Storage Management in RDBMS

Wenguang Wang
Department of Computer Science
University of Saskatchewan
57 Campus Drive
Saskatoon, SK S7N 5A9
Canada
Email: wew036@cs.usask.ca

August 17, 2001

Abstract

Storage management is important to the performance of DBMS. This paper gives a comprehensive overview of storage management in relational DBMS. The storage management is divided into three levels (logical, physical in-memory, and physical on-disk) and discussed separately. The logical level caches logical units (tuples or index values) of the DBMS based on the logical information to improve buffer pool usage efficiency. The physical in-memory level caches physical pages directly in the buffer pool to reduce disk accesses. Many algorithms have been designed for the general buffer or for the buffer pool of DBMS. Complex algorithms can achieve better performance than simple LRU or CLOCK algorithms. However, the overhead is high and the advantage diminishes when the buffer is big. The physical on-disk level borrows many techniques from file systems and storage systems. Because of the excellent potential write performance of log-structured organization, the Log-structured File System and related topics are discussed. Each level has its advantage and limitation for further improving performance. To achieve better performance of storage management systems of DBMS, all these three levels should be considered.

1 Introduction

In relational Database Management Systems (DBMS[1]), the database is placed on secondary storage (disk). When the data in the database are required, they need to be read into memory and may be written back to disk if updated. The data must be organized on the disk for efficient accessing. In order to increase the efficiency of storage management, part of memory is used to cache some of these data. The management of data on disk and in memory belongs to the storage management of DBMS.

The disk access is the bottleneck of the database system in many workloads, because the access time of disk is several magnitudes longer than that of memory. The objective of storage management is to improve the efficiency of disk access, therefore to improve the performance of the whole system. The storage management can be divided into three levels:

1. **Logical level** is the higher level. In this level, the logical information of the DBMS is used to optimize the data stored in memory. Not much research work is on this level and many DBMSs do not have this level in their storage management.
2. **Physical in-memory level** is the middle level. In this level, the popular database pages are cached in the in-memory space *buffer pool*. The buffer pool can absorb most requests to the disk therefore reduce the number of disk accesses. This approach is used in almost all DBMSs and has important effects on their performance. There are many studies about the general buffer management and the buffer pool management in DBMS.

[1]DBMS refers to relational DBMS in this paper unless otherwise stated

3. **Physical on-disk level** is the lower level. The layout of database data and meta-data on the disk is studied to improve the disk I/O efficiency. It can be further divided into two sub-levels: *software level* and *disk level*. In the software level, the on-disk data structure and data placement strategy are designed based on the logical structure of data. This area has been studied intensively in the design of file systems. In the disk level, the logical structure of data is not considered. Instead, only the physical nature of disk is utilized to improve the disk I/O. This area has been studied in the design of storage systems (e.g. RAID systems).

This survey gives a comprehensive overview to important techniques used in or related to storage management in DBMS. First the typical workloads of DBMS are introduced in § 2. The three levels of storage management are discussed after that. The physical in-memory is discussed first in § 3. The physical on-disk level is discussed in § 4. The logical level is discussed in § 5. § 6 is the conclusion.

2 Typical Workloads of DBMS

The workloads of DBMS applications show diversity due to the application of DBMS in different areas. Important workloads can be categorized into three typical types: OLTP (On-Line Transaction Processing), OLAP (On-Line Analytical Processing), and their combination. The Transaction Processing Performance Council (TPC)[2] is a non-profit corporation founded to define transaction processing and database benchmarks. The TPC benchmarks are widely used for testing database performance. These typical workloads and their corresponding TPC benchmarks are briefly discussed in this section.

2.1 On-Line Transaction Processing (OLTP)

In the OLTP workload, most transactions are simple queries. The execution time of each transaction is short (can be tens of milliseconds or shorter). Many users access the database at the same time. The database is updated frequently. Examples of the OLTP workload are air tickets ordering system and ATM for banks.

The TPC-C benchmark is often used to test the performance of DBMS on the OLTP workload. It consists of a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. The performance metric reported by TPC-C is a "business throughput", measuring the number of "orders" processed per minute. Multiple transactions are used to simulate the business activity of processing an order, and each transaction is subject to a response time constraint. There are five kinds of transactions in the TPC-C benchmark, which are listed in Table 1.

Name	Characteristic	Percentage
New-Order	read-write, mid-weight	45%
Payment	read-write, light-weight	43%
Order-Status	read-only, mid-weight	4%
Delivery	read-write	4%
New-Order	read-only	4%

Table 1: TPC-C Transactions

Because the New-Order transaction is the backbone of the workload, the performance metric of the TPC-C benchmark is expressed in Transactions Per Minute (TPM), which is the number of New-Order transactions executed per minute. The size of the TPC-C database is given by the number of "warehouses" defined. There are about 100M bytes of data for one warehouse.

2.2 On-Line Analysis Processing (OLAP)

In the OLAP workload, many complex queries are used to query large amount of data in the database for decision support. The execution time of each query is long (from tens of seconds to hours). Normally only one user is running the query. The queries mainly read the database without updating.

[2] http://www.tpc.org

The TPC-D benchmark is used to model the OLAP workload. However, it is obsolete and has been split into two benchmarks TPC-R and TPC-H. If the types of queries are known ahead of time, some optimization techniques such as the AST (Automatic Summary Table) tables can be used to speed up the query greatly. The TPC-R benchmark (R for business Reporting) models this type of workload. If the types of queries are not known ahead of time, optimization techniques like AST cannot be used. The TPC-H benchmark (H stands for ad-Hoc queries) simulates this workload.

The performance metric reported by TPC-H is called the TPC-H Composite Query-per-Hour Performance Metric (QphH@Size), and reflects multiple aspects of the capability of the system to process queries. These aspects include the selected database size against which the queries are executed, the query processing power when queries are submitted by a single stream, and the query throughput when queries are submitted by multiple concurrent users. Similarly, the performance metric reported by TPC-R is called QphR@Size.

2.3 Mixing OLTP and OLAP

Some real workloads has been analyzed and compared with the TPC-C and TPC-D benchmark [23, 24]. It was found that many characteristics of the real workloads can be represented by TPC-C and TPC-D. However, the real workloads show both type of characteristics, and their characteristics change over time.

In transactional e-commerce applications, both OLTP and OLAP workload exist. Users of these applications can buy or browse products through network. When users buy products, they generate OLTP workload. When users browse products (say, search the top 10 hottest products), the workload generated is OLAP like. Normally the management and tuning of DBMS for OLTP and OLAP are different. When these two types of workloads occur together, it is challenging for the tuning of DBMS. The TPC-W benchmark simulates this workload.

3 Physical In-memory Level – Buffer Pool Management

The physical in-memory level of the storage management caches disk pages in an in-memory buffer called *buffer pool* without knowing the logical structures of the pages. This level is call buffer pool management conventionally. The objective of buffer pool management is to reduce the page miss ratio thus reduce number of disk accesses.

The buffer pool management in DBMS is similar to many other buffer (CPU cache, virtual memory, file system buffer, etc.) management strategies. In all these buffer management systems, there is a speed gap between the buffer and the lower level storage. The access to the buffer is much faster than the access to lower level storage. However, the buffer is more expensive than the lower level storage. Because of the locality of references to the data, popular data can be kept in a small buffer. It is likely that the data can be found in the buffer when they are needed. Therefore, the overall access time is close to the buffer access time, while the overall cost per storage unit is close to the lower level storage.

Some general buffer replacement algorithms can be used in the buffer pool management of DBMS because of the similarity of all buffer management algorithms. These general algorithms are discussed in § 3.1. The reference behaviour of DBMS buffer pool has its own properties. In order to design proper replacement algorithms for buffer pool, the reference behaviour of DBMS applications were studied in many works. This part is discussed in § 3.2. Many buffer pool replacement algorithms are designed based on the reference pattern of DBMS. Some representative algorithms of them are discussed in § 3.3.

3.1 General Buffer Management

3.1.1 Basic replacement algorithms

The basic replacement algorithms that have been considered in the literature include LRU (least recently used), FIFO (first-in-first-out), CLOCK, and LFU (least frequently used).

The LRU algorithm replaces the page in memory that has not been referenced for the longest time. By the principle of locality, this should be the page least likely to be referenced in the near future. The LRU algorithm is very simple and shows good performance in many cases. However, because it need to update the recency information on each reference, the overhead of LRU is unacceptable when the buffer is managed

by or partly by hardware (e.g. CPU cache and virtual memory). In DBMS, the buffer pool is managed by software, and the overhead of LRU is acceptable. In old version of DB2, the LRU was used as the buffer pool replacement algorithm [56]. A variation of LRU is also used in the ADABAS [47]. Whenever the LRU queue is updated, it need to be latched, which occurs on every page reference. Because there are many users accessing the database at the same time, the LRU queue latch can become a bottleneck. Multiple LRU queues are used in ADABAS to reduce the latch contentions on the LRU queue. These LRU queues are accessed in a round robin manner. When searching an LRU queue, only this queue is latched.

The FIFO policy replaces the page that has been in memory the longest. It is easy to implement and has low overhead but performs relatively poorly than the LRU policy. This is because the hot pages will be replaced from the buffer periodically.

The CLOCK algorithm is derived from the FIFO algorithm and is an approximation of the LRU algorithm. In this algorithm, a reference bit is associated with every page. When the page is referenced, its use bit is set to 1. All pages in the buffer are organized into a circular queue. A clock pointer is associated with the circular queue. When a replacement is needed, the clock pointer will go through the buffer in one direction from the last position until finding a page with reference bit 0. All pages scanned by the clock pointer have their reference bit set to 0. The CLOCK policy has similar overhead as the FIFO policy but approximate the performance of LRU.

The LFU policy replaces the page in memory that has been referenced the least times. A problem of the LFU algorithm is that some pages may build very high reference frequency and can not be replaced. There are some variations of the LFU algorithm to overcome its problem (LFU*, LFU-Aging, LFU*-Aging) [25, 61, 1]. The high overhead of LFU also limits its application. The CLOCK algorithm can be modified to approximate LFU, if there are several reference bits, and they are increased by 1 on every reference. Linux uses this algorithm to manage its virtual memory [15]. Algorithms that consider both recency and frequency can achieve better performance. The algorithm LRFU that combines the LRU and LFU shows good performance [29]. This algorithm will be discussed below.

3.1.2 More sophisticated algorithms

GCLOCK and DGCLOCK

There are several variations of the basic CLOCK algorithm [16]. In the GCLOCK (Generalized CLOCK) algorithm, each page in the buffer is assigned a reference counter. There are different variations of GCLOCK algorithm based on how the reference counter is updated. When a page is brought into the buffer, an initial value (weight) is assigned to the counter of that page. The weights may be different for different data types. On a buffer page hit, the counter associated with that page is reset (not necessarily to the initial count) or incremented. On a page miss, a circulating clock pointer sweeps through the buffer pages decrementing their counters until a page with a count of zero is found. The GCLOCK algorithm can be refined to DGCLOCK in which dynamically calculated, page-related weights are assigned. The GCLOCK and DGCLOCK algorithm are flexible due to their various versions and parameters. Optimal parameter values are difficult to determine and have to be tailored to a specific DBMS and specific applications. Because of its tuning ability, the GCLOCK algorithm was used in the Starburst project [22]. When proper weights are used for special reference patterns, better performance than LRU can be achieved. However, in complex situations that the parameters are hard to tune, its performance is not as good as LRU.

The GCLOCK algorithm is further studied in [38]. An approximate analytical model under the Independent Reference Model (IRM) is developed. The model is applied to a case similar to the TPC-A benchmark. It is found that with equal weights assigned to all partitions, the performance of the GCLOCK algorithm approaches that of the LRU algorithm as the weight is increased. The higher the weight, the closer the GCLOCK algorithm to the LRU algorithm. The reason is that when hot pages are re-referenced, their weights can be reset to a high value, and these are not replaced in the next clock sweep.

Another strategy presented in [38] assigns different weights to different partitions of data, when the information about the access skewness is used. The objective of this strategy is that the colder partition is given a lower priority than the hotter partitions. The simulation and analysis results show that the overall buffer hit probability is better than that of the LRU algorithm if weights are properly assigned. The optimal allocation is to load as much as the hottest partition in the buffer, and as much as the second hottest partition in the rest buffer spaces, and this allocation continues until the buffer is filled up and a partition is allocated

less space than its size. To achieve this allocation, the last partition, which is partly in memory, requires a higher weight. The other hotter and colder partitions need a smaller weight. By assigning appropriate weight to different partitions based on buffer size and partition characteristics, the hit ratio of the GCLOCK algorithm can be very close to the optimal algorithm. In order to set optimal weights for this algorithm in practical environments, both the access skew and the buffer size need to be known in advance. However, the skewness of data is hard to analysis and it can change over time. Therefore, selecting optimal weight values is not easy in real world. If the skewness of the data can be detected on-line, the on-line tuning can be used to improve the performance. However, this could be difficult, as stated in [28]: "looking for data base reference chanracteristics in a raw database reference string is like looking for the structure of eggs in a plate of scrambled eggs", because the interleaving of transaction references would obscure the database reference characteristics.

LRU-K

LRU-K is a replacement algorithm proposed in [39]. The basic idea of LRU-K is to keep track of the times of the last K references to popular pages, and use this information to predict future references. Recording the last K references can overcome a problem of the LRU algorithm: LRU does not discriminate well between frequently and infrequently referenced pages, and once a page is brought in, a long buffer life is guaranteed. In the LRU-K algorithm, the page whose last (k+1)st reference is the oldest will be selected for replacement. When K=1, LRU-K degrades to the LRU algorithm.

In [39], this algorithm was tailored to DBMS after defining two parameters: *Correlated Reference Period* and *Retained Information Period*. After pages are brought into memory and within the correlated reference period, the system should not drop a page immediately after its first reference. At the same time, successive references to this page within the correlated reference period are not recorded as the referencing history of this page, because these references may be correlated. The referencing history of pages are retained no matter these pages are in memory or not if the current time is within the Retained Information Period since the most recent access of these pages.

Simulation experiments were performed on synthetic traces and an OLTP trace. It was shown that when K>2, LRU-K is slightly better than LRU-2, but is less responsive than LRU-2 in the sense that it needs more references to adapt itself to dynamic changes of reference frequencies. Therefore, the authors advocated LRU-2 as a generally efficient policy. The simulation results showed that when the buffer pool is very small, LRU-K is superior to LRU and LFU. However, when the buffer pool is big enough that the hit ratio is more than 38%, the hit ratio difference becomes insignificant. In modern DBMS systems, normally the buffer pool size is big enough that the hit ratio is above 90%. Whether LRU-2 can show significant improvement when the hit ratio is high is not tested in this paper. The overhead of LRU-2 is quite big: each page access requires $\log N$ work to manipulate a priority queue where N is the number of pages in the buffer [26].

2Q

To decrease the overhead of the LRU-2 algorithm, a 2Q (Two Queue) replacement algorithm was proposed [26]. This algorithm has constant time overhead, performs as well as LRU-2, and requires no tuning. The 2Q algorithm uses the same principle of the LRU-2. It not only keeps the last reference to a page, but also keeps the second-last reference. On the first reference to a page, 2Q places it in a special buffer, the A1 queue, which is managed as a FIFO queue. The A1 queue is partitioned into A1in and A1out. The most recent first accesses will be in the memory (up to a certain threshold), which is in the A1in queue. When the A1in queue is full, the older pages in the A1in queue will be thrown out of memory but their header information is remembered in the A1out queue. The size of the A1out queue has the similar effect as the *Retained Information Period* of the LRU-K algorithm. If the page is re-referenced when it is in the A1in queue, nothing happens, because the second access may simply be a correlated reference. When the page is re-referenced when it is in the A1out queue, it is moved to the Am queue, which is managed as an LRU queue. The size of the A1in queue has the similar effect as the *Correlated Reference Period* of the LRU-K. When pages in the Am queue are replaced, they are not remembered in the A1out queue because they have not been referenced for a while. When the size of A1in queue is 25% of the buffer size and the size of A1out queue is 50% of the buffer size, the algorithm works well.

Several simulation experiments were performed on synthetic traces and real traces. They showed that 2Q provides a few percent improvement over LRU for synthetic traces and real traces including an OLTP trace. It was also shown that the hit rate of 2Q suffers less than the hit rate of LRU when workloads not suitable for LRU exist. When the buffer hit rate is high (more than 70%), the improvement of 2Q over LRU is less than 1%.

SEQ

LRU performs well in common cases, but performs badly in some cases (e.g. when a loop which is bigger than the buffer size exists). Several algorithms (SEQ [19], EELRU [52]) are designed to overcome this problem.

The intuition behind the SEQ replacement algorithm is to detect long sequences of page faults and apply MRU replacement to such sequences. The key point of the SEQ algorithm is how to detect and manage sequences on-the-fly. Only a series of page faults to consecutive virtual addresses is considered as a sequence. Therefore, the loop access to arrays is a sequence. However, loop access to linked lists or trees are not considered as sequences because the consecutive items in such data structures do not necessarily have consecutive virtual addresses. Therefore, only part of loop access can be detected. The SEQ algorithm keeps trace of many sequences detected (say, 200). Only sequences of length greater than a threshold (currently 20 pages) are replaced by MRU. Simulation results show that SEQ performs significantly better than LRU and quite close to optimal for applications with clear access patterns. For other applications, the performance of SEQ is similar to LRU.

EELRU

The EELRU algorithm [52] was designed to overcome the problem that LRU cannot process big looping behaviour well. It uses only the recency information of LRU. Recency information of pages that have been evicted is also maintained. An early eviction point is used in the LRU chain. Pages can be evicted either from the early eviction point or the end of the LRU chain. A benefit function is defined to express the benefit for evicting pages from an early eviction point. The location of the early eviction point is computed dynamically based on the benefit function. In common situations where LRU is good, EELRU is also good because it behaves like LRU. In common worst cases for LRU, EELRU is significantly better. This algorithm shows good performance for loops bigger than the buffer size. In the worst cases, EELRU is at most a constant factor worse that LRU. In most cases, EELRU is better than LRU.

LRFU

The idea of the LRFU (Least Recently/Frequently Used) algorithm [29] is to consider both the recency and frequency information of past references when making replacement decisions. Therefore, it is a combination of the LRU and LFU algorithms. This algorithm records the recency and frequency information of the reference to the page through the CRF (Combined Recency and Frequency) value which is associated to each page in the buffer. A recent reference to a page has a higher weight in the CRF value than an older reference. This CRF value is directly used to select victim pages. A parameter λ is used to control the CRF value. When λ changes between 0 and 1, the algorithm can behave between LRU and LFU.

Several simulation experiments were performed on file system traces, database traces, and OLTP traces. Simulation results show that LRFU has better hit ratio than LRU-2, 2Q, and LRU algorithm when the buffer size is small. Proper implementation of this algorithm only has small overhead.

This algorithm was implemented in FreeBSD and showed better performance than the LRU algorithm. When the buffer size is very large, the hit ratio difference between different algorithms becomes very small. However, due to the huge penalty of physical disk I/Os, a slight increase in the hit ratio results in a considerable improvement on the throughput. In the FreeBSD tests, when the cache size is 100 pages, the hit rate difference between the LRU and the LRFU is less than 0.5%, but this leads to more than a 4% difference in throughput.

The optimal value of parameter λ is different for different workloads. A preliminary self-adapting λ algorithm is proposed. Under this algorithm, the λ is adjusted periodically based on the change of the buffer hit ratio. If the hit ratio of period i is better than that of period $i-1$ and the λ value increased, the λ

is incremented. On the other hand, if the hit ratio decreased from period $i-1$ to period i, and the λ is increased, the λ is decreased.

Cache Partitioning

All algorithms discussed above are global algorithms. In these algorithms, the buffer is considered as one entity. Therefore, the pages in the whole buffer are considered when selecting a replacement. Another scheme is called partitioning, in which the buffer consists of several partitions. When a page is needed, only pages in the same partition is considered for replacement. Therefore, the behaviour of one partition will not affect other partitions. In [55], a model for studying the optimal allocation of cache memory among two or more partitions is developed. It shows that the optimal fixed allocation of cache among two or more partitions is an allocation for which the miss-rate derivative with respect to cache size is equal for all partitions. The paper also studies the transient in buffer management when program behaviour changes. When program behavior changes, the LRU algorithm moves quickly toward the steady-state allocation if it is far from optimal, but converges slowly as the allocation approaches the steady-state allocation.

An adaptive algorithm for managing fully associative cache memories shared by several identifiable processes is proposed in [57]. This algorithm partitions the cache in disjoint blocks whose sizes are determined by the locality of the processes accessing the cache. When the optimum partition is reached, the frequency-weighted miss rate derivatives of all processes with respect to cache size are identical. The frequency-weighted miss rate when the cache size is i is approximated by the difference between the number of misses of a process in cache size i and the number of misses of this process in cache size $i+1$. To measure the history of a process and to adapt the change of the characteristics, the past intervals are weighted by factors that diminish by powers of a parameter C_r. Each partition is managed by LRU. This partition algorithm can achieve 1-2% better performance than LRU. When the disk load is heavy, the response time can improve significantly.

3.1.3 Algorithms with application level hints

Application controlled caching

Because the workloads in computer systems or DBMS are very diversified, it is hard to find a universal replacement algorithm that is good in all cases. Application controlled caching [4] is proposed to overcome this problem. This approach can apply application-specific knowledge to improve file cache performance. It is implemented by a two-level cache management strategy. The kernel allocate blocks to processes. Each process can submit its prefetches in batches, assign priorities to their own files or file blocks, and for each priority level, to specify file cache replacement policies.

The measurements showed that this technique can greatly improve the performance of the file system. The running time is reduced by 3% to 49% (average 26%) for single-process workloads and by 5% to 76% (average 32%) for multiprocess workloads.

The application that uses this approach must be modified to inform the file system to select proper replacement policies and prefetch data. This may not be feasible for some applications.

Detecting reference patterns on-line

An adaptive block management scheme DEAR (Detection based Adaptive Replacement) is proposed in [8]. More discussion and measurements about this algorithm can be found in [9]. Unlike Cao's approach, the DEAR scheme detects the reference pattern of an application on-line. *Sequential reference, looping reference, temporal localized reference*, and *probabilistic reference* can be detected from the frequencies, backward distances and forward distances of page references. The MRU algorithm is applied to the sequencial and looping reference. The LRU algorithm is applied to the temporal localized reference. The LFU algorithm is applied to the probabilistic reference. The buffer manager consists of two modules: Application Cache Manager (ACM) and System Cache Manager (SCM). An ACM is allocated to each application. It gathers block attributes and detects the reference patterns periodically. The SCM takes charge of block allocation, list management, and the sending of replacement requests to ACMs when necessary. When a buffer space is needed, the SCM selects an ACM to send a replacement request based on a simple heuristic. This heuristic tries to minimize the cost for replacing a page.

Simulation results show that the DEAR always performs better than LRU for single application and multiple applications. Although the overhead of DEAR is higher than LRU, the saved IO cost of DEAR can compensate this overhead when the buffer is small.

In both the application controlled caching and the DEAR algorithm, the reference pattern is simple for each file, because references to a file are normally from one process. If a file is shared by many processes/threads (which is a typical case in DBMS), the reference pattern is hard to detect.

3.1.4 General buffer management and buffer pool management

The general buffer replacement algorithms discussed in this subsection are not designed for DBMS, although they may be adapted to DBMS buffer pool replacement algorithms.

In an Operating System, applications may not request file access frequently. The typical pattern is reading the file, performing the computation, and writing the result. However, in a DBMS, the pages in the buffer pool are accessed very frequently because all database operations are based on the data of the pages. The request rate to the buffer pool in a DBMS is much higher than that of a file buffer in an Operating System. This implies more requirements for the buffer pool management algorithms:

- *Low overhead.* Very small overhead of the buffer pool replacement algorithm is desired, because the buffer pool has a higher request rate. When the buffer pool is big, the miss ratio between simple algorithms and complex algorithms is small and the overhead may become essential.
- *Simultaneous access.* There are normally many users using the DBMS. Therefore, many processes or threads access the buffer pool concurrently. Because of the high request rate to the buffer pool, the access contention must be solved so that the mutual exclusive access to the data structures of buffer pool cannot become a bottleneck. For example, the LRU algorithm used in ADABAS [47] splits one LRU chain to many LRU chains to avoid the access contention. When other algorithms are adapted to DBMS, it is important to change them so that the buffer pool can be accessed simultaneously.

3.2 Reference Characteristics of DBMS Buffer Pool

Key to the design of buffer pool management strategies is an understanding of how database applications refer to pages from the databases. Empirical studies of database reference behavior over the past mainly focused on the Hierarchical [27, 28, 53, 42] and the CODASYL database systems [17, 58]. Many conflicting results were reported by them.

1. **Randomness**. A study on the reference behavior in CODASYL Database Systems [17] showed that some databases have no locality at all. Moreover, no physical sequentiality was found. The application under study was a single-user retrieval-only on-line application. The size of the databases used was small.
2. **Sequentiality**. Rodriguez-Rosell [42] studied an on-line control system running on IMS which is a hierarchical DBMS. The results showed the lack of locality in database reference behavior within one transaction. However, strong sequentiality was found. Based on this sequentiality, some prefetching techniques were studied to improve the performance [53].
3. **Locality is observed, but less than the locality of programs**. Effelsberg [17] found that database reference shows less locality than do programs under virtual memory operating systems. Contrary to [42], this locality comes from inter-transaction references. Verkamo [58] studied the reference behavior of some batch programs running on a CODASYL DBMS. The locality was found in the database references. However, there was a continuous change in the active group, and no fixed locality set was found. Sequentiality was not found in general cases.
4. **Undeniable Locality**. Kearns and DeFazio's study on the reference behavior of IMS in 1983 [27] showed that the locality of references is an undeniable characteristic of hierarchical database systems. Furthermore, database locality of reference is in a sense more regular, more predictable, and hence, more exploitable than the localized reference activities found in programs in general. The workload studied is generated by batch programs which deals with typically expensive and time-consuming applications. Their later study in 1989 [28] concludes many previous results. They presented their

study on a per-transaction and per-database basis. The results showed that the database reference behavior is predictable and exploitable in this perspective.

5. **Diversity**. In [10], the reference behavior of Relational DBMS was analyzed based on the operation set of relational databases, not by measurements from reference strings. The QLSM model was proposed that maps every database operation into a number of simple reference patterns. Sequential references, random references, and hierarchical references are included in this model. However, the locality of references and the effect of multiple users on the reference pattern were not discussed.

Previous studies showed that the database reference behavior has many different characteristics on different applications and DBMSs. The overall characteristics of the DBMS reference behavior is a mix of different reference patterns.

3.3 Buffer Pool Management Algorithms for DBMS

The general buffer management algorithms discussed in § 3.1 are designed for general purposes like virtual memory, cache, file system buffer, not for the buffer pool management in DBMS. The reference behaviour in DBMS buffer pool is different with that in virtual memory, cache, or file system buffer. In a Relational DBMS, there are a few basic operations defined by the relational algebra [11]. Therefore, only a few reference patterns exist. The reference behaviour of hierarchical indexes is discussed in § 3.3.1. By analyzing a query execution plan before executing it, it is possible to predict its reference behavior. Many buffer pool management algorithms are designed to utilize this property, and show better performance than the general algorithms. This type of algorithm is discussed in § 3.3.2.

3.3.1 Indexes

In relational DBMS, index is built on some fields of the relation and is used to search for a record quickly based on the indexed values. B^+ tree [12] is a commonly used structure for index. When the index is accessed, the index tree is traversed from the root level (lowest level) consisting of a single page, down to the data level (highest level). Therefore, pages in the lower level has a higher probability to be accessed than pages in the higher level.

ILRU and OLRU

Two replacement algorithms ILRU and OLRU for index access are proposed in [44]. These algorithms try to make better replacement decisions for index page buffering by the level information of index accesses. In the ILRU (Inverse LRU) policy, when a page $p_{i,j}$ at level i is accessed, it is placed at the i-th position of the LRU stack. If the depth of the LRU stack b is less than the number of levels i, the currently referenced page is placed at level b. ILRU is guaranteed to be always no worse than LRU. It can keep the low level pages in the buffer even when the buffer size is small. When the buffer gets bigger, the advantages of ILRU over LRU tend to decrease.

If the access to data pages is uniform distributed, The OLRU (optimal strategy) can be used to achieve optimal performance. For the OLRU (optimal strategy), the buffer pool is logically partitioned into L independent regions, each managed by a local LRU chain. Region j contains pages from the j-th level in the index. When the reference distribution is uniform, the OLRU is optimal under the IRM model. The more the reference density distribution deviates from uniformity, the more the ILRU is beneficial. When the reference distributions severely deviate from uniformity, ILRU tends to become more efficient than OLRU.

Priority based index caching

A priority based buffer management algorithm of indexes is proposed in [6]. Each page in the buffer is assigned a priority value which can be dynamically modified by the upper layer of DBMS to reflect its usefulness relative to other buffered pages brought into memory. An example of range access in indexes is discussed. In the range access, a range of pages are accessed through an index. After the first page is accessed, the index is backtracked to access the following pages. Several rules are used to set the priorities of pages. A buffer page is considered useful if it will be re-referenced again in the same query and useless

otherwise. This information is known because the reference pattern of the range access is known. Useful pages are assigned higher priority than useless pages. Within useless pages, higher priority is assigned to pages on lower level, because they have higher probability to be referenced by other queries. Within useful pages, higher priority is given to more recently used pages because they will be re-referenced sooner when the index tree is backtracked. However, the range access does not backtrack indexes in practice because adjacent leaf pages are linked. When the range access degrade to one data page, this algorithm becomes ILRU algorithm.

3.3.2 Predictive algorithms based on query access plan

Hot Set Model

The Hot Set model [46] is a query behavior model for DBMS that integrates advance knowledge on reference patterns into the model. It analyzes the loop behavior in the query execution of DBMS, and defines the set of pages over which there is a looping behavior as a *hot-set*. If the access plan is executed within a buffer whose size exceeds the size of the hot sets, the query processing will be fast, because the pages that are re-referenced in a loop will stay in the buffer. On the other hand, if the buffer cannot contain the hot set required, the query processing will be much slower. Because there may be more than one looping pattern, several different hot set sizes may exist in one query. Based on this hot set model, a buffer management schema is proposed in [45]. The buffer is partitioned into separated regions, one for each process. Each process is assigned the number of buffers equaling to the hot set of that process. Each region is managed by local LRU algorithm. Therefore, the hot set model is used to make load control decisions.

QLSM model, DBMIN algorithm

The Query Locality Set Model (QLSM) [10] is similar to the Hot-Set model, but is a more deliberate model. In this model, typical access behavior of DBMS is analyzed and summarized into some basic reference patterns: sequential references, random references, and hierarchical references. Different locality set size and replacement algorithms like LRU or MRU for different reference behaviors can be determined prior to the execution of the query. The DBMIN algorithm was designed based on the QLSM model. In the DBMIN algorithm, buffers are allocated and managed on a per file instance (relation or index) basis. Each file instance is given a local buffer pool to hold its locality set, whose size is determined in advance by the QLSM model. For each locality set, different local replacement algorithm is selected based on the page reference patterns of the locality set. Some basic rules are listed here:

1. *Straight sequential references* – pages are referenced and processed one after another. The locality set size is 1.
2. *Looping sequential references* – a sequential reference to a set of pages is repeated several times. The locality set size is as bigger as possible, up to the point where the entire loop can fit in buffer. MRU is the best algorithm for managing this kind of locality set.
3. *Independent random references* – pages are accessed randomly. The size of the locality set is determined by the Yao's formula [62].
4. *Looping hierarchical* – the index tree is accessed repeatedly. The pages at lower level (close to root) should be kept in the buffer. Because the fan-out of an index page is high, the probability that an index page at higher level (close to leaf nodes) is referenced again becomes very low. Therefore, the LIFO algorithm and 3-4 buffers for the locality set is recommended.

DBMIN uses a dynamic partitioning scheme, in which the total number of buffers assigned to a query may vary as files are opened and closed. The load controller is activated when a file is opened or closed. Once a file is opened, the load controller checks whether all locality sets can fit in memory. If so, the query is allowed to proceed; otherwise, it is suspended. When a file is closed, this condition is re-checked to see whether any suspended query can resume.

Some simulation experiments were performed based on some single query trace strings recorded from a DBMS. The simulation experiments focused on the effect of load control. It showed that the DBMIN can achieve a better throughput than CLOCK, FIFO, Working-Set, and Hot Set algorithm, especially when the number of concurrent queries is high. After combining a feedback load controller with the "50% rule"

[31] in which the utilization of the paging device is kept busy about 50%, the achieved throughput of the simple global algorithms is close to that of the DBMIN algorithm. However, because the DBMIN algorithm can predict the use of buffer space more accurate than the feedback load controller, it still outperforms the simple global algorithms.

DBMIN algorithm with a predictive load controller

A new load control algorithm over the DBMIN algorithm is proposed in [30]. It dynamically reallocates the buffer spaces used by queries to maximize the effectiveness of buffer allocation. The *return on memory consumption* is used to measure the *effectiveness of buffer allocation*. When a new query comes in, this algorithm attempts to get spaces from the queries that are using their buffer less efficiently. Compared with the other load control algorithms, this algorithm shows 10-15% throughput improvement when the buffer pool size is very small (70). No results are shown on big buffer sizes.

DBMIN-like algorithm with marginal gains and predictive load controller

Several flexible and adaptable buffer management techniques which are refinements for the DBMIN algorithm are proposed in [18]. Similar to the QLSM model, the reference behaviour is decomposed into several simple reference patterns which are managed by local algorithms in local buffers. The marginal gains are computed for each type of reference pattern to determine the expected number of extra page hits that would be obtained by increasing the number of allocated buffers. A query can be assigned less buffers than it needs if the marginal gain of losing some buffers is not too much. This can increase the number of concurrent queries in the system and reduce the response time of system. A new predictive load control mechanism is designed. In this load controller, a waiting query is activated with some buffers, if this admission is predicted to improve the performance of the current state of the system. A throughput predictor and an Effective Disk Utilization (EDU) predictor are defined to measure the performance of the system. A query is activated only if it will increase the system throughput or the effective disk utilization based on the load control predictors. Simulation results indicate that the adaptable algorithms are more effective and flexible than DBMIN. Some of them are capable of adapting to changing workloads.

DBMIN-like algorithm with prefetching

An approach similar to the QLSM and DBMIN algorithm is proposed in [3]. A *resident set* (RS) model, which is very similar to the QLSM model [10], is proposed. Similar to the DBMIN algorithm, a RESBAL algorithm is built based on the RS model. The difference between RESBAL and DBMIN is that prefetching is used in RESBAL when locality or sequentiality of reference has been predicted for a particular reference pattern. The RESBAL, DBMIN, HOT SET, and LRU algorithms are compared when performing a join operation. When the buffer cannot hold the inner relation, the RESBAL yields much less buffer page faults due to the prefetch policy. However, the evaluation tests performed is very simple which limited the generality of this conclusion. When no prefetching can be used or when the buffer is too big or too small, no significant advantage of the RESBAL algorithm is found than other algorithms.

In all the algorithms discussed in this subsection, the reference pattern of DBMS is utilized. Because the reference behavior is well predicted before the execution, the performance is often superior to the global algorithms. It can work well in circumstances when similar queries are executed repeatedly. However, in multi-user situations, the query plan analysis can overlap in complicated ways, and these algorithms do not take this effect into account. When the buffer is big and the hit ratio is high, the advantage of these complex algorithms over the simple global algorithms (LRU, CLOCK) is small.

4 Physical On-disk Level – Optimization of Disk I/O

Besides using an in-memory buffer, another important way of improving the performance of storage management is to improve the disk I/O efficiency. This approach can be categorized into two levels: the *software level* and the *disk level*. In the software level, the on-disk data structure and data placement strategy are

designed based on the logical structure of data. While in the disk level, the logical structure of data is not considered. Instead, only the physical nature of disk is utilized to improve the disk I/O efficiency.

Understanding the factors affect disk performance is essential in order to optimize the disk performance. The structure of disk and RAID system are introduced in § 4.1 and § 4.2 before discussing the optimization techniques. The software level approaches are discussed in § 4.3 and the disk level approaches are discussed in § 4.4.

4.1 Disk structure

A magnetic disk has a set of circular platters stacked vertically on the same axle [54, p.487,p.515]. Data are recorded on both sides of each platter. A disk head is used on each surface to read and write data. While the disk drive is working, the platters are rotating at constant speed. The concentric set of rings on the platter that can be accessed by the disk head while the platters are turning are called *tracks*. The same sized tracks on different platters' surfaces consist of a *cylinder*. In modern disks, there are thousands tracks per surface. Data are transferred to and from the disk in blocks. Typically, the block is smaller than the capacity of the track. Accordingly, data are stored in block-size regions known as *sectors*. There are typically several hundred sectors per track. For most disk drives, a fixed sector size of 512 bytes is used. Before accessing the data, the disk head should be moved to the correct track and wait for the beginning of the sector goes beneath the head. The time it takes to position the head at the track is known as *seek time*. The time it takes for the beginning of the target sector to reach the head is known as *rotational latency*. The time it takes to read or write the data is known as *transfer time*. The total disk access time can be computed as:

$$T_{access} = T_{seek} + T_{rotate} + T_{transfer} \tag{1}$$

The disk I/O bandwidth is wasted during the seek time and rotational latency because data cannot be transferred to and from the disk in this time. If small blocks (less than 16K bytes) on random locations of the disk are accessed which is typical in the OLTP workload, more than 60% disk I/O bandwidth is wasted on the the seek time and more than 30% disk I/O bandwidth is wasted on the rotational latency [33]. If requests have good locality, i.e. the next cylinder to access has higher probability to be close to the previous access cylinder, which is typical in the Fast File Systems (§ 4.3.2), the rotational latency will dominate the access time because the disk head need not go to cylinders far away. Only when transferring very large data blocks, the transfer time dominates the access time.

The seek time can be dramatically decreased by using disk scheduling algorithms (§ 4.4.1) and/or putting related data into cylinder groups (§ 4.3.2). The rotational latency can be decreased by increasing the spindle speed of disk. However, it cannot be decreased as effectively as that of seek time. Most techniques discussed in this section are aimed at decreasing seek time. Only the FFS with read and write clustering (§ 4.3.3) tries to reduce rotational latency for sequential I/O. The freeblock scheduling (§ 4.4.4) tries to utilize the wasted bandwidth on rotational latency.

A new storage technology, based on microelectromechanical systems (MEMS), has come forward promising significant performance and cost improvements relative to magnetic disks [21]. MEMS-based storage consists of thousands of small, mechanical probe tips (read/write head) that access a movable non-volatile magnetic media sled. The magnetic media is movable above an array of several thousand fixed probe tip. Each probe tip access a small rectangular area of the magnetic media. Because the positioning of probe tips is much more accurate than positioning a read/write head on a modern disk drive, the storage density of MEMS-based storage is much higher than magnetic disk drives. Similar to disks, the data on the MEMS-based storage is organized into cylinder, track, and sectors. Moreover, the access time to the data depends on the distance between the probe tips and the data on the magnetic media. The access time difference is much smaller in MEMS than in megnetic disks. Simulation results [21] show that the average access times of MEMS-based storage devices achieve access times that are 6.5X faster than a modern disk.

4.2 Redundant Array of Independent Disks (RAID)

To overcome the limitations of the performance of a single disk, the *RAID (Redundant Array of Independent Disks)* was proposed [41, 7]. RAID is a set of physical disk drives viewed by the operating system as a single logical drive. Different disks in RAID can be accessed simultaneously. All the data are distributed across

the disks. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

RAID has several levels that provide different I/O characteristics and redundancy methods. The most commonly used levels are RAID-0 (non-redundant), RAID-1 (mirrored), and RAID-5 (block-interleaved distributed parity).

In RAID-0, the data are distributed across all of the disks in the array. It provides good performance but does not contain redundancy information. In RAID-1, all data are duplicated on different disks. It provides good performance and redundancy is achieved. The problem of RAID-1 is that only half of the disk space can be utilized.

In RAID-5, a parity strip is calculated across corresponding strips on each data disk. The parity strips are stored on all disks in a round-robin manner. If the user data require N disks, N+1 disks are required for RAID-5, because the parity information will occupy the space of one disk. RAID-5 provides good read performance and reliability. However, when writing small blocks, old parity blocks may need to be read and written again after the new parity block is computed. This incurs *write penalty*. If big blocks are written into RAID-5, the new parity block can be calculated without reading the old parity block. Therefore, writing big blocks is desirable for RAID-5.

4.3 Software level approaches

The representative study in software level approaches to improve disk performance lies in file systems. The logical structure of the data (files or directories) is known. The on-disk data structures and the placement of files on disks are studied.

4.3.1 Traditional UNIX file system

In the traditional UNIX file system, an index structure *inode* is used to control the allocation of blocks [54, p.552]. The location inodes is separated from that of file data. Thus, accessing a file normally incurs a long seek from the file's inode to its data. The free blocks are managed by storing the block numbers of all free blocks in some free blocks. The disk location of the free blocks are not considered when allocated to files. After the file system has been used for some time, the free block list becomes very fragmented and the blocks of files are allocated randomly over the disk. The block size is 512 bytes, which is quite small. Under this condition, the next sequential data block often is not on the same cylinder, so seeks between 512-byte data transfers are required frequently. Less than 2% of the maximum disk throughput can be achieved [35, p. 269]. Disk seek time dominates performance in this case. The first attempt to improve the performance is to double the block size. The performance also doubled, but is still only 4%.

4.3.2 Fast File System

In order to overcome the problem of the traditional file system, The Berkeley Fast File System (FFS) was designed [36]. The on-disk data structure of FFS is the same as the traditional UNIX file system, but the placement algorithm is carefully designed to reduce disk seek time.

In FFS, the disk partition is divided into one or more areas, each of which is called a *cylinder group*. Each cylinder group comprises one or more consecutive cylinders on the disk. Some inodes are allocated for each cylinder group. When the file data are allocated, FFS attempts to allocate space close to the inodes that describe them. Data blocks for a file are tried to be allocated in the same cylinder group.

Most systems do not have a dedicated I/O processor. If many blocks are allocated sequentially on a track, after reading or writing of one block, the processor needs to process interrupt, prepare the next transfer before accessing the next block. When the request for the next block is sent to the disk, that block may just passed by the disk head, and almost a whole disk revolution need to be wait before that block is ready. The rotational delay is too expensive in this case. Therefore, sequential blocks are allocated with a gap on the disk. The size of the gap depends on the expected time for the processor to service an interrupt and to schedule a new disk transfer. Blocks allocated with this gap are called *rotationally optimal* blocks.

A large block size (say, 8K bytes)is used to increase disk transfer efficiency. To reduce internal fragmentation caused by large block size, the last block of a file can occupy only part of the block (fragment), and the unused fragments can be allocated to other files.

FFS can read and write individual files at roughly 50% of the disk bandwidth [35, p.285]. It cannot achieve higher throughput because of the gap between rotationally optimal allocated blocks is normally one block.

4.3.3 Fast File System with I/O clustering

In order to reduce the rotational latency of FFS when accessing sequential blocks of a file, the I/O clustering was designed [37]. Sequential dirty buffers are accumulated in a buffer with up to 64K bytes space. A write request is sent until the buffer is full or no more sequential data writes are needed. Therefore, no gap between blocks are needed and the rotational latency is minimized. When reading files, if the file system discovers that a file is being read sequentially, it allocates a buffer big enough to hold the contiguous set of blocks and read these blocks in a single I/O request.

The FFS with I/O clustering can utilize almost all available disk bandwidth when accessing individual files [35, p.285].

4.3.4 Log-structured File System

Both FFS and FFS with I/O clustering are optimized for accessing sequential blocks for the same file. In a multi-programming system, different users access different files simultaneously. Therefore, the disk head may need to move between different locations of the disk. Because many reads can be absorbed by the main memory buffer, the performance problem to be solved in modern file systems is that of writing a large volume of data, from multiple files, to the disk.

The Log-structured File System (LFS) [40, 43, 48], as proposed by Ousterhout and Douglis, attempted to address this problem. In LFS, all file system data are stored in a single contiguous log. The LFS is optimized for writing, and no seek is required between writes, regardless of the file to which the writes belong. Because it writes big blocks sequentially, it is suitable when working with RAID-5 to avoid write penalty.

The LFS uses almost the same data structure (inodes and directories) as FFS. Where the LFS differs from the FFS is in the layout of the inode, directory and file data blocks on disk. The underlying structure of the LFS is a sequential, append-only *log*. The log is statically partitioned into fixed-sized contiguous segments which are generally 0.5 to 1M bytes (this should be larger on RAID systems). All writes to the disk are appended to the logical end of the log. Because the disk space is not infinite, portions of the log that have already been written must be made available periodically for reuse. This process is called *cleaning*, and the utility that performs this reclamation is called the *cleaner*. The cleaner runs in user mode, and flexible cleaning policies can be used in the cleaner. The logical order of the log is not fixed, and the log should be viewed as a linked list of segments, with segments being periodically cleaned, detached from their current position in the log, and re-attached after the end of the log. The segments are partitioned into one or more partial segments so that the LFS can write data less than a segment. A partial segment is a result of a single write operation to the disk.

In ideal operation, the LFS accumulates dirty blocks in memory. When enough blocks have been accumulated to fill a segment, or when they have to be written (because of a *fsync* or *sync* call, or because of closing a file), they are written to the disk in a single, contiguous I/O operation. Because the location of data blocks are changed when they are written, the inodes have to be changed accordingly. These modified inodes are written into the segment at the same time as the data. As a result of this design, inodes are no longer in fixed locations on the disk, and the LFS uses an additional data structure called *inode map*, which maps inode numbers to the current disk addresses of the blocks containing them.

The inode map is stored in the index file (*ifile)*, which is maintained as a regular, read-only file visible in the file system. Because the ifile's inode and data blocks are themselves written to new locations each time that they are written, the superblock, which is on a fixed location on the disk, is used to find the ifile's inode. In addition to the inode map, the ifile contains the number of clean and dirty segments and the segment-usage information. This information is shared between the kernel and the cleaner.

When reading data from the LFS, the steps are very similar to that in FFS: get the inode address and read the inode information from the inode number, get the physical disk address from the inode based on the logical disk number, and read data from the physical disk address. The only difference is that because the inode address cannot be obtained directly from the inode number, a lookup in the inode map is needed

before the above steps. Normally, the most frequently used part of the inode map is cached in memory. Therefore, the LFS has similar overhead as FFS when reading files.

When a dirty block must be flushed to the disk, the LFS gathers all the dirty blocks for writing. The dirty blocks are sorted by file and logical block number, and then are assigned disk addresses. Their associated meta-data blocks (inodes and indirect blocks) are updated to reflect the new disk addresses. The dirty blocks and the meta-data blocks are formatted into one or more partial segments. Each partial segment is written to the disk sequentially in one write request.

In LFS, cleaning must be done periodically to make new segments available for writing. Cleaner is a process running in user mode. It reads the ifile and computes the *cleaning cost* and *benefit* for cleaning segments. The segment with the best *benefit-to-cost* ratio is selected for cleaning. The cleaner reads the segment into memory, and only writes the live data of the segment out. The dead data (data that had been written again) will be discarded. This segment will be marked as clean after the cleaning.

The design of the cleaning algorithm is the most challenging part of the LFS because of its impact on performance. Cleaning must be done before the file system fills up. If the cleaning is done when the file system is lightly utilized, it is called *background cleaning*; otherwise, it is called *on-demand cleaning*. The background cleaning does not adversely affect the normal file system activity, while on-demand cleaning can result in big performance degradation. The cleaning algorithms have been studied in some research works [43, 2, 34]. A better cleaning cost and benefit model is used in [34] to select optimal segment size and cleaning method. A simple heuristic is proposed in [2] to determine idle period and attempts perform all the cleaning work in background. Therefore, the cleaning won't affect the user response time in most cases.

In previous research works [50, 48, 51, 49], on-demand cleaning is found to have adverse effect on the performance of system, especially in OLTP environments. Peak or near-peak transaction rates must be sustained at all times in OLTP workload. Therefore, there may not be enough idle time for the system to do the cleaning. Moreover, because the updated data are randomly distributed on the disk in the OLTP workload, most segments are fairly full before cleaning. 60-80% writes and 31% of all blocks transferred are generated by the cleaner [48]. The cleaning overhead is high that a performance degradation of about 35 to 50 percent was observed under these workloads. After this performance degradation, the LFS provides comparable but sometimes worse performance to FFS with I/O clustering.

The *freeblock scheduling* method [33] which will be discussed in § 4.4.4 can let LFS do the cleaning without affecting foreground response times even in never-idle systems.

In the MEMS-based storage devices, the access time difference when accessing data on different data is smaller than that on the traditional disk drives [21]. Therefore, the data organization will have less effect on the performance of the system but still useful.

4.4 Disk level approaches

4.4.1 Disk scheduling policies

In a multiprogramming environment, typically there are many requests for each disk. They are maintained in a queue by the operating system or the disk controller. If the items are selected from random order, the disk head has to visit different tracks randomly, giving the worst possible performance. The disk scheduling policies can be used to improve disk performance [54].

If the items are selected in a *first-in-first-out (FIFO)* sequence, it is fair to all requests but the performance may still suffer for the same reason. Some disk scheduling policies other than FIFO can be used to reduce the disk seek time.

The *Shortest Service Time First (SSTF)* policy is to select the disk I/O request that requires the least movement of the disk head from its current position. This algorithm has better performance than FIFO. The problem of SSTF is that it may always select requests close to the current disk head position and starve other requests.

The *SCAN* algorithm can prevent this kind of starvation. In the SCAN policy, the disk head moves in one direction only, satisfying all waiting requests, until no more requests are in that direction. The head then moves in reversed direction and again picking up all requests in order. The SCAN policy has similar performance as SSTF but the starvation is less likely to happen. This policy favors requests that are for tracks nearest to both innermost and outermost tracks. The *C-SCAN (circular SCAN)* policy can avoid this

problem of SCAN. In C-SCAN, requests are fulfilled only when the head is moving in one direction. If no more requests are on that direction, the head returns to the opposite end of the disk and scans again.

Both SCAN and C-SCAN may favor the latest arriving jobs and starvation may happen if requests for the same track keep coming. The *N-step-SCAN* can address this problem. The N-step-SCAN segments the disk request queue into sub-queues of length N. Only one sub-queues can be processed at a time. All new coming requests are added to some other queues.

In the MEMS-based storage devices, because the access times depend on the relative locations and motions of the movable media and the desired data, appropriate request scheduling can be used to reduce access times. Simulation results in [21] show that most of the algorithms and insights from previous disk scheduling research will also be relevant to systems with MEMS-based storage devices.

4.4.2 HP AutoRAID hierarchical storage system

Different RAID levels have different characteristics. It is hard to configure and hard to change the configuration. The HP AutoRAID hierarchical storage system [60] provides a solution to this problem. A two-level storage hierarchy is implemented in a disk array. The higher level uses mirrored (RAID-1) to store active updated data, which provides redundancy and good performance. The lower level uses RAID-5 (distributed parity) to provide storage cost for inactive data, at somewhat lower write performance.

Because data may migrate between the two storage hierarchy levels, direct physical addresses cannot be used. The logical disk addresses are sent to the AutoRAID disk controller. They are converted to physical addresses transparently. The AutoRAID presents one or more *logical units* to its hosts. Each logical unit is treated as a virtual device inside the array controller.

Some mapping structures are defined to translate logical addresses in a logical unit to physical addresses. Each disk is partitioned into large objects called *Physical EXtents (PEXes)*. PEXes are 1MB in size. PEXes on different disks (at least three) can be combined into a *Physical Extent Group (PEG)*. A PEG may be assigned to either the RAID-1 or the RAID-5 storage class. The logical space provided by AutoRAID is divided into 64KB units called *Relocation Blocks (RBs)*. RBs are allocated in PEGs. RBs are basic units of migration in the system. A *virtual device table* is defined for each logical unit. It lists the RBs and pointers to the PEGs in which they reside. A PEG table is defined for each PEG. It holds list of RBs in the PEG and list of PEXes used to store them. A PEX table is defined for each physical disk drive.

When there is enough space in the array, all data are stored in mirrored storage. When the free space of the array is less than a low-water mark, the migration begins to migrate data from mirrored storage to RAID-5. The data selected for migration are based on an approximate Least Recently Written algorithm. Migrations are performed in the background until the free space in the mirrored storage class exceeds a high-water mark that is chosen to allow the system to handle a burst of incoming data.

When data in RAID-5 storage are updated, they are promoted to mirrored storage because they are considered active data. When the RAID-5 storage is written, an approach similar to LFS is used. The RAID-5 storage is written as an append only log, and all new writes happens at the end of the log. Cleaning are needed periodically to clean the holes in the RAID-5 storage.

4.4.3 Virtual log for a programmable disk

In order to minimize the latency of small synchronous writes to disks, a *virtual log* approach is proposed [59]. This approach is designed for a single disk. The basic idea is to write to a disk location that is close to the current head location.

When some data are modified, they may be written on different locations of the disk. This non-overwrite idea is similar to the approach of LFS. The difference is that new data can be written to any location of the disk, thus constructs a virtual log. An analytical model shows that even when the disk has little free space, the average time to locate a free sector is still significantly smaller than a half-rotation delay. This result is also consistent with simulation results.

The physical address of a disk block can change when it is updated. Therefore, data in the virtual log are not necessarily physically contiguous. Logical addresses are used for the virtual log. An *indirection map* can convert logical disk addresses to physical addresses. This indirection map is updated whenever new data are written. The new updated part of the map is written close to the head position. A backward pointer to

the previous part of the map is maintained. All parts of the indirection map constructs a single-linked list. All data of the indirection map can be retrieved from the last part of the indirection map.

Since the virtual log can utilize any free space on the disk, no cleaning is required. This is an advantage to the LFS approach. However, it has the same disadvantage as LFS that randomly updated data may have bad sequential read performance.

4.4.4 Extracting free bandwidth from busy disk drives

When a disk drive is serving foreground applications, there are rotational latencies between requests. This potential free bandwidth is wasted. The *freeblock scheduling* proposed in [33] can provide these bandwidth to background applications without affecting foreground response time.

The freeblock scheduling predicts how much rotational latency will occur before the next foreground data transfer, squeeze some additional transfers into that time, and still seek to the destination track in time for the foreground transfer. The additional transfers may be on the current or destination track or another track.

For random workload, 27-36% of disk head usage can be attributed to rotational latency. This amount of potential free bandwidth is available for freeblock scheduling. For large random requests (e.g., 256KB), the potential free bandwidth is still 15%. If requests have access locality (e.g., accessing requests in a cylinder group repeatedly), the potential free bandwidth increases.

The background application should register many disk requests to the system with freeblock scheduling. When one of the request is satisfied, the background application is notified through a callback function with the requested data. The background application can then process the data received.

This approach can be used in applications that are low priority, have large sets of desired blocks and no particular order of access. The cleaning in LFS satisfies these requirements. The cost of cleaning can be close to zero when applying the freeblock scheduling. The logic of cleaning is to examine the segments, read in live blocks and write them out. The live blocks can be read in any order via freeblock scheduling and moved into new locations at once. To increase the chance that the live block requested can be read in quickly, the cleaner can select several segments and clean them in parallel. This cleaning activity is not entirely for free because the live data need to be written out to the log with the foreground requests. When all live blocks are cleaned from a segment on the disk, that segment becomes available for future use. The on-demand cleaning only kicks in when the background cleaner cannot generate free segments fast enough.

One experiment showed that on a busy LFS system dominated by small-file activities, the cleaner using freeblock scheduling can perform all cleaning for almost free when the disk utilization is less than 93%. Another experiment was to study data mining via freeblock scheduling on a OLTP system. The freeblock scheduling increases useful disk head utilization from 3% to 24%, and it allows the data mining application to complete over 47 full scans per day on the 9GB disk with no effect on foreground OLTP performance.

The freeblock algorithm needs to know detailed knowledge of many disk performance attributes, including layout algorithms and time-dependent mechanical positioning overheads. This may require that freeblock scheduling decisions be made by disk firmware.

4.4.5 Logical disk

The *Logical Disk (LD)* [14] defines a new interface to disk storage. It separates file management and disk management by using logical block numbers and block lists. Although modern disk systems provide logical interfaces to the upper layer, some physical details of the disk is still needed for many systems. For example, the cylinder organization and rotational latency are important parameters for the FFS file system [36]. This logical disk interface is easy to support multiple file systems.

The basic abstractions provided by LD are *logical block numbers, block lists, atomic recovery units*, and *multiple block sizes*. No cylinder, track and sector information are visible from the LD interface. The data placement are managed transparently by LD. Because the logical block numbers are used, LD can change the physical address of blocks without notifying the file system. The file system can give hints to LD to guide effective placement of data. The file system can put related blocks into ordered block lists. LD will try to cluster blocks on the same list together on disk.

After this LD interface has been implemented, the file system can be easily changed to use LD. Different implementations of LD can be applied without changing the file system. A Log-structured LD (*LLD)*

interface was implemented and tested in MINIX. It shows similar performance characteristics as the Sprite LFS.

5 Logical Level – Smarter Use of Memory

In the current buffer pool management of DBMS, the database pages from the disks are stored directly in the buffer pool. If the logical information of the pages are utilized and other database entities are stored in the buffer pool, the buffer space may be utilized more efficiently. These database entities can be index pointers, table tuples, or other logical data entities.

5.1 Index pages

In RDBMS, the index is often structured as a B^+-tree. When the index is searched, the index tree should be traversed from the root node to the leaf node. The reference to these pages consists of an "access path". Only when all pages on the access path are in the buffer pool, no page faults will occur.

A buffer management strategy for B^+-tree indexes (GHOST) is proposed in [20]. The used "paths" of the B^+-tree is buffered in part of the buffer by the ILRU algorithm, while the remainder is devoted to maintaining a Splay-tree with pointers to leaf pages containing frequently used leaf pages. The Splay-tree is a self-adjusting binary search tree. The Splay-tree does not try to maintain a balanced tree, but try to put frequently referenced nodes close to the root. Whenever a node of the Splay-tree is referenced, this node is moved to the root through the algorithms used in the AVL tree. In the long run, popular nodes of the Splay-tree are close to the root. Therefore, the reference to these nodes requires less search path and the average access time is bounded by $O(\log n)$. Through the Splay-tree, the buffer manager can maintain pointers to leaf nodes long after the paths leading to the leaf nodes have been replaced, thus maintaining "ghost" paths to the nodes. It normally requires less space for storing the pointers to leaf nodes than storing the access path from the index root to the data page. Because most frequently used leaf pages are stored in the Splay-tree, many access may find the pointer to data page from the Splay-tree and only need one I/O to read the data page. If the page cannot be found in the Splay-tree, the index tree will be searched directly with the help of the ILRU buffer. Popular index pages is likely to be obtained from the ILRU buffer without additional disk I/O.

Three different types of queries are tested in the experimental study: uniform, skewed-unclustered, and skewed-clustered. 50% of the available memory is devoted to the Splay tree and the remaining is reserved for buffering index nodes under the ILRU strategy. The test results showed that the GHOST shows higher hit ratio and lower I/O cost when the buffer size is big for all workloads. For the non-uniformed query workloads, the gain is about 25%-30%. For the range queries, the GHOST scheme also shows up to 20% lower I/O count. By varying the amount of memory used by the Splay tree of the GHOST scheme, it is found that when the total available memory is small, allocating more memory to the Splay tree is better. On the other hand, with a large buffer size, the performance hits the performance limit rapidly even with small allocations to the Splay tree.

5.2 Data pages

Typically, a data page contains many tuples of a table. If only some of the tuples in a page is hot, caching the whole page wastes buffer space. Studies in [32] show that in the TPC-C benchmark, if hot tuples are clustered into same pages (which is allowed by the benchmark), the performance can improve a lot. However, because the hot set of tuples may change over time in real workloads, the static clustering is not much meaningful in the real world.

If the tuples instead of pages are cached in the buffer pool, the wastes of buffer pool space can be decreased. The experiments in [13] shows that when the tuples are not clustered well, tuple caching can yield much shorter response time than page caching. However, because the overhead of tuple-caching is much higher than page-caching, pure tuple-caching may not be beneficial.

[5] presented a hybrid algorithm *HAC* for caching management in the client cache in a client/server structured OODBMS. HAC is an adaptive combination of object-caching and page-caching. When the clustering of objects is good, the algorithm performs like a page-caching algorithm which decreases overhead.

When the clustering of objects is bad, the algorithm performs like a object-caching algorithm, which decreases miss ratio.

The object header is 32 bits which consists of a 22-bit *pid* and a 9-bit *oid*. The pid is the page number the object resides. The oid identifies the object within its page. Each page contains an *offset table* that maps oid to 16-bit offsets within the page. The existence of object table allows the compaction of the object within the page.

The client cache is partitioned into a set of page-sized frames. If many objects in the frame are hot objects, this frame is intact; if only a few objects in the frame are hot, this frame is compacted and only hot objects are retained. When objects are cached in the client cache, their object headers are translated to pointers pointing the memory addresses of the objects via an *indirect pointer swizzling* table. This translation is only done when this object is actually referenced. This lazy translation method avoids much overhead of translating many never used object headers. The use of indirection table allows HAC to move and evict objects easily.

The object usage and frame usage information are maintained for replacement decisions. The object usage contains both recency and frequency of accesses. The frame usage is computed from the object usage of the objects it contains. The less the frame usage value, the less the usefulness of the frame. Because computing the frame usage and selecting a frame with the lowest usage are expensive, they are not computed frequently. A *candidates set* of frames is maintained. HAC only selects the victim frames from the candidates set. If the time a frame in the candidates set is longer than a threshold, its frame usage information is likely to be out of date and this frame is removed from the candidates set. A variant of the CLOCK algorithm is used to select frames into the candidates set. The cache is organized as a circular queue of frames. A *primary scan pointer* and *N secondary scan pointers* are used to scan the queue. All the pointers are equidistant. Each time when a frame needs to be freed, the primary scan pointer scans S frames, computes the frame usage, and adds them to the candidates set if necessary. At the same time, each secondary scan pointer also scans S frames, find frames with a large number of uninstalled objects and add them into the candidates set if necessary. If there are E frames in the candidates set, $O(\log E)$ time is needed to find the lowest-usage frame.

Performance experiments showed that HAC achieves lower miss rate than the best page-caching and dual-buffering systems in the literature. Although this work is for the client cache in a client/server structured OODBMS, its principles can be applied to RDBMS.

Most queries in the OLTP workload are simple select queries. The result of a query is one tuple. Tuple-caching is suitable in this workload. In the OLAP workload, queries are complex, which include many scan operations on tuples. Typically all tuples in a page are referenced when a page is accessed. Therefore, there is no need to do tuple-caching in OLAP workload.

6 Conclusion

This survey gives a comprehensive overview to the storage management of DBMS. The storage management is divided into three levels (logical , physical in-memory, and physical on-disk) and discussed separately.

The physical in-memory level has the most impact on the performance of the performance of DBMS and was studied intensively for decades. The variations of LRU and CLOCK algorithms are popularly used in real systems because they are easy to implement, have small overheads, and good performance. Many sophisticated replacement algorithms were designed. Some algorithms are designed for the special reference patterns of DBMS queries. These algorithms provide lower miss ratio than LRU and CLOCK especially when the buffer size is small. The miss ratio difference between the algorithms becomes small when the buffer size is big, which is the case in modern DBMS systems.

Many algorithms of the physical on-disk level used in DBMS are borrowed or derived from file systems or storage systems. The physical on-disk level is important to the performance of DBMS because this level is typically the bottleneck. The buffer pool can absorb reads but not writes effectively when it is big. Therefore, improving the performance of writes becomes more and more important. The Log-structured File System provides excellent potential write performance especially for systems using RAID. However, the cleaning cost eliminates most advantages of LFS. Several approaches to decrease the cleaning cost are discussed. Many of these approaches are only verified by simulation. This log-structured approach was mainly studied

in file system context. Further research work is needed to test whether this idea can improve the efficiency of storage management of DBMS.

The logical level of the storage management has few attention from researchers. By caching objects/tuples of the database, the memory can be utilized better than caching disk pages. This approach has been used in OODBMS but not in relational DBMS. The GHOST algorithm stores a small index search tree in memory to occupy smaller space than storing index pages. Both approaches may decrease the buffer pool miss ratio.

There are many approaches in each level of the storage management of DBMS. Studying only one level may not yield significant boost to the performance. All these three levels should be studied and integrated to further improve the performance of storage management of DBMS.

References

[1] M. F. Arlitt. A performance study of internet web servers. Master's thesis, University of Saskatchewan, 1996.

[2] T. Blackwell, J. Harris, and M. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference*, pages 277–288, New Orleans, LA, USA, 16–20 1995.

[3] F. F. Cai, M. E. C. Hull, and D. A. Bell. Buffer management for high performance database systems. In *Proceedings of the High-Performance Computing on the Information Superhighway (HPC-Asia '97)*, pages 633 – 638, Seoul, Korea, April – May 1997.

[4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling. *ACM Transactions on Computer Systems*, 14(4):311–343, 1996.

[5] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid Adaptive Caching for distributed storage systems. *Operating Systems Review*, 31(5):238–251, December 1997.

[6] C. Y. Chan, B. C. Ooi, and H. Lu. Extensible buffer management of indexes. In *Proceedings of the* 18^{th} *International Conference on Very Large Data Bases*, pages 444–454, Vancouver, Canada, August 1992.

[7] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.

[8] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. An adaptive block management scheme using on-line detection of block reference patterns. In *Proceedings of the 1998 International Workshop on Multimedia Database Management Systems (IW-MMDBMS '98)*, pages 172–179, Dayton, Ohio, August 1998.

[9] J. Choi, S. H. Noh, S. L. Min, and Y. Cho. Towards application/file-level characterization of block references: A case for fine-grained buffer management. In *Proceedings of the SIGMETRICS 2000 International Conference on Measurements and Modeling of Computer Systems*, pages 286–295, Santa Clara, California, June 2000.

[10] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the* 11^{th} *International Conference on Very Large Data Bases (VLDB'85)*, pages 174–188, Stockholm, Sweden, August 1985.

[11] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[12] D. Comer. The ubiquitous B-Tree. *Computing Surveys*, 11(2):121–137, 1979.

[13] S. Dar, M. J. Franklin, B. T. Jónsson, D. S., and M. Tan. Semantic data caching and replacement. In *Proceedings of* 22^{th} *International Conference on Very Large Data Bases (VLDB'96)*, pages 330 – 341, Mumbai (Bombay), India, September 1996.

[14] W. de Jonge, M. Kaashoek, and W. Hsieh. The logical disk: a new approach to improving file systems. In *Proceedings of the* 14^{th} *ACM Symposium on Operating System Principles (SOSP 1993)*, pages 15–28, Asheville, North Carolina, December 1993.

[15] R. Dube. A comparison of the memory management sub-system in FreeBSD and Linux. Technical report, University of Maryland, September 1998.

[16] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Transactions on Database Systems*, 9(4):560–595, December 1984.

[17] W. Effelsberg and M. E. S. Loomis. Logical, internal, and physical reference behavior in CODASYL database systems. *ACM Transactions on Database Systems*, 9(2):187–213, June 1984.

[18] C. Faloutsos, R. T. Ng, and T. K. Sellis. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers*, 44(4):546–560, April 1995.

[19] Gideon Glass and Pei Cao. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 115–126, Seattle, Washington, June 1997.

[20] C. H. Goh, B. C. Ooi, D. Sim, and K.-L. Tan. GHOST: Fine granularity buffering of indexes. In *Proceedings of* 25^{th} *International Conference on Very Large Data Bases (VLDB'99)*, pages 339–350, Edinburgh, Scotland, UK, September 1999.

[21] J. L. Griffin, S. W. Schlosser, G. R. Ganger, and D. F. Nagle. Modeling and performance of MEMS-based storage devices. In *Proceedings of the 2000 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 56–65, Santa Clara, CA, USA, June 18 - 21 2000.

[22] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. G. Lindsay, H. Pirahesh, M. J. Carey, and E. J. Shekita. Starburst mid-flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):143–160, March 1990.

[23] W. W. Hsu, A. J. Smith, and H. C. Young. Analysis of the characteristics of production database workloads and comparison with the TPC benchmarks. Technical Report CSD-99-1070, University of California, Berkeley, November 1999.

[24] W. W. Hsu, A. J. Smith, and H. C. Young. I/O reference behavior of production database workloads and the TPC benchmarks - an analysis at the logical level. Technical Report CSD-99-1071, University of California, Berkeley, November 1999.

[25] M. V. Devarakonda J. T. Robinson. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134–142, Boulder, Colorado, USA, May 22-25 1990.

[26] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of* 20^{th} *International Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, Santiago de Chile, Chile, September 1994.

[27] J. P. Kearns and S. DeFazio. Locality of reference in hierarchical database systems. *IEEE Transactions on Software Engineering*, SE-9(2):128–134, March 1983.

[28] J. P. Kearns and S. Defazio. Diversity in database reference behavior. In *Proceedings of the 1989 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 11 – 19, Berkeley, California, May 1989.

[29] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *Proceedings of the SIGMETRICS 1999 International Conference on Measurements and Modeling of Computer Systems*, pages 134–143, Atlanta, Georgia, May 1999.

[30] S. Lee and S. Lee. Applying dynamic buffer allocation to predictive load control. In *Proceedings of the* 13^{th} *International Conference on Technology and Education (ICTE 1995)*, pages 150 – 155, Orlando, Florida, March 1995.

[31] J. Leroudier and D. Potier. Principles of optimality for multi-programming. In *Proceedings of the International Symposium on Computer Performance Modeling, Measurement, and Evaluation*, pages 211 – 218, Cambridge, Massachusetts, March 1976.

[32] S. T. Leutenegger and D. M. Dias. A modeling study of the TPC-C benchmark. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 22 – 31, Washington, D.C., May 1993.

[33] C. Lumb, J. Schindler, G. Ganger, D. Nagle, and E. Riedel. Towards higher disk head utilization: Extracting free bandwidth from busy disk drives. In *Proceedings of the* 4^{th} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Deigo, CA, October 2000.

[34] J. N. Matthews, D. R., A. M. Costello, R. Y. Wang, and T. E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *Operating Systems Review*, 31(5):238–251, October 1997.

[35] M. K. McKusick. *The design and implementation of the 4.4BSD operating system.* Addison-Wesley Longman, Reading, MA, 1996.

[36] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, 1984.

[37] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the Winter USENIX 1991*, pages 33–44, Dallas, Texas, January 1991.

[38] V. F. Nicola, A. Dan, and D. M. Dias. Analysis of the generalized clock buffer replacement scheme for database transaction processing. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE'92 International Conference on Measurement and Modeling of Computer Systems*, pages 35–46, Newport, Rhode Island, June 1992.

[39] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 297–306, May 1993.

[40] J. K. Ousterhout and F. Douglis. Beating the I/O bottleneck: A case for Log-structured file systems. *Operating Systems Review*, 23(1):11–28, January 1989.

[41] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, pages 109–116, Chicago, Illinois, June 1988.

[42] J. Rodriguez-Rosell. Empirical data reference behavior in data base systems. *Computer*, 9(11):9 – 13, November 1976.

[43] Mendel Rosenblum. *The Design and Implementation of a Log-Structured File System.* PhD thesis, University of California, Berkeley, June 1992.

[44] G. M. Sacco. Index access with a finite buffer. In *Proceedings of the* 13^{th} *International Conference on Very Large Data Bases (VLDB'87)*, pages 301–309, Brighton, England, September 1987.

[45] G. M. Sacco and M. Schkolnick. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the* 8^{th} *International Conference on Very Large Data Bases (VLDB'82)*, pages 257–262, Mexico City, Mexico, September 1982.

[46] G. M. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Transactions on Database Systems*, 11(4):473–498, December 1986.

[47] H. Schoening. The ADABAS buffer pool manager. In *Proceedings of the* 24^{th} *International Conference on Very Large Databases*, pages 675–679, New York City, New York, August 1998.

[48] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Technical Conference*, pages 307–326, San Diego, CA, USA, 25–29 1993.

[49] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File system logging versus clustering: A performance comparison. In *Proceedings of the USENIX 1995 Technical Conference*, pages 249–264, New Orleans, LA, 16–20 1995.

[50] M. I. Seltzer. *File System Performance and Transaction Support*. PhD thesis, University of California at Berkeley, 1992.

[51] M. I. Seltzer. Transaction support in a log-structured file system. In *Proceedings of the* 9^{th} *International Conference on Data Engineering*, pages 503–510, Vienna, Austria, 19–23 1993.

[52] Y. Smaragdakis, S. Kaplan, and P. Wilson. EELRU: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS 1999 International Conference on Measurements and Modeling of Computer Systems*, pages 122–133, Atlanta, Georgia, May 1999.

[53] A. J. Smith. Sequentiality and prefetching in database systems. *ACM Transactions on Database Systems*, 3(3):223–247, September 1978.

[54] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4^{th} edition, 2000.

[55] H. S. Stone, J. Turek, and J. L. Wolf. Optimal partitioning of cache memory. *IEEE Transactions on Computers*, 41(9):1054–1068, 1992.

[56] J. Z. Teng and R. A. Gumaer. Managing IBM database 2 buffers to maximize performance. *IBM Systems Journal*, 23(2):211–218, 1984.

[57] D. Thiébaut and H. S. Stone. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665–676, 1992.

[58] A. I. Verkamo. Empirical results on locality in database referencing. In *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 49 – 58, Austin, Texas, August 1985.

[59] R. Y. Wang, T. E. Anderson, and D. A. Patterson. Virtual log based file systems for a programmable disk. In *Proceedings of the* 3^{rd} *USENIX Symposium on Operating Systems Design and Implementation (OSDI 1999)*, pages 29–43, New Orleans, Louisiana, February 1999.

[60] J. Wilkes, R. Golding, C. Staelin, and T. Sullivan. The HP AutoRAID hierarchical storage system. In *Proceedings of the* 15^{th} *ACM Symposium on Operating Systems Principles*, pages 96 – 108, Copper Mountain, Colorado, 1995.

[61] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network fileservers. In *Proceedings of the* 10^{th} *International Conference on Distributed Computing Systems(ICDCS 1990)*, pages 212–219, Paris, France, May 1990.

[62] S. B. Yao. Approximating block accesses in database organization. *Communications of the ACM*, 20(4):260–261, April 1977.