Evolution of Buffer Management in Relational Database Management Systems

Kaveh Razavi

November 1, 2010

Abstract

Database buffer management has gone through tremendous changes since the beginning of relational database management systems. This report is going to look through this evolution between 1975 and 2000. Most notably, traditional algorithms are discussed and compared with query aware memory allocation like hot set, QLSM and marginal gains. New bottlenecks as memory becomes cheaper and thus larger is identified and solutions on overcoming them are then discussed.

1 Introduction

Underlaying structure and functionality for data management is a key element in performance characteristics of a database management system (DBMS). Buffer manager is a name given to a DBMS subsystem which is in control of the lowest level of memory management. Basically it is responsible for bringing pages containing the data from the hard disk to memory by means of a "PIN" operation and write the pages back from memory to disk by means of an "UNPIN" operation. When to do these operations and which pages to do these operations on, are the main tasks of the buffer manager and they have been a topic of research for a while. The problem is thus simple: Given the fact that buffers are there to hide the latency of disks, buffer manager should decide which pages of the database it should keep into the limited memory it is provided by the operating system.

The first buffer managers used the basic page replacement algorithms which performed well for operating system memory manager. However, it turned out that by exploiting the predictability of database queries' memory requirement, buffer managers could use the available buffers in a much more sophisticated way. There is a road path that researchers took to make the

best out of this predictability. The emergence of highly parallel workloads also becomes apparent during this time and researchers studied the behavior of the proposed buffer management algorithms on these workloads as well. The next two sections is going to discuss two of these new algorithms. In the next section, after looking at traditional algorithms for buffer management, I am going to overview "hot set" which is the first algorithm to exploit predictability of database memory references and then I am going to discuss "DBMIN" an improvement to "hot set" algorithm. In section 3, an algorithm based on "marginal gains" which outperforms previous algorithms is then reviewed.

Around a decade later, with increased amount of system memory, the problem shifted one layer closer to CPU, where the data actually gets processed. System researcher realized in most of the cases the main memory is no longer the bottleneck. CPU caches are very fast on-chip memory which act as a buffer to hide main memory's latency. To be fast and to be able to run on a fast clock to have low latency with CPU, cache memory is usually small and from a few kilo-bytes up to a few mega-bytes. Cache memory itself is usually divided into multiple levels called L1, L2 and sometimes L3. The higher the number, more memory can be buffered and thus slower it gets. Section 4, discusses this problem and possible solutions in more detail and the last section concludes this report.

2 Query aware buffer management

This section first discusses the traditional methods for buffer management. Then the hot set model is described and after that comes query locality set model (QLSM) which DBMIN algorithm is based on.

2.1 Traditional methods for buffer management

Traditionally, database systems all used basic LRU or some forms of it to minimize the number of page faults in the system. It has been shown [1] that LRU among other basic page replacement algorithms performs best under typical workloads and apart from that, it is simple to implement which makes it a good candidate for high performance subsystems. However as it is discussed, many situations occur during processing a query which LRU does not perform well. For example, when executing multiple queries together, effects of one query could have bad effects on the execution of the others. Sequential scans are an example of these queries. Another problem which happens frequently is during nested loop joins when the buffer is not

big enough to hold the pages of inner relation in memory. In this situation, LRU performs the worst. Other problems arise due to the fact that LRU has no mechanism to prevent thrashing under heavy workload. These problems are referred to as internal and external thrashing [5]. To avoid such problems two new algorithms were proposed:

Domain Separation Reiter [2] suggested that it is important to differentiate between the "type" of a page. For example, consider a query which randomly accesses data pages through a B-Tree index. In this example, the root of the B-Tree is always going to be access in each data reference. The pages in the second level of the B-Tree is not as important as the root page but still is going to be accessed with a high probability. And generally an index page is much more important than a data page. Based on these observations, he suggested that each type of pages should have its own domain and LRU should be taken in each domain and domains can use frames from other domains in case they run out of memory. As a simple scheme, he used one domain for each level of the B-Tree index and one domain for the leaf level (data pages) and it showed $8 - 10\%$ throughput improvement in comparison to basic LRU.

The main limitation of Domain Separation is that the concept of domain is static. An importance of a page could vary between different queries. The very unimportant data page in a B-Tree reference could become very important if used as an outer relation of a nested loop join. It also does not take into account the relative importance of one domain to another. And there are a couple of other problems which I do not mention for the sake of briefness. To address these problems, a couple of buffer management algorithms based on or similar to Domain Separation were proposed.

"New" algorithm Another step forward in the improvement of database buffer management was these two observations by Kaplan [4] in his Master thesis: importance of page is defined by the relation it belongs to and not the page itself and each relation should have a "working set" [3]. Based on these observations, Kaplan designed "new" algorithm, which worked by defining a working set for each relation. These sets are put in a priority queue with the free list on top. Whenever a page fault occurs, a search is started from the priority list to find a suitable frame. If pages of a relation is less likely to be accessed it is put on top of the priority queue. Each relation can have one "fixed" page and the working set is handled with MRU algorithm. Although "new" algorithm brought in a novel idea in to

the area of buffer management, it failed to perform better in comparison to LRU and the reason for this was mainly because the use of MRU within each working set is not very well justified. However, the novelty of the idea was reused in the hot set model.

2.2 The hot set model

The host set model [5], is one of the first models which tried to take database memory reference predictability into account. That made it able to cope with most of the deficiencies of traditional methods. It tries to predict the size of pages that a query spends most of its calculation based on **the size of the relations** that the query is working on and **the type of database operation** it needs. This size σ is the size of the query's hot set. For example, if it is required to do a nested loop join to answer a query, then hot set size of the query is $1 + sizeof(R2)$ where $R2$ is the inner relation. Based on different database operations, different heuristics are used to predict the hot set sizes of given queries. These heuristics are then used with a buffer management algorithm. As one can see, the idea of allocating memory to relations is followed from the "new" algorithm.

The implemented algorithm based on the hot set model works as follows:

1. Initialize: put all the available frames in a free LRU list.
2. New query comes in: allocate a new empty list with two values $numref = \sigma$ and $numall = 0$. Get as σ many page as you can from free list and set $numall$ to that number. Note that if not enough frames are available then $numall$ will be less than σ.
3. Query requests a page:
   (a) Page is in memory: if in local LRU list, update the usage information. Else, do nothing.
   (b) Page is not in memory: page fault. If we have free frames, bring the page in local LRU list. If not, if there is an unfixed page, based on LRU principle, flush it to disk and use it and if all pages are fixed try to get one from free list based on LRU and increase $numall$. If free list is empty get one unfixed page (again based on LRU) from the query with largest $numref - numall$ since they are already bad (i.e. do not have enough pages for their hot set).

4. Fix the page: increase its referee count. Do computation. Fixing means the page is now in the hot set.

5. Unfix the page: decrease reference count.

6. Release page: query does not this page anymore. We take lazy step and do nothing here.

7. Query leaves: Find queries with $numref - numall > 0$ and try to satisfy their needs. If there is none, give pages back to the free list.

The hot set model clearly overcomes or avoids most of the problems of traditional algorithms for buffer management like internal and external thrashing by predicting the memory needs of a query. Next subsection, after introducing DBMIN, shows performance comparison results of hotset and DBMIN with traditional algorithms.

2.3 Query locality set model

Query locality set model [6] (QLSM) took the idea of hot set model one step further in the sense that it tries to separate the modeling of reference behavior from any particular buffer management algorithm like LRU or MRU. Roughly speaking, QLSM uses more information about the type of operation for page replacement instead of just simply using LRU. QLSM does this by decomposing the memory reference pattern of each operation into the composition of a number of simple reference patterns. To do this, QLSM uses the notion of *file* which is allocated a certain number of memory frames. Each query, based on the relation which it works on during its execution, opens and closes these files.

There are two questions that need to be answered with this model: How many frames should be allocated to each file and what local replacement policy should be used within each file? The answer to the first question is somewhat what hot set model tried to come up with. However, the second question is the main idea behind the QLSM because now the replacement policy for each relation is not something general like LRU and instead based on the reference pattern of the type operation the query opened the file for. Examples of these patterns are sequential references, random references and hierarchical references. These patterns are then further divided depending on the type of operation.

DBMIN is a buffer management algorithm based on QLSM. Assume there are N frames in the system. l_{ij} is the maximum number of frames

that can be allocated to jth file of query i and r_{ij} is the actual number of frames that is currently allocated to jth file of query i. At first all the N frames are in a global free list DBMIN avoids thrashing by suspending a query if it requests a file of size l and $(l + \Sigma\Sigma l_{ij})$ becomes bigger than N. All the pages are also accessible through two global free and busy lists.

When a file is opened, its l and r values are passed to the buffer manager. When a page is requested by a query a search is made through global lists. Three possible scenarios might happen:

1. Page is in global and local list: its information required by the local replacement policy is updated

2. Page is in global but not in local list: if it has an owner simply add it to the locality list. If not, add it to the locality set and increase r now if $r => l$, choose one page according to local replacement policy and add it to the free list.

3. Page is not in memory: initiate a disk read to a buffer in free list and then proceed like 2.

It is obvious that these local files are just there to maintain the working set of the relation which query is working on and the actually swapping happens in the global lists. The last thing to discuss for the implementation of DBMIN is how to determine l and the replacement policy of each file. As mentioned while describing QLSM, these are identified by the study of memory reference pattern in different database operation required to execute a query. I will briefly explain two examples of these patterns and others could be found in [6].

Straight Sequential (SS) References Happens for example in the outer relation of a nested loop join where every data page is accessed only once. Hence only one memory page is required and is overwritten when a the next page is requested.

Looping Sequential (LS) References Happens for example in the inner relation of a nested loop join where the whole data pages are scanned repeatedly. Here obviously MRU performs best and it is good to give the file as many page as possible till the whole file could fit in memory. Here l is the size of file then.

Thus, when a nested loop join is to be performed, the buffer manager opens two files; One with SS policy for the outer relation and another with

LS policy for the inner relation. Chou and DeWitt also introduced a new hybrid simulation for assessing the performance of DBMIN over hot set and other traditional algorithms which I am not going into for the sake of briefness.

2.4 Comparison of buffer management algorithms

There are many figures in [6] comparing different queries and environments. The authors decided on three query mixes which is described below.

Query number	Query operations*	Selectivity factor (%)	Access path of selection	Join method	Access path of join
I	Select (A)	1	Clustered index	—	—
II	Select (B)	1	Nonclustered index	—	—
III	Select (A) join B	2	Clustered index	Index join	Clustered index on B
IV	Select (A') join B	10	Sequential scan	Index join	Nonclustered index on B
V	Select (A) join B'	3	Clustered index	Nested loops	Sequential scan over B'
VI	Select (A) join A'	4	Clustered index	Hash join	Hash on result of select (A)

* A, B: 10K tuples; A': 1K tuples; B': 300 tuples; 182 bytes per tuple.

Query mix	Type I (%)	Type II (%)	Type III (%)	Type IV (%)	Type V (%)	Type VI (%)
M1	16.67	16.67	16.67	16.67	16.66	16.66
M2	25.00	25.00	12.50	12.50	12.50	12.50
M3	37.50	37.50	6.25	6.25	6.25	6.25

I am only putting the figures regarding query mix M3 here since it seems to be more realistic in comparison to real world workloads. Figure on the left shows the comparison without data sharing and the figure on the right shows the results using full data sharing between queries.

Clearly, DBMIN is working better compared to traditional buffer managements and also the one using the hot set model.

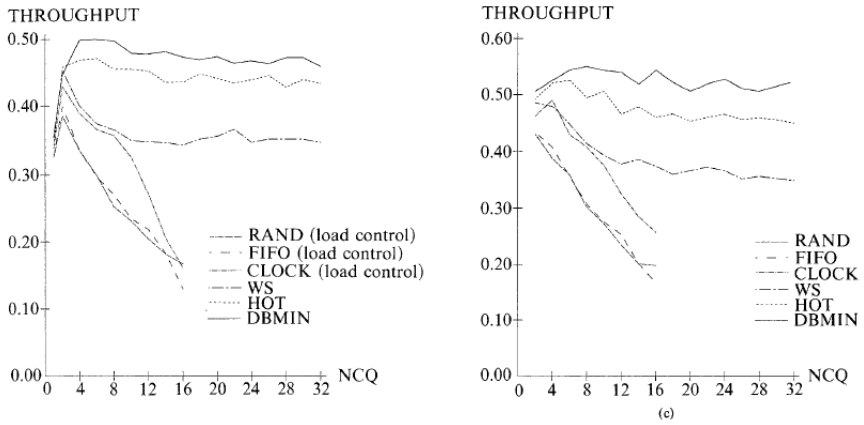


Figure 1: Performance comparison of DBMIN with other algorithm in workload M3

3 A more flexible solution using marginal gains

As NG et la [7] suggested, there are a couple of problems with QLSM which causes low buffer utilization and throughput. For example, if a file for the inner relation of nested loop join requires 10 buffers and there are only 9 free buffers available in the system, DBMIN does not start the operation. Or even if there are 5 free buffers in the system, DBMIN always allocates one buffer for a random reference (for example through a hash join). They argue that traditional buffer management algorithms only take into account the availability of buffer and newer algorithms like hot set or DBMIN care only about memory reference pattern and only reason about available buffers in a static manner.

Based on this observation, a flexible buffer management based on marginal gains was introduced. This new model reasons about available buffers in the system as well as memory reference patterns of the operation. Thus, it is actually very similar to DBMIN except the fact that current available buffers are also taken into account when allocating buffers to files. The goal of the new model is to maximize page hits when allocating buffers to database operations.

To explain the new model a formalization of database memory access pattern is given and then based on that the new model is described. I will briefly go through the process of modeling random references using Markov chains seems it seems to be the most interesting case and then I will explain

the proposed new algorithm based on this new model and after that we see how it performs compared to DBMIN.

3.1 Modeling of random references using marginal gains

Reference Ref of length k to a relation is a sequence of page references $< P_1, P_2, ..., P_k >$ of that relations that needs to be read in order.

Random Ref $R_{k,N}$ of length k of a relation of size N pages is a reference of size k, where the page references are uniformly distributed over all the N pages of the relation and each reference is independent of the other one.

$\mathbf{Ef(R_{k,N}, s)}$ is the expected number of faults using s buffers in a random reference $R_{k,N}$.

$\mathbf{mg(R_{k,N}, s)}$ is the marginal gain of using s buffers instead of $s-1$ for a random reference $R_{k,N}$ and equals $Ef(R_{k,N}, s-1) - Ef(R_{k,N}, s)$.

Using this definition, it is clear that marginal gain is more flexible than that of locality set sizes in DBMIN because now the algorithm can decide for available buffers although they may not be enough for a certain operation (e.g. a looping reference). What remains to be discussed is how to calculate Ef or expected number of faults using Markov chains.

Assume $P(f, k, s, N)$ is the probability that there there f number of faults in k access in the relation of size N using s buffers. Then:

$$Ef(R_{k,N}, s) = \sum_{f=1}^{k} f.P(f, k, s, N)$$

To calculate the probability P two cases needs to be considered. Either kth access has produced a fault or it has not. Based on this and the fact the number of faults could be bigger or smaller than the number of allocated buffer, section 3.2 of [7] devises the following formula in the mentioned cases using Markov chains:

$$P(f, k, s, N) = \begin{cases} f/N * P(f, k-1, s, N) + (N-f+1) * P(f-1, k-1, s, N) & f \leq s \\ f/N * P(f, k-1, s, N) + (N-f+1) * P(f-1, k-1, s, N) & f > s \end{cases}$$

And the base case is $P(0, 0, s, N) = 1$. However, there does not exist a simple closed formula for Ef but below is a very close approximation to the answer:

$$Ef(R_{k,N}, s) = \begin{cases} N * [1 - (1 - 1/N)^k] & k < k_0 \\ s + (k - k_0) * (1 - s/N) & k \geq k_0 \end{cases}$$

$$k_0 = ln(q - s/N)/ln(1 - 1/N)$$

More detail about the formula could be found in the mentioned section. The important thing is that the calculated formula could be used in mg formula to derive the number of buffers that needs to be allocated to this random reference to maximize the number of page hits based on the available buffers in the system.

3.2 MG-x-y a new algorithm based on marginal gains

The best buffer allocation algorithm best on marginal gain would be whenever there are N free buffers in the systems, choosing the queries in the waiting list which maximizes the total marginal gain. This computation is expensive however. MG-x-y is a class of algorithms based on marginal gains with low computational costs. The only difference between DBMIN and MG-x-y is that MG-x-y takes the available buffers into account when trying to allocate buffers to a query. After this number is determined MG-x-y works exactly like DBMIN. Whenever buffers are released by a new completed operation, MG-x-y invokes the following algorithm for the queries in the waiting FIFO queue in a loop until there are no more queries or free buffers left. x and y are the inputs of MG-x-y and R is the reference pattern at the head of the queue:

- R is a looping reference: (not discussed, refer to [7]) (argument x is used here)
- R is a random reference $R_{k,N}$: as long as the mg is positive allocate as many buffers as possible till the number of allocation is $\leq$ min(Available buffers, y)
- R is a sequential reference: allocate 1 buffer

3.3 Comparison of MG-XY with DBMIN

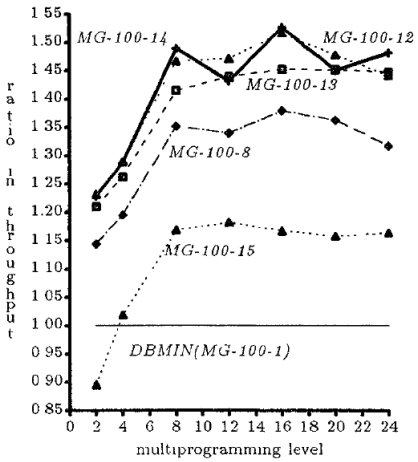
The figures below is the environment setup for the simulation. The same simulation setup as DBMIN is used.

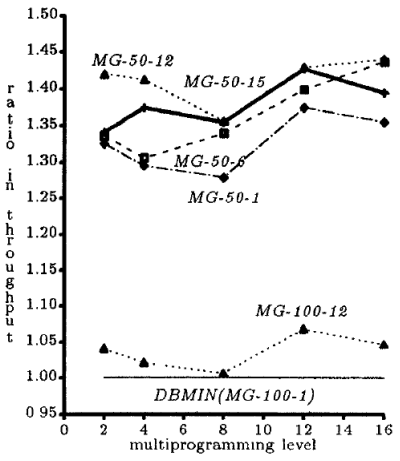
query type	query operators	selectivity	access path of selection	join method	access path of join	reference type
I	select(A)	10 %	clustered index	–	–	$\mathcal{S}_{50,500}$
II	select(B)	10 %	non-clustered index	–	–	$\mathcal{R}_{30,15}$
III	select(A) ⋈ B	1 %	sequential scan	index join	non-clustered index on B	$\mathcal{R}_{100,15}$
IV	select(A) ⋈ B	4 %	clustered index	nested loop	sequential scan on B	$\mathcal{L}_{300,15}$

relation A	10,000 tuples
relation B	300 tuples
tuple size	182 bytes
page size	4K

	I	II	III	V
mix 1	10%	45%	45%	–
mix 2	10%	10%	10%	70%

The purpose of mix 1 is to evaluate the effect of allocation of buffers in random references. Mix 2 is out of the scope of this report. Figures below show the performance comparison of various MG-x-y and DBMIN. Note that DBMIN is actually a specific case of MG-x-y. The left figure shows the comparison of mix 1 and the results with full data sharing (queries have read access to the pages of other queries) in mix 2 is on the right for the sake of completeness.





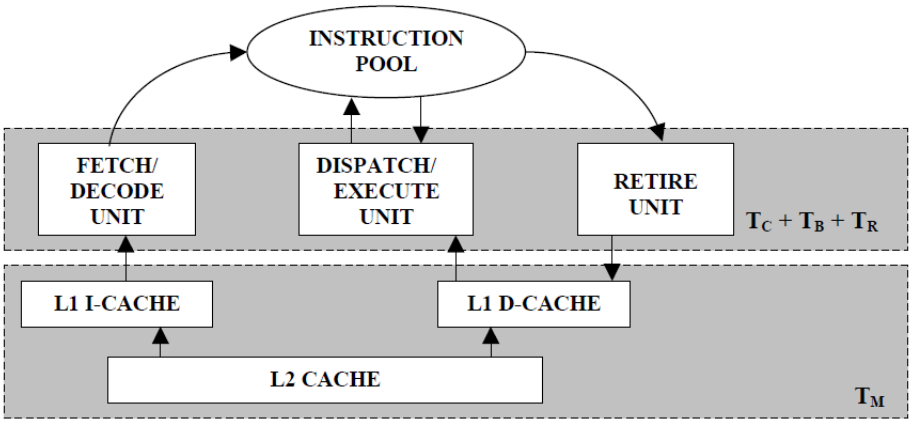
Multiprogramming here means the number of concurrent queries running in the system. Clearly, MG-x-y is superior in terms of throughput compared to DBMIN.

4 Buffer management on modern hardware

Later in 1999, with the emergence of new processors with larger cache memory, Ailamaki et la [8] discovered that even if the database in completely in memory, 50% `of the time` the CPU is waiting on cache stalls. In this section, I am going to go through why this is the case and how database researchers could overcome these problems.

4.1 Instruction execution on modern processors

Figure below shows the simplified block diagram of a modern processor operation. Whenever the processor wants to execute an instruction, the instruction decode unit reads the first byte from the I-cache and then based on that, it reads the other bytes. If an L1 I-cache miss occurs here the processor needs to wait for L2 and if L2 cache miss occurs then it needs to wait for main memory. The instruction is then put in the instruction pool to be scheduled for execution. When executing, it could be that the processor needs data which is not in the registers. Then it tries to read them from L1 D-cache and if it misses it goes through (data and instruction unified) L2 cache and then if another miss occurs here, the processor needs to go to main memory. The retire unit then checks among executed instruction to see if they can be committed.



During the first two phase of execution (fetch and execution), it might be that a miss occurs and the processor needs to stall the pipeline. Processor tries to do some useful work while waiting for data like: **Non-blocking caches, out-of-order execution and Speculative execution**

with branch prediction. The details of these methods could be found in the section 3.1 of [8].

Even with these effective methods, it could still be that the processor needs to stall for memory to become available while executing a query. And thus the execution time T_Q for a query could be formulated as follow:

$$T_Q = T_C + T_M + T_B + T_R - T_{OVL}$$

Where T_C is the useful computation time, T_M is time waiting for memory, T_B is branch mis-prediction penalty, T_R is resource stalls (enough resources like ALU is not available) and since these stalls could overlap T_{OVL} is introduced in the equation.

T_C			**computation time**
T_M			**stall time related to memory hierarchy**
	T_{L1D}		stall time due to L1 D-cache misses (with hit in L2)
	T_{L1I}		stall time due to L1 I-cache misses (with hit in L2)
	T_{L2}	T_{L2D}	stall time due to L2 data misses
		T_{L2I}	stall time due to L2 instruction misses
	T_{DTLB}		stall time due to DTLB misses
	T_{ITLB}		stall time due to ITLB misses
T_B			**branch misprediction penalty**
T_R			**resource stall time**
	T_{FU}		stall time due to functional unit unavailability
	T_{DEP}		stall time due to dependencies among instructions
	T_{MISC}		stall time due to platform-specific characteristics

The breakdown of these waiting times is in the table above.

4.2 Measurement of different stall times

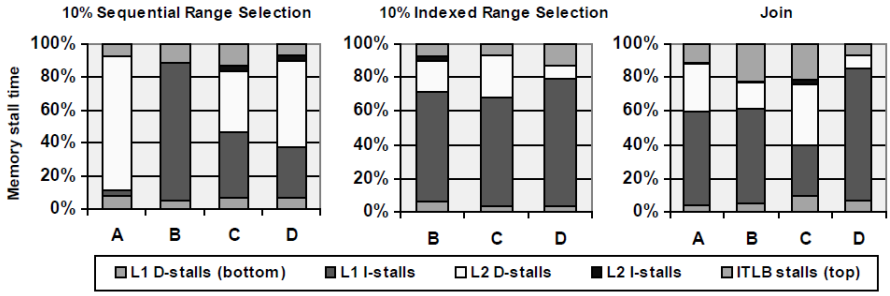
Section 4 and 5 of [8] goes deeply into details of setting up the environment and measuring the results which I am not going into for the sake of briefness. A real world representative database workload is consisting of sequential range selection, indexed range selection and a sequential join was devised to run on a Pentium II Xeon with the following specification:

Characteristic	L1 (split)	L2
Cache size	16KB Data 16KB Instruction	512KB
Cache line size	32 bytes	32 bytes
Associativity	4-way	4-way
Miss Penalty	4 cycles (w/ L2 hit)	Main memory
Non-blocking	Yes	Yes
Misses outstanding	4	4
Write Policy	L1-D: Write-back L1-I: Read-only	Write-back

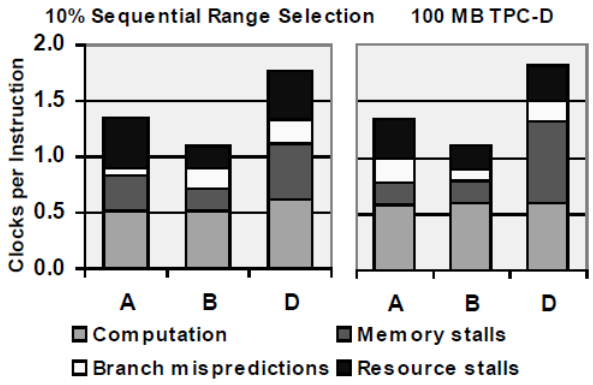
The table below shows the method of measuring each of the stall time components:

Stall time component		Description	Measurement method
T_C		computation time	Estimated minimum based on µops retired
T_M	T_{L1D}	L1 D-cache stalls	#misses * 4 cycles
	T_{L1I}	L1 I-cache stalls	actual stall time
	T_{L2} T_{L2D}	L2 data stalls	#misses * measured memory latency
	T_{L2I}	L2 instruction stalls	#misses * measured memory latency
	T_{DTLB}	DTLB stalls	Not measured
	T_{ITLB}	ITLB stalls	#misses * 32 cycles
T_B		branch misprediction penalty	# branch mispredictions retired * 17 cycles
T_R	T_{FU}	functional unit stalls	actual stall time
	T_{DEP}	dependency stalls	actual stall time
	T_{ILD}	Instruction-length decoder stalls	actual stall time
T_{OVL}		overlap time	Not measured

The workload was executed on the Pentium II Xeon using 4 different commercial databases and the main results for different stall times are shown below:

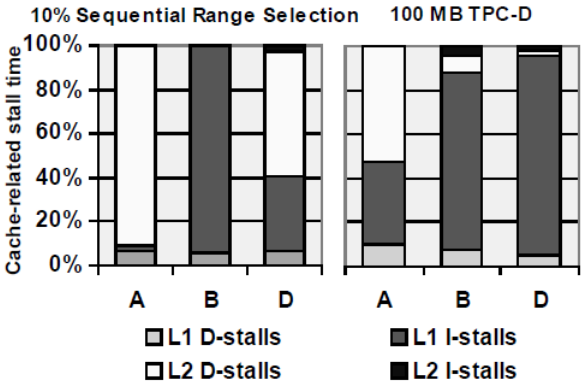


The ITLB stalls are due to virtual to physical memory translation in the instruction pages of the NT operating system. Another experiment including the sequential range scan and TPC-D workload calculated based on clock per instruction comes below:



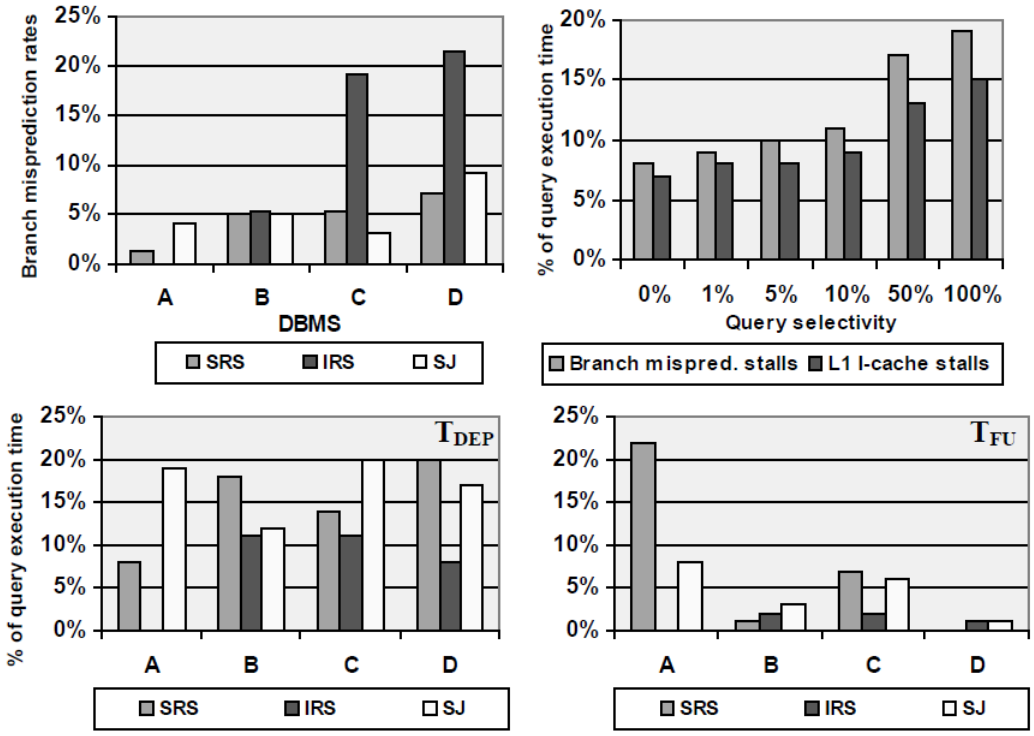
4.2.1 Breakdown of different stall times

Below is the cache related stall times while executing 10% sequential range scan and TPC-D workload:



And below are the stall times for branch mis-prediction and resource stalls (T_{DEP}) like low level dependency between instruction (i.e. one needs to be completed for the other to start or continue) which prevents the usage of pipeline and also functional unit (T_{FU}) stalls caused due to lack of resources to answer burst of CPU instructions:

SRS: Sequential Selection, IRS: Indexed Selection, SJ: Sequential Join



4.2.2 Places with prominent optimization

As one could see from the main results presented in section 4.2, most of the waiting time of the processor are due to memory stall times. Branch mis-prediction penalty and also stalls due to resource contention are out of the hand of database developers and needs subtle design changes in CPUs.

However, database developers could benefit a lot by reducing the memory stalls and as the graph with cache related stall suggests mainly by reducing L1 I-cache misses and L2 D-cache misses. L1 I-cache misses are mainly due to volcano model of processing and L2 D-cache misses are mostly due to row wise data storage in most of the queries. Thus, new design in relational databases are required to address these issues. MonetDB X100 [9] is one example of these new designs.

5 Conclusion

This report studied the evolutions of buffer managements in relational database systems roughly between 1975 and 2000 up until the realization of a new bottleneck in buffer management. The road path in database memory management started to part ways by that of the operating system by introducing domain separation. The "new" algorithm sparked the way for accounting memory to database relations instead of generally allocation memory. Based on this the hot set model was introduced which worked by allocating memory based on the memory requirement of each operation. QLSM pushed the idea even further by separating the memory reference pattern and replacement policy and finally a model based on marginal gains pushed the buffer allocation to its edges.

More than 10 years ago researchers realized that the memory -becoming cheaper- is no more a bottle neck for database performance. What is not studied in this report is the past 10 years of research which mostly focuses on data processing on modern hardware. Realizing the i-cache misses due to volcano model of processing and also high d-cache misses due to row-wise storage are now the main bottlenecks in database performance, researchers started implementing databases based completely on new ideas [9].

What I could not find, is research on adapting DBMS memory management to new NUMA processors. New non cache coherent processors are also going to be a challenge for database researchers. This might remain a topic of research for a while since OS researchers are still trying to come up with answers regarding these systems and these answers would greatly influence database performance based on its memory behavior.

References

[1] T. Lang, C. Wood, and I. B. Fernandez: Database buffer paging in virtual storage systems, ACM Trans. Database Systems, 1977.

[2] A. Reiter: A study of buffer management policies for data management systems, Technical Summary Report 1619, Mathematics Research Center, University of Wisconsin, Madison, 1976.

[3] P. J. Denning: The working set model for program behavior, Comm. ACM, 11 (1968), 323-333.

[4] J. A. Kaplan: Buffer management policies in a database environment, Master Report, University of California, Berkeley, 1980.

[5] G. M. Sacco and M. Schkolnick: A mechanism for managing the buffer pool in a relational database system using the hot set model, VLDB 1982: 257-262.

[6] Hong-Tai Chou, David J. DeWitt: An Evaluation of Buffer Management Strategies for Relational Database Systems, VLDB 1985: 127-141.

[7] 16. Raymond T. Ng, Christos Faloutsos, Timos K. Sellis: Flexible Buffer Allocation Based on Marginal Gains, SIGMOD 1991: 387-396.

[8] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, David A. Wood: DBMSs on a Modern Processor: Where Does Time Go?, VLDB 1999: 266-277

[9] M. Zukowski, P. A. Boncz, N. Nes, S. Heman: MonetDB/X100 - A DBMS In The CPU Cache. IEEE Data Engineering Bulletin, 28(2):17-22, June 2005.