1 Buffer Management in DBMS versus Disk Cache

Modern disks often have their own main memory caches, from 8 to 32 MB nowadays, and use this to prefetch pages. The rationale for this technique is the empirical observation that, if a disk page is requested by some (not necessarily database!) application, 80% of the time the next page is requested as well. So the disk gambles by reading ahead.

1. Give a nontechnical reason that a DBMS may not want to rely on prefetching controlled by the disk.
2. Explain the impact on the disk's cache of several queries running concurrently, each scanning a different file.
3. Is this problem addressed by the DBMS buffer manager prefetching pages? Explain.
4. Modern disks support *segmented caches*, with about four to six segments, each of which is used to cache pages from a different file. Does this technique help, with respect to the preceding problem?

Solution Sketch

The disk cache, or disk buffer, acts as a buffer between the computer and the disk and it is controlled by a micro-controller in the disk. One might not want to rely on prefetching controlled by the disk for the same reason as not to rely on prefetching controlled by the OS, i.e., the DBMS can often predict the page reference patterns much more accurately than is typical in a disk cache.

The disk cache in principle supports only one file scan efficiently. In case of several parallel scans, each subsequent prefetching would overwrite the total cache, making it useless. On the other hand, the prefetching done by the DBMS buffer manager won't overwrite pages still being used.

To address the preceding problem, disk cache is divided into several *cache segments*. Each cache segment is a unit of prefetching, allowing a speed up in performance of multiple scans, as long as the number of concurrent scans is not higher than the number of cache segments.

2 SSDs vs. Hard Disks

One major disadvantage of Solid-state drives is their limited write cycles. This problem is reduced by spreading writes over the entire device (so-called wear leveling), rather than rewriting files in place. Assuming the following specs of a SSD disk:

Capacity:	64GB
Write endurance rating:	1 Mio. cycles
Sustained write speed:	50MB/sec

1. Assuming a perfect wear leveling, how long have you got before the disk is trashed if you perform non-stop writing operations?
2. Discuss the differences between SSDs and hard disks in terms of read/write operations an sequential/random access.
3. Give (at least) one example of a DB operation where SSDs outperform hard disks by a large amount.

Solution Sketch

1. Wear-leveling is a technique for arranging data so that erasures and re-writes are distributed evenly across the medium, therefore prolonging its lifetime. A 100% wear level is hard to get in practice, but for simplicity we assume this is the case in this exercise. If the write endurance rating is 1 Mio. cycles and assuming perfect wear level, it means we need to fill the disk 1 million times to get to the write endurance limit. Given the size of the disk we can then write

$$64GB \times 1MB = 64 \times 10^{15} \text{ bytes.}$$

   Writing at a speed of 50M bytes/sec, it would take us

$$(64 \times 10^{15})/50M = 1280 \text{ Mio. seconds.}$$

   Which is approximately 40,59 years (by that time you probably bought a newer disk already :)).

2. Reads are comparable between SSDs and HDs, but writes are much more expensive in SSDs, because memory cells must first be erased completely before new bits of data can be written. Regarding sequential vs. random accesses, in hard disks, due to the delays created by seeking time and head switching, random accesses are much more expensive and should be avoided as much as possible. In SSDs, however, since no mechanical parts are involved in the accesses, the difference between sequential and random accesses is not that big (but sequential accesses are still cheaper).

3. Basically, every operation that involves a big number of read-only random accesses will have a better performance in SSDs. One example is search engines, that usually have a big index of terms, and query processing involves accessing the documents lists from each query term.

3 Horizontal vs. Vertical Partitioning

Consider a table with 10 attributes and we want to perform an operation that involves only 2 of these attributes. For each of the following storing models: NSM, DSM, PAX, and Fractured Mirrors, answer:

1. What percentage of the data is loaded into the memory and the L1 cache, respectively?
2. If you want to write one row into the table, how many I/O operations are needed?

Solution Sketch

Short summary:

- N-ary Storage Model (NSM): All attributes of a record are stored consecutively in one page as <record_id, attribute_set>. Enables good performance for full-access queries however it does not perform well for partial access queries;
- Decomposition Storage Model (DSM): Stores each attribute of a record in a different page as <record_id, attribute> format. Good for partial-access queries but on the other hand for full-access queries that requires multiple attributes from the same record, join operations must be applied between different pages;
- Partition Attributes Across Model (PAX): Compared to NSM improves cache performance. Stores all attributes of a record in the same page like NSM, but reorganizes the records in a page. Given a record that consists of n attributes, PAX partitions each page into n mini pages and puts each attribute of the record into different minipages such as putting n^{th} attribute in the n^{th} minipage;

- Fractured Mirrors: Store data in NSM format as well as DSM format to have small query execution time under both partial and full access queries.

1.

	NSM	DSM	PAX	Fractured Mirrors
Memory	100%	20%	100%	20% (same as in DSM)
L1 cache	100%	20%	20%	20% (same as in DSM)

2.

	NSM	DSM	PAX	Fractured Mirrors
# of I/O's	1	10	1	11

4 Compression

Given an algorithm that requires 50 CPU clocks per byte to decompress a given data. If we have a single 2GHz CPU machine and the I/O operations can be done at a speed of 100MB/s, what is the minimum compression ratio that justifies the use of the compression?

Solution Sketch

Since CPU and I/O operation can be done in parallel, the compression ratio has to satisfy the following inequation:

$$50 \text{ clocks/byte} \frac{x}{CR} \frac{1}{2\text{GHz}} \leq \frac{x}{100\text{MB/s}}$$

where x is the size of the data and CR is the compression ratio. The rationale behind the inequation is that the time that it takes from the CPU to handle the amount compressed data should not be higher than the time that it takes to read/write the uncompressed data. After doing the math we find out that the compression rate should be at least equal to 2.5 to justify the use of the compression algorithm.