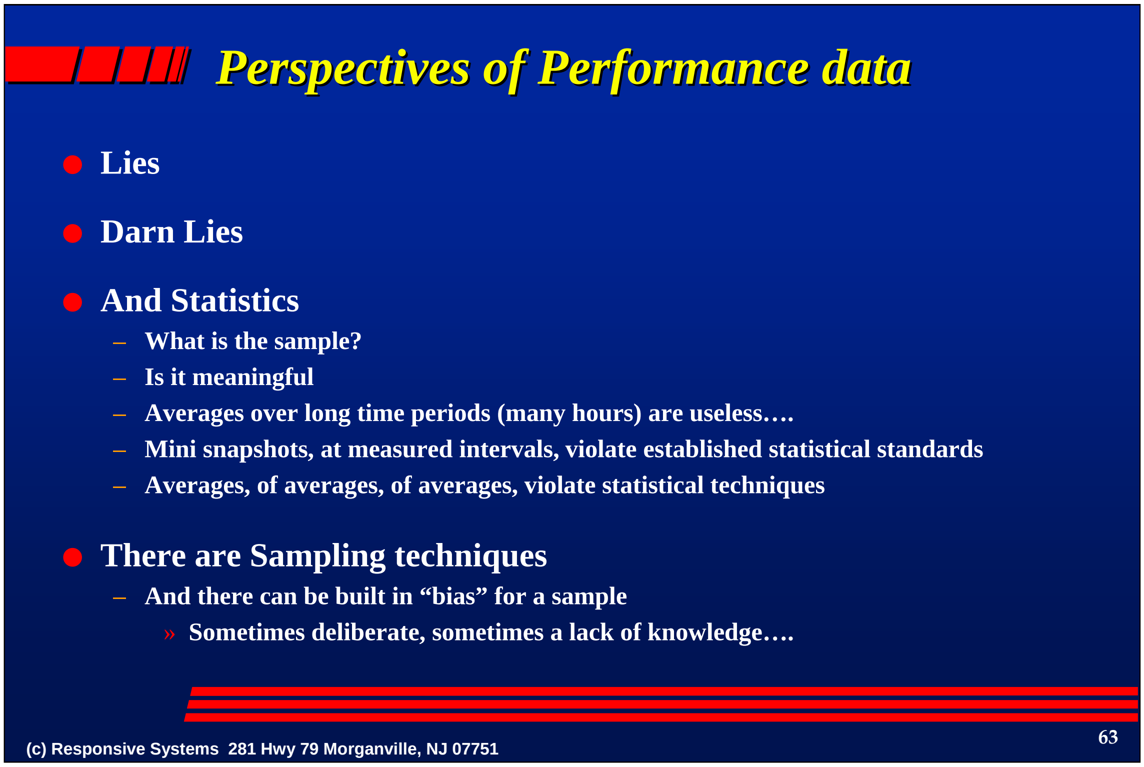
If the pool is increased by 20,000 buffers (80Meg) to 220,000 buffers, this will save 30 I/O sec. Increasing beyond this size is a complete waste of memory.

We started with a very large pool – in this case making it even larger (+10%) will help.

But – just throwing huge amounts of memory at pools does not always improve performance!

64Bit memory has some great possibilities, but *more memory resources do not necessarily mean better performance.*

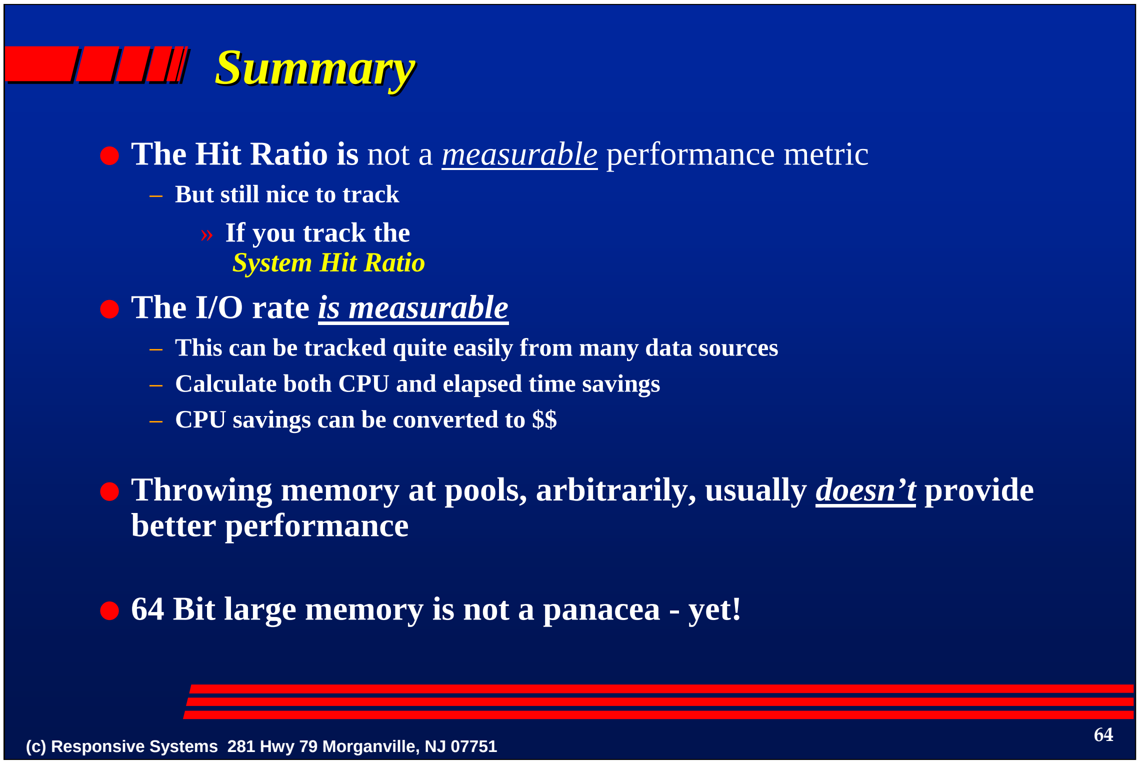
The proper grouping of objects, and effective use of memory can provide substantial performance improvements.



There are proper ways to collect data, that provide statistical validity, and these techniques are clearly documented by the National Bureau of Standards. Data collection techniques that violate accepted standards, rarely, if ever, produce reliable results.

When comparing sets of performance data, the data must be from the same timeframe, same duration, and hopefully a workload that is reasonably close to the first. Now, unless you have a well documented, canned, and repeatable benchmark process, there will always be some degree of variation in your workload. You always need to look at the level and type of activity that occurred within your data intervals, and make you own determination if they are close enough for reasonable comparisons.

If you take small snapshots of data across a day, they need to be compared individually – and grouping the data together into averages usually produces gross errors of both performance and perspectives. Looking at performance data across long periods like 12 or 24 hours is completely useless for tuning and analysis purposes – because it will mask all performance spikes and problems.



Most sites can optimize performance using 6 to 8 pools, some larger ones may need a dozen pools. A small number of very large pools, ***absolutely will not***, provide good performance. They key to good performance remains the proper grouping of objects into pools based upon access type, and working set size. Until we can provide TeraBytes of memory, pool tuning will remain an important performance issue.

Thank you for coming

Are there any Questions?

Joel Goldstein

Responsive Systems Company

Joel@responsivesystems.com