An Observation on Database Buffering Performance Metrics

Robert B. Hagmann

Xerox Palo Alto Research Center

Abstract: Database buffering normally assumes that a proper measure of performance is the number of pages transferred. This paper justifies that the number of I/O's is the proper unit of measure. From this we derive a buffering policy that improves over known buffering policies for nested loop joins, we derive some buffering policies for hashing joins, and we make an observation about selections and query optimization.

1. Introduction

Buffer management is a component of database systems that has received much attention over the years. Normally, the performance metric used was the number of page replacements (e.g., [Seli79, Chou85]). With changing hardware costs and performance, there should be a shift in the metric used to judge the performance of the algorithms. In this paper, we will propose a change in the metric, and show three instances where the metric change is important.

Performance metrics for buffer management are used in at least three ways in a database system and its measurement. First, the query optimizer has to estimate the cost of a query using a metric. Second, query execution implicitly or explicitly uses a metric when it makes buffering decisions. Finally, performance measurement of database systems uses a metric to compare different systems or to tune a system.

In this paper, we will develop some initial results using as our metric the *number of Disk I/O's* instead of the conventional metric of the *number of page replacements.* This metric is justified because of three changes: in software systems, hardware cost, and hardware performance. Throughout this paper, we will use the term "disk" to refer to any secondary storage device, but our example and primary model will be moving head, rotating, magnetic disks. We also assume that the database system has control over the granularity of the transfer to secondary memory. The control may be probabilistic or approximate (e.g., bad page substitution may slow down some transfers).

First of all, adjacent pages in files are now often clustered together on the disk. For the UNIX Operating System, the high performance file system for 4.2 BSD [McKu84] has replaced the original file system [Ritc74]. Where the original file system would often allocate adjacent file pages to widely separated disk addresses, the new file system allocates pages in contiguous runs (with high probability). Many systems have always had this property (e.g., System R [Astr76]), but many of the studies of database systems were done using an INGRES Database System [Ston76] which used the original UNIX file system.

Second, main memory costs have decreased so that large buffers of thousands of pages are possible. It is common to buy high performance workstations with 2, 4, or more megabytes of main memory. It is therefore possible to allocate more than a few pages to operations such as joins.

Third, while disk access times have decreased over the years, the real performance gain has been in the transfer rate. The Fujitsu M2350A transfers, in parallel, on up to five disk heads at once and has a sustained formatted transfer rate of 8 megabytes per second [Fuji84]. The Fujitsu M2351 "Eagle" has one fifth of this transfer rate, and is a common high performance drive. The characteristics of this drive are a 5.5 millisecond track to track seek, an 18 millisecond average seek, a 35 millisecond maximum seek, a 7.6 millisecond average latency, and about a 1.57 megabyte per second sustained formatted transfer rate. To see whether access or transfer times dominate for the "Eagle," we will compare the times for some transfer sizes (a similar discussion occurs in [Wein82]). Access time (average seek plus average latency) accounts for 95% of the total time when two 1024 byte pages are read. Access and transfer time are equal when 39 pages are read. Transfer time accounts for 95% of the I/O time when 748 pages are read. For the M2350A, the number of pages for 5%, 50%, and 95% transfer time are 10, 200, and 3800. From this we infer that the time cost of doing a typical I/O for a database system will be dominated by, or at least can be approximated by, the access cost.

By using the "rotational synchronization" feature of the Fujitsu M2350A, up to 8 drives can be synchronized to transfer in parallel. Hence, this disk subsystem has a sustained formatted transfer rate of 64 megabytes per second. If the transfer size remains the same, the use of this feature emphasizes the dominance of the access time even more.

One further observation is that the CPU cost of an I/O is usually nearly the same irrespective of how many pages are read or written. The system call time, context switch times, and interrupt processing are mostly constant for any disk I/O.

Therefore, doing fewer I/O's for the same number of pages will save CPU time. As many systems are CPU bound, saving processor time may be the dominant effect from a change in the buffering.

However, extending transfer size per I/O can have a negative performance impact if the additional data is not used. The I/O's require additional main memory for buffers, may force out desirable data, and contend for the machine's backplane, memory modules, and memory bus. Virtual memory systems face a similar problem while satisfying a page fault, and many systems read in an interval of pages including the faulted page in the expectation that the extra pages will be useful (the author added this to the Cedar Programming Environment [Swin85]). In addition, there has been, over the years, an increase in the page size for file systems and database systems because of the decrease in cost of main memory and the increase in transfer rate.

For some types of channels, the channel controller is used to initiate the I/O to a disk and then the controller is freed. The controller is reacquired when the transfer is about to occur. If the controller is a bottleneck, then making larger transfers may degrade system performance.

Some systems have many disk drives per disk controller and allow parallel seeks to occur on multiple drives. The seek time is thus overlapped with other seeks and transfers. This lessens the impact of the seek time on the *bandwidth* of the disk sub-system, but in no way decreases the query response time (*latency*). Even the bandwidth is affected since rotational latency (average of 7.6 milliseconds above) often consumes a large part of the time the controller is dedicated to an I/O. With disks storing more data per spindle (sometimes with slower access times, as is common with today's optical disks), the benefits of parallel seeks decreases, since contention for each drive increases.

It is only reasonable to do extensive buffering where tuples are clustered and stored in (nearly) contiguous file pages based on the clustering criterion. Most implementations of ISAM, B-Tree, and heap satisfy this requirement. Hashed access also partially satisfies this requirement if few pages contain no tuples and we are doing a full relation scan. We must avoid buffering pages that contain no useful information.

We will use a relational database system in the examples below. Relational systems are non-navigational. Because of this, buffering decisions can be made during query planning.

2. Buffer Management for Database Systems

Buffer management can be critical for good database performance. It has been studied by many researchers. We will not attempt a full literature survey here, but only attempt to show the breadth of the field. There have been studies of the double paging problem and operating system interference [Tuel76, Lang77, Ston81, Wein82, Trai82]. Query optimization normally estimates CPU, communication, and page replacement cost [Ston76, Seli79, Good79, Yao79, Brat84]. A few papers that deal directly with buffer management are [Reit76, Kapl80, Sacc82, Nybe84, Chou85]. Many researchers have studied sequentiality [Powe77, Smit78].

The prime difference of this paper from preceding work is the buffering component of the optimization metric. As stated in the abstract, this paper justifies the *number of Disk I/O's* instead of the conventional metric the *number of page replacements* as the new metric to be used for disk accesses.

A very recent paper [Mack86] uses the number of I/O's as the I/O metric. We also assume that many commercial database systems use the number of I/O's as their metric, but the literature still uses the page replacements.

Won Kim introduced the nested-block method of computing nested loop joins [Kim80]. A nested loop join was previously performed by joining a single tuple from the inner relation with all the tuples of the outer relation. The nested-block method loads pages from the inner relation and joins them with the outer relation. Part of his cost function, however, is the count page replacements – not the number of disk I/O's. His paper proposes a heuristic, where we have an approximation of an exact solution for a two way join. Our solution can be exact because our metric yields a cost function which is mathematically simpler to minimize and because we only deal with a two way join.

3. Nested Loop Join

Consider the example of a nested loop join [Seli79]. Nested loop joins are simple to program and appear in many relational database systems. Conventional wisdom is that nested loop joins only work well when all of the smaller relation, plus one or more pages of the larger relation fits in memory.

We will consider the case where no pre-selection of either relation is possible (e.g., a Cartesian product or natural join with no selection conditions) and there is no collocation of tuples from different relations (i.e., tuples on a page are all from the same relation). It is thus necessary to process all the tuples in one relation against all the tuples in the other relation. Let us derive the number of I/O's necessary to read all the data to perform the join. We will omit any I/O's necessary to write the join results since the amount of data will be the same when generated using any method.

We have two relations, R_1 and R_2, where we can assume, without loss of generality, that R_1 is smaller than R_2 (although we do not use this fact in the derivation but only in the comparison to previous methods). The sizes of R_1 and R_2 will be denoted by $\|R_1\|$ and $\|R_2\|$. Assume that we have assigned N pages for the join from the buffer pool, and have divided the pages into N_1 pages for R_1 and N_2 pages for R_2 ($N_1 + N_2 = N$). Conventional wisdom states that you load one page from the larger relation, R_2, and as much of the smaller relation, R_1, into the buffer pool as will fit, and then do the join. Some algorithms actually do only one tuple at a time from the R_2 page, but it is much better to do all the tuples on the page at once [Kim80]. The inner loop is reading pages from R_2 a page at a time into the buffer, and the outer loop is bringing in sections of R_1.

Below we will need to know the "number of buffer-fulls" it takes to read a whole relation. We will call these $BufferFull_1$ and $BufferFull_2$ and these are just the $RoundUp(\|R_1\|/N_1)$ and $RoundUp(\|R_2\|/N_2)$ respectively (RoundUp is a function from reals to integers that returns the smallest integer greater than or equal to its argument).

Using a typical algorithm to do the join, the number of pages read and number of I/O's neccessary to do the join are:

pages read: $||R_1|| + ||R_2|| * BufferFull_1$

number of I/O's: $(BufferFull_2 + 1) * BufferFull_1$

The system repeatedly loads up N_1 pages of R_1 into the buffer pool, and joins all of R_2 with it. The join is done by reading R_2 from start to finish. There are $BufferFull_1$ iterations, and each iteration takes one I/O to read pages from R_1 and $BufferFull_2$ I/O's to read R_2.

This is not the best that can be done. Instead of reading the relation from start to finish each iteration, the system can process the inner relation first forward, then backward ("rocking the relation through the buffer") [Kim80]. This decreases by one the number of I/O's per iteration, except for the first iteration. More important to our purpose here, it simplifies the "number of I/O" equation.

pages read:
$||R_1|| + N_2 + (||R_2|| - N_2) * BufferFull_1$

number of I/O's:
$1 + 1 + (BufferFull_2 - 1 + 1) * BufferFull_1$

N_2 fewer pages per iteration are read. There is only one I/O saved per iteration, but there is an addition of one extra I/O to get the algorithm started.

The number of I/O's reduces nicely to $2 + BufferFull_1 * BufferFull_2$. If we assume that the relations are quite a bit bigger than the buffers, then we can approximate the discrete case, computed here, by the continuous case. To do this we just omit the "RoundUp" in the computation of the "number of buffer-fulls", and we get

number of I/O's:
$2 + RoundUp(||R_1||/N_1) * RoundUp(||R_2||/N_2)$

$\approx 2 + (||R_1||/N_1) * (||R_2||/N_2)$

$= 2 + ((||R_1|| * ||R_2||)/(N_1 * N_2))$

Differentiating by N, and setting the result to zero to find the maximum:

$0 = ((||R_1|| * ||R_2||)/(N-N_1)^2 * N_1) - ((||R_1|| * ||R_2||)/(N-N_1) * N_1^2)$

Now, if $N_1 \neq 0$, $||R_1|| * ||R_2|| \neq 0$, and $N \neq N_1$

(i.e., neither relation is empty and we give at least one page for buffering to each relation)

$0 = (1/(N-N_1)) - (1/N_1)$

$\Rightarrow N_1 - (N - N_1) = 0$

$\Rightarrow 2 * N_1 - N = 0$

$\Rightarrow N_1 = N/2$

$\Rightarrow N_1 = N_2$

Thus, the best buffering policy is to split the buffers evenly between the two relations! Note that this was computed by approximating the discrete case with the continuous one, so care should be used to insure the approximation is reasonable. For example, if $||R_1|| < N/2$, then R_1 clearly does not need N/2 buffers.

To see how this works, consider a case where we have 100 pages in a pool for a join. $||R_1||$ is 99 pages, and $||R_2||$ is 10,000 pages. Conventional wisdom says to read all 99 pages of R_1 into the buffer and process R_2 a page at a time against R_1. This takes 10,001 I/O's, but only transfers each page once. If we split the buffer space, then it now takes 2 I/O's to read R_1, but 200 I/O's to read R_2 forward, and 199 I/O's to read R_2 backward. The total is 401 I/O's. We read more pages (about double), but we do only about 4% as many I/O's.

An extension of this technique is to do double buffering of the outer relation so as to overlap the join execution with the fetch of the next run of pages. This will do more disk I/O, but may decrease response time since the join may run in less time due to the overlap.

4. Hashing Join

There are many variants of the hashing join. The one considered here is the hashing join that partitions both relations described in [Brat84]. The idea is to first hash one relation, then the other. The corresponding hash buckets from the two relations can then be joined. The hash should be chosen so that (ideally) the corresponding hash buckets both fit in memory.

Assuming we are using single buffering, what is the best division of the buffers for a hash partition? Assume that we have assigned a total of N pages from the buffer pool. Let F denote the number of pages fetched for input of relation R. The output for the hash buckets uses c buffers (one for each value of the hash) of size W pages each. The value of c is the size of the range of the hash function. Both N (total buffer pages) and c (number of hash buckets) are constants for the derivation. As each tuple is hashed, restriction processing and duplicate elimination is also done. There will be less fragmentation in the hashed output than in the original relation since page boundaries do not have to be observed in the output. The combined compression effect of restriction, duplicate elimination, and less fragmentation will be combined into one factor called Compression.

number of buffers (N) = $F + c * W$

number of I/O's: $RoundUp(||R||/F) + c * RoundUp(Compression * (||R||/c)/W)$

The first factor in the number of I/O's is for reading the input relation. For each output buffer, it must be written about $Compression * (||R||/c)/W$ times if we assume a uniform distribution ($||R||/c$ is about the number of output pages; they are written W pages at a time with a compression of Compression). Taking an upper bound for RoundUp gives:

$||R||/F + 1 + c * Compression * (||R|| / c) / W + c$

$= ||R||/F + Compression * ||R|| / W + c + 1$

The distribution of the tuples now makes no difference since we took the upper bound. Differentiating by F gives:

$-||R||/X^2 + ||R|| * Compression * (c / ((N - F)^2))$

Assuming $||R|| \neq 0$, $N \neq F$, and $F \neq 0$, and setting the result to 0 to find the minimum gives:

$0 = -1/X^2 + N * Compression / (N - F)^2$

$= N^2 - 2 * F * N + X^2 - X^2 * c * Compression$

$= (1 - c * \text{Compression}) X^2 - 2 * F * N + N^2$

If c * Compression = 1, this reduces to F = N / 2 (use half the buffer space for input). Otherwise, apply the quadratic formula to find F:

$F = (2 * N \pm \text{SQRT}(4*N^2 - 4*(1-c*\text{Compression})*N^2))/(1 - c * \text{Compression})$

$= (N - N * \text{SQRT}(c * \text{Compression})) / (1 - c * \text{Compression})$

Normally, $1 - c * \text{Compression} \leq 1$, so only the "−" term applies.

For example, an eight way partition with compression of 0.5 has F = N / 3 while a 16 way partition with no compression has F = N / 5.

5. Selecting by an Attribute that is Indexed and Clustered

Suppose we have a relation that is clustered by an attribute, and we wish to fetch the tuples via this attribute, where we also have an index on the attribute. A B-Tree, indexed by the appropriate attribute, is an example of such a storage structure for a relation. We may be doing the inner or outer relation scan for a nested loop join, doing a semi-join step, initializing for a hashing join, or performing a combined selection and projection.

A typical way to find the tuples is to use the index (e. g., the internal nodes of the B-Tree) to successively find tuples for the join, and let the page buffering take care of itself. The access to the disk may appear to have some sequentiality or may appear to be somewhat random.

If the order of the tuples is not important (e.g., they are going to be sorted anyway), then a better way to find all the tuples and minimize I/O's is to first use the index to discover all the pages that contain tuples we need. Compose these page numbers into page runs (sets of contiguous pages) of a size appropriate for the buffer pool. Finally, read in and process each page run.

At worst, this takes the same number of I/O's as doing a tuple at a time access. However, it is likely that this algorithm will do fewer I/O's, but will read the same number of pages.

Runs that are near to each other can be processed together by adding the pages in between the runs to form one run, reading the merged run, and ignoring the extra pages once they have been read. Head scheduling may also be done to minimize seek time.

A database system can often predict the I/O's it will need ahead of time [Ston81]. An operating system can only react to stimuli or take hints as they are given. Here we compute the pages necessary to be read before running (part of) the query, and plan how to read the pages minimizing the number of I/O's.

6. Conclusions

In this paper we have justified the use of the *number of Disk I/O's* instead of the conventional *number of page replacements* as a performance metric for database buffering. From this metric, we then derived a new buffering policy for nested loop joins, computed the buffering for the partitioning of a hashing join, and made an observation about how to process selections for attributes that are indexed and clustered.

Trademarks

UNIX is a trademark of Bell Telephone Laboratories.

References

[Astr76] Astrahan, M. et al. System R: Relational Approach to Database Management. *ACM TODS* **1**, 2, June 1976, 97-137.

[Brat84] Bratbergsengen, B. Hashing Methods and Relational Algebra Operations. *Proceeding of the 10th International Conference of Very Large Data Bases.* Singapore, Aug. 1984, 323-333.

[Chou85] Chou, H., and DeWitt, D. An Evaluation of Buffer Management Strategies for Relational Database Systems. *Proceeding of the 11th International Conference of Very Large Data Bases.* Stockholm, Aug. 1985, 127-141.

[Fuji84] Fujitsu America, Inc. "OEM Parallel Data Transfer Disk Drive," Dec. 1984.

[Good79] Goodman, N., Bernstein, P., Wong, E., Reeve, C., and Rothnie, J. *Query Processing in SDD-1: A System for Distributed Databases.* Computer Corporation of America Technical Report CCA-79-06, Oct. 1979.

[Kapl80] Kaplan, J. *Buffer Management Policies in a Database Environment.* Master's Report, UC Berkeley, 1980.

[Kim80] Kim, W. A New Way to Compute the Product and Join of Relations, *Proceeding of ACM-SIGMOD 1980.* Santa Monica, May 1980, 179-187.

[Lang77] Lang, T., Wood, C., and Fernández, I. Database Buffer Paging in Virtual Storage Systems. *ACM TODS* **2**, 4, Dec. 1977, 339-351.

[Mack86] Mackert, L. F., and Lohman, G. M. R* Optimizer Validation and Performance Evaluation for Local Queries. *Proceedings of SIGMOD '86,* Washington, May 1986, 84-95.

[McKu84] McKusick, M., Joy, W., Leffler, S., and Fabry, R. A Fast File System for UNIX. *ACM TOCS* **2**, 3, Aug. 1984, 181-197.

[Nybe84] Nyberg, C. *Disk Scheduling and Cache Replacement for a Database Machine.* Master's Report, UC Berkeley, 1984.

[Powe77] Powell, M. The DEMOS File System, *Proceedings of the Sixth Symposium on Operating Systems Principles.* West Lafayette, Nov. 1977, 33-42.

[Reit76] Reiter, A. *A Study of Buffer Management Policies for Data Management Systems.* Technical Summary Report #1619, Mathematics Research Center, University of Wisconsin-Madison, March 1976.

[Ritc74] Ritchie, D., and Thompson, K. The UNIX Time-sharing System. *CACM* **17**, 7, July 1974, 265-375.

[Sacc82] Sacco, G., and Schkolnick, M. A Mechanism for Managing the Buffer Pool in a Relational Database System using the Hot Set Model. *Proceedings of the 8th International Conference of Very Large Data Bases.* Mexico City, Sept. 1982, 257-262.

[Seli79] Selinger, P., Astranhan, M., Chamberlin, D., Lorie, R., and Price, T. *Access Path Selection in a Relational Database Management System.* IBM San Jose RJ2429, 1979.

[Smit78] Smith, A. Sequentiality and Prefetching in Database Systems. *ACM TODS* 3, *3*, Sept. 1978, 223-247.

[Ston76] Stonebraker, M, Wong, E., and Kreps, P. The Design and Implementation of INGRES. *ACM TODS* 1, *3*, Sept. 1976, 189-222.

[Ston81] Stonebraker, M. Operating System Support of Database Management *CACM* **24**, *7*. July 1981, 412-418.

[Swin85] Swinehart, D., Zellweger, P., and Hagmann, R. The Structure of Cedar, *Proceeding of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*. Seattle, June 1985, 230-244 (SIGPLAN Notices **20**, *7*).

[Trai82] Traiger, I. Virtual Memory Management for Data Base Systems *OSR* **16**, *4*, Oct. 1982.

[Tuel76] Tuel, W., Jr. An Analysis of Buffer Paging in Virtual Storage Systems. *IBM Journal of Research and Development* **20**, *5*, Sept. 1976, 518-520.

[Wein82] Weinberger, P. Making UNIX Operating Systems Safe for Databases. *Bell System Technical Journal* **61**, *9*, Nov. 1982, 2407-2422.

[Yao79] Yao, S. B. Optimization of Query Evaluation Algorithms. *ACM TODS* **4**, *2*, June 1979, 133-155.