The VLDB Journal (2016) 25:673–694
DOI 10.1007/s00778-015-0414-1

SPECIAL ISSUE PAPER

Flash as cache extension for online transactional workloads

Woon-Hak Kang[1] · **Sang-Won Lee**[1] · **Bongki Moon**[2]

Received: 31 January 2015 / Revised: 20 August 2015 / Accepted: 14 November 2015 / Published online: 30 November 2015

Abstract Considering the current price gap between hard disk and flash memory SSD storages, for applications dealing with large-scale data, it will be economically more sensible to use flash memory drives to supplement disk drives rather than to replace them. This paper presents *FaCE*, which is a new low-overhead caching strategy that uses flash memory as an extension to the RAM buffer of database systems. *FaCE* aims at improving the transaction throughput as well as shortening the recovery time from a system failure. To achieve the goals, we propose two novel algorithms for flash cache management, namely *multi-version FIFO replacement* and *group second chance*. This was possible due to *flash write optimization* as well as *disk access reduction* obtained by the *FaCE* caching methods. In addition, *FaCE* takes advantage of the nonvolatility of flash memory to fully support database recovery by extending the scope of a persistent database to include the data pages stored in the flash cache. We have implemented *FaCE* in the PostgreSQL open-source database server and demonstrated its effectiveness for TPC-C benchmarks in comparison with existing caching methods such as Lazy Cleaning and Linux Bcache.

Keywords Flash memory SSDs · Cache · Recovery

✉ Sang-Won Lee
swlee@skku.edu

Woon-Hak Kang
woonagi319@skku.edu

Bongki Moon
bkmoon@snu.ac.kr

[1] School of Information and Communication Engineering, Sungkyunkwan University, Suwon 440-746, Korea

[2] Department of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea

1 Introduction

As the technology of flash memory solid-state drives (SSDs) continues to advance, they are increasingly adopted in a wide spectrum of storage systems to deliver higher throughput at more affordable prices. For example, it has been shown that flash memory SSDs can outperform disk drives in terms of throughput, energy consumption, and cost-effectiveness for database workloads [26,27]. Nevertheless, it is still true that the price per unit capacity of flash memory SSDs is higher than that of disk drives, and the market trend is likely to continue for the foreseeable future. Therefore, for applications dealing with large-scale data, it may be economically more sensible to use flash memory SSDs to supplement disk drives rather than to replace them.

In this paper, we present a low-overhead strategy and its implementation for using flash memory as an extended cache for a recoverable database. This method is referred to as *Flash as Cache Extension* or *FaCE* for short. Like any other caching mechanism, the objective of *FaCE* is to provide *the performance of flash memory* and *the capacity of disk* at as little cost as possible. We set out to achieve this with the realization that flash memory has drastically different characteristics (such as no overwriting, slow writes, and nonvolatility) than DRAM buffer, and they must be dealt with carefully and exploited effectively. There are a few existing approaches that store frequently accessed data in flash memory drives by using them as either faster disk drives or an extension to the RAM buffer. The *FaCE* method we propose is in line with the existing approaches using flash memory as an extension to the DRAM buffer, but it is also different from them in several ways.

First, the *FaCE* framework aims at *flash write optimization* as well as *disk access reduction*. Unlike DRAM buffer that yields almost uniform performance for both random and

sequential accesses, the performance of flash memory varies considerably depending on the type of operations (i.e., read or write) and the pattern of accesses (i.e., random or sequential). With most contemporary flash memory SSDs, random writes are slower than sequential writes approximately five to ten times. (See Table 1.) The *FaCE* framework provides flash-aware strategies for managing the flash cache that can be designed and implemented independently from the RAM buffer management policy. By turning small random writes to large sequential ones, high sequential bandwidth and internal parallelism of modern flash memory SSDs can be utilized more effectively for higher throughput [12]. To achieve this, we propose a flash cache replacement method called *multi-version FIFO* (*mvFIFO*).

Second, *FaCE* takes advantage of the nonvolatility of flash memory and extends the scope of a persistent database to include the data pages stored in the flash cache. Once a data page evicted from the RAM buffer is staged in the flash cache, it is considered having been propagated to the persistent database. Therefore, the data pages in the flash cache can be utilized to minimize the recovery overhead, accelerate restarting the system from a failure, and achieve transaction atomicity and durability at the nominal cost. The only additional processing required for a system restart is to restore the meta-data of flash cache, but it does not take more than just a few seconds. In addition, since most data pages that need to be accessed during the recovery phase can be found in the flash cache, the recovery time will be significantly shortened. The recovery manager of *FaCE* provides mechanisms that fully support database recovery and persistent meta-data management for cached pages.

Third, *FaCE* is a low-overhead framework that uses a flash memory drive as an extension of a RAM buffer rather than a disk replacement. The use of flash cache is tightly coupled with the RAM buffer. Unlike some existing approaches, a data page is cached in flash memory *not on entry* to the RAM buffer, but instead *on exit* from it. This is because a copy in the flash cache will never be accessed while another copy of the same page exists in the RAM buffer. The run-time overhead is very small, as there is no need for manual or algorithmic elaboration to separate hot data items from the cold ones. *FaCE* reduces but never increases the amount of traffic to and from disk, because it does not migrate data items between flash memory and disk drives just for the sake of higher cache hits. When the flash cache is full, if a page is to be removed and it is dirty, it will be written to disk, very much like a dirty page evicted from the RAM buffer and flushed to disk.

We have implemented the *FaCE* framework in the PostgreSQL open-source database server. Most of the changes necessary for *FaCE* are made within the buffer manager module, the checkpoint process, and the recovery daemon. In the TPC-C benchmarks, we observed that *FaCE* yielded hit rates from the flash cache in the range from low 72 to 91 % and reduced disk writes 46–89 % consistently. This high hit rate and considerable write reduction led to substantial improvement in transaction throughput by up to a factor of two or more. Besides, *FaCE* reduced the restart time by more than a factor of four consistently over the various checkpoint intervals.

The rest of this paper is organized as follows: Section 2 reviews the characteristics of contemporary flash memory and our motivation as well as cost-effectiveness of flash memory cache. Section 3 reviews the previous work on flash memory caching and presents the analysis of the cost-effectiveness of flash memory as an extended cache. In Sect. 4, we overview the basic framework and design alternatives of *FaCE* and present the key algorithms for flash cache management. Section 5 presents the database recovery system designed for *FaCE*. In Sect. 6, we analyze the performance impact of the *FaCE* system on the TPC-C workloads and demonstrate its advantages over the existing approaches. Lastly, Sect. 7 summarizes the contributions of this paper.

2 Motivation

A flash memory SSD is a nonvolatile storage device that has different characteristics than a traditional magnetic disk drive. With no mechanical parts inside, the SSD can achieve low access latency and high random throughput utilizing ample internal parallelism. However, it does not allow any data to be updated in place and takes longer time to process a write operation than a read operation. This section motivates our work on *FaCE* by analyzing the characteristics of SSDs with respect to random and sequential performance, especially for write operations, and the cost-effectiveness of flash memory as a cache extension.

2.1 Disparity in SSD write performance

As the technology of flash memory SSDs continues to innovate, the performance has improved remarkably in both sequential bandwidth and random throughput. Table 1 compares the performance characteristics of flash memory SSDs and magnetic disk drives. The throughput and bandwidth in the table were measured by the Flexible I/O tester [4] when the devices were in the steady state. The flash memory SSDs are consumer and datacenter-oriented products. The magnetic disks are enterprise-class 15k RPM drives, and they were tested as a single unit or as a 16-disk RAID-0 array.

The currently available consumer-grade SSDs use a SATA3 interface running at 6.0 Gb/s. In Table 1, Samsung SSD 840 Pro and Intel DC S3500 use a SATA3 interface. There exists a substantial disparity between random and sequential performance measured in terms of bandwidth for

Table 1 Performance characteristics of flash memory SSDs and magnetic disk drives

Storage media	RAND (IOPS)		SEQ (MB/s)		Capacity in GB	Market price as of 2015	Price per GB($)	Release year
	Read	Write	Read	Write				
Samsung 840 Pro[a]	65,803	13,979	534	398	256	$159	0.62	Nov. 2012
Samsung 850 Evo[a]	98,591	6,094	518	300	250	$98	0.39	Dec. 2014
Intel DC S3500[a]	60,311	21,572	470	425	480	$554	1.15	Aug. 2013
Intel NVMe P3700[b]	457,355	86,949	2,671	676	400	$1198	2.97	Jun. 2014
Single disk[c]	409	343	156	154	146.8	$26	0.18	Jul. 2011
16-disk[c] RAID-0	5,166	5,169	1723	1645	2,340	$416	0.18	Jul. 2011

RAND : 4 kB random IOPS, SEQ : 128 kB sequential bandwidth
SSD: [a]SATA 3(6 Gb/s), [b]PCIe
[c]Disk: Seagate Cheetah 15.6 K 146.8 GB (SAS 3 Gb/s)

write operations. The sequential write bandwidth of 840 Pro SSD was about seven times higher than its random write bandwidth. The ratio was five in the case of Intel DC S3500 SSD. The DC S3500 is a datacenter-oriented SSD. Our work on *FaCE* aims at developing a new flash caching method that exploits maximally the superior sequential write performance of flash memory SSDs.

It should also be noted that SSD vendors have continuously improved the random write performance in resource-intensive ways such as over-provisioning and battery-backed cache [24] to diminish the disparity of random and sequential performance. For instance, Intel DC P3700 NVMe SSD, as given in Table 1, yields just about twice higher sequential write bandwidth than its random write bandwidth. The advantages of *FaCE* over LC may vanish with those high-end SSDs. However, NVMe SSDs are about five times more expensive than commodity SSDs in terms of $/GB. One of the goals of *FaCE* is the cost-effective use of commodity SSDs. Given the low cost of commodity SSDs and the life span that can be extended by *FaCE*, we still believe *FaCE* is a caching scheme that is superior to LC with respect to performance, cost, and endurance.

2.2 Write amplification and flash endurance

Many vendors have been manufacturing SSD products using TLC NAND flash chips in order to increase the storage capacity while keeping the cost low. TLC (three-level cell) NAND flash chips achieve higher density than SLC (single-level cell) or MLC (two-level cell) NAND flash chips by storing three bits in a cell, but offer a much lower level of endurance. TLC NAND flash chips allow just about 1000 program/erase (or P/E in short) cycles, and consequently, SSD products with TLC NAND chips are more prone to durability and reliability issues.

As for the endurance of an SSD, *write amplification factor (WAF)* is the most commonly used parameter, which indicates how effectively the scarce P/E cycles of NAND flash chips are utilized. WAF is a numerical value that can be computed by the following equation.

$$\text{WAF} = \frac{\text{Physical writes to NAND}}{\text{Logical writes by HOST}}$$

The lower the WAF value is, the more effectively the P/E cycles are utilized and hence the longer the life span of an SSD is expected for the same workload.

The amount of logical or physical writes can be measured by using the S.M.A.R.T (Self-Monitoring, Analysis, and Reporting) technology, which is supported by most contemporary SSD products for accessing the internal state such as *wear leveling count* [32] or *media wearout indicator* [21]. To measure the amount of physical writes to NAND, we collected the `wear_leveling_count` (ID #177) from the Samsung SSD and the `media_wearout_indicator` (ID #233) from the Intel SSD. The amount of logical writes was obtained by collecting the `total_LBAs_written` (ID #241) from the both devices.

The wearout of an SSD is affected by several factors such as the randomness, locality, and sizes of writes. Table 2 compares the level of wearout measured when the unit size of writes was 256 or 4 kB. We measured WAF values for two different commodity SSDs after executing random reads and writes with a command queue depth of 32 for 10 h. With a larger unit of writes, WAF was lower about ten times for the Samsung SSD product and about three times for the Intel SSD

Table 2 Effect of writes sizes on WAF

Storage media (total NAND size in GB)	Write sizes	
	256 kB	4 kB
Samsung 840 Pro (256 GB)	1.03	5.15
Intel DC S3500 (526 GB)	1.08	3.04

product. This clearly indicates that *FaCE* can help extend the life span of SSDs by turning many small random writes into a few large sequential writes.

2.3 Cost-effectiveness of flash cache

Suppose a page is evicted from the RAM buffer to disk and is fetched back from disk to the RAM buffer again. The I/O cost involved in this process will be a disk read and an optional disk write if the page was dirty.

With a buffer replacement algorithm that does not suffer from Belady's anomaly [6], an increase in the number of buffer frames is generally expected to provide a fewer page faults. Tsuei *et al.*have shown that the data hit rate is a linear function of log ($BufferSize$) in OLTP workloads when the database size is fixed [38]. Based on this observation, we analyze the cost-effectiveness of flash memory as a cache extension. The question we are interested in answering is *how much flash memory will be required to achieve the same level of reduction in I/O time obtained by an increase in DRAM buffer capacity.* In the following analysis, C_{disk} and C_{flash} denote the time taken to access a disk page and the time taken to access a flash page, respectively.

Suppose the DRAM buffer size is increased from B to $(1+\delta)B$ for some $\delta > 0$. Then, the increment in the hit rate is expected to be

$$\alpha \log((1+\delta)B) - \alpha \log(B) = \alpha \log(1+\delta)$$

for a positive constant α. The increased hit rate will lead to reduction in disk accesses, which accounts for reduced I/O time by $\alpha C_{\text{disk}} \log(1+\delta)$. If the δB increment of DRAM buffer capacity is replaced by an extended cache of θB flash memory, then the data hit rate (for both DRAM hits and flash memory hits) will increase to $\alpha \log((1+\theta)B)$. Since the rate of DRAM hits will remain the same, the rate of flash memory hits will be given by

$$\alpha \log((1+\theta)B) - \alpha \log(B) = \alpha \log(1+\theta).$$

Each flash memory hit translates to *an access to disk replaced by an access to flash memory*. Therefore, the amount of reduced I/O time by the flash cache will be $\alpha(C_{\text{disk}} - C_{\text{flash}}) \log(1+\theta)$.

The break-even point for θ is obtained by the following equation

$$\alpha C_{\text{disk}} \log(1+\delta) = \alpha(C_{\text{disk}} - C_{\text{flash}}) \log(1+\theta)$$

and is represented by a formula below.

$$1+\theta = (1+\delta)^{\frac{C_{\text{disk}}}{C_{\text{disk}} - C_{\text{flash}}}} \quad (1)$$

For most contemporary disk drives and flash memory SSDs, the value of $\frac{C_{\text{disk}}}{C_{\text{disk}} - C_{\text{flash}}}$ is very close to one. For example, with a Seagate disk drive and a Samsung flash memory SSD shown in Table 1, the value of the fraction is approximately 1.006 for read-only workload and 1.025 for write-only workload.

This implies that the effect of extended caching by flash memory is almost as good as that of extended DRAM cache. Given that NAND flash memory is almost ten times cheaper than DRAM with respect to price per capacity and the price gap is expected to grow, this analysis demonstrates that the cost-effectiveness of flash cache is indeed significant, especially when the system is IO bound.

3 Related work

3.1 Faster disk versus buffer extension

In some of the existing approaches [10,25], flash memory drives are used as yet another type of disk drives with faster access speed. Flash memory drives may replace disk drives altogether for small- to medium-scale databases or may be used more economically to store only frequently accessed data items for large-scale databases. When both types of media are deployed at the same time, a data item typically resides exclusively in either of the media unless disk drives are used in a lower tier in the storage hierarchy [17].

One technical concern about this approach is the cost of identifying hot data items. This can be done either statically by profiling [10] or dynamically by monitoring at run-time [25], but not without drawbacks. While the static approach may not cope with changes in access patterns, the dynamic one may suffer from excessive run-time and space overheads for identifying hot and data objects and migrating them between flash memory and disk drives. It is shown that the dynamic approach becomes less effective when the workload is update intensive [10].

In contrast, if flash memory SSDs are used as a RAM buffer extension or a cache layer between RAM and disk, it can simply go along with the data page replacement mechanism provided by the RAM buffer pool without having to provide any additional mechanism to separate hot data pages from cold ones. There is no need to monitor data access patterns hence very little run-time overhead and no negative impact from the prediction quality of future data access patterns.

Two important observations can be made in Table 1 in regard to utilizing flash memory as a cache extension. First, there still exists considerable bandwidth disparity between random writes and sequential writes with both commodity and datacenter SSDs. The random write bandwidth was in the range of merely 10–20 % of sequential write bandwidth, while random read enjoys bandwidth much closer (in the

range of 44–50 %) to that of sequential read. Unlike DRAM buffer that yields uniform performance regardless of types or patterns of accesses, the design of flash cache management should take the unique characteristics of flash memory into account.

Second, disk arrays are a very cost-effective means for providing high sequential bandwidth. However, they still fall far behind flash memory SSDs with respect to random I/O throughput, especially for random reads. Therefore, the caching framework for flash memory should be designed such that random disk I/O operations are replaced by random flash read and sequential flash write operations as much as possible.

3.2 Flash cache as database bufferpool extension

Flash memory SSDs have recently been adopted by commercial database systems to store frequently accessed data pages. For example, for data warehouse applications, Oracle Exadata Smart Flash Cache caches hot data pages in flash memory when they are fetched from disk in the write-around fashion [1,30]. Hot data selection is done statically by the types of data such that tables and indexes have higher priority than log and backup data. A more recent edition of Oracle Exadata supports the write-back sync policy as well for OLTP workloads [37].

Similarly, the Bufferpool Extension prototype of IBM DB2 proposes a temperature-aware caching (TAC) scheme that relies on data access frequencies [9,11]. TAC monitors data access patterns continuously to identify hot data regions at the granularity of an extent or a fixed number of data pages. Hot data pages are cached in the flash memory cache *on entry* to the RAM buffer from disk. TAC adopts a write-through caching policy. When a dirty page is evicted from the RAM buffer, it is written to both the flash cache and disk. Consequently, the flash cache provides caching effect for read operations but no effect of reducing disk write operations. Besides, the high cost of maintaining cache meta-data persistently in flash memory degrades the overall performance. (See Sect. 5.1 for more detailed descriptions of the persistent meta-data management.)

In contrast, the Lazy Cleaning (LC) method caches data pages in flash memory upon exit from the RAM buffer and handles them by a write-back policy if they are dirty [14]. It uses a background thread to flush dirty pages from the flash cache to disk, when the percentage of dirty pages goes beyond a tunable threshold. The LC method manages the flash cache using LRU-2 replacement algorithm. Hence, replacing a page in the flash cache incurs costly random read and random write operations.

The LC method was recently enhanced with a recovery mechanism [13]. The recoverable LC uses the flash cache mapping directory and database transaction log to avoid checkpointing dirty pages cached in flash memory. The mapping directory stores the logical to physical address mapping information for data pages cached in the flash memory. Since the mapping directory is stored in the log, the mapping information can be rebuilt from the log during the system restart. The recoverable LC method can reduce ramp-up time. However, *FaCE* has already shown that the flash cache can accelerate database recovery by utilizing the persistency of flash memory and meta-data checkpointing [23].

The Cost-Adjusted Caching (CAC) adopted the Greedy Dual algorithm to manage the buffer pool and pages cached in the flash memory SSD [28]. CAC elaborates on the benefit of caching individual pages to determine which pages to cache in flash memory. This requires maintaining the reference information for all the data pages cached in the SSD. However, it is unclear in the study how much the cost of maintaining references will increase as the number of pages in the flash cache increases. The authors of CAC reported that *mvFIFO* performed poorly for TPC-C workloads due to the low utilization of SSD. However, they neglected the fact that the low utilization of SSD is an evidence that *mvFIFO* turns random writes to sequential ones effectively. The low utilization also indicates that the impact of flash cache will be sustained longer without saturation as the size of a disk farm increases. We have observed that *mvFIFO* kept the utilization of an SSD as a flash cache in the range of 60–80 % for a RAID-0 array with 16 disk drives.

Among the aforementioned flash caching approaches, the LC method is the closest to the *FaCE* caching scheme presented in this paper. In both approaches, data pages are cached in flash memory upon exit from the RAM buffer and managed by a write-back policy. Other than this similarity, the *FaCE* method differs from the LC methods in a few critical aspects of utilizing flash memory for caching and database recovery purposes and achieves 2.3 times higher transaction throughput than the LC method.

First of all, *FaCE* considers not only cache hit rates but also *write optimization* of flash cache. It manages the flash cache in the first-in-first-out fashion and allows multiple versions left in the flash cache, so that random writes are avoided or turned into sequential ones. The FIFO style replacement allows *FaCE* to replace a group of pages at once so that internal parallelism of a flash memory SSD can be exploited. It also boosts the cache hit rate by allowing a second chance to stay in the flash cache for warm data pages. In general, a flash caching mechanism improves the throughput of an OLTP system as the capacity of its flash cache increases, which will peak when the entire database is stored in the flash cache.

Second, *FaCE* extends the scope of a persistent database to include the data pages stored in the flash cache. It may sound a simple notion but its implications are significant. For example, database checkpointing can be done much more

efficiently by flushing dirty pages to the flash cache rather than disk and by not subjecting data pages in the flash cache to checkpointing. Furthermore, the persistent data copies stored in the flash cache can be utilized for faster database recovery from a failure. *FaCE* provides a low-overhead mechanism for maintaining the meta-data persistently in flash memory. We have implemented all the caching and recovery mechanisms of *FaCE* fully in the PostgreSQL system.

3.3 Flash cache in the OS layer

Recently, several flash caching methods have been developed into the operating system block layer for more general applications. Some of them are commercial products: Intel's Cache Acceleration Software(CAS) [20], FusionIO's ioTurbine [16], and SanDisk's FlashSoft [33]. Other are open source: Google's Bcache [31], Facebook's flashcache [35], dm-cache [40] and STEC's enhance IO [36]. In this section, we briefly describe Bcache, which has been available in the block layer since Linux kernel version 3.10, in more detail.

Bcache aims at optimizing the caching performance of a flash memory SSD by minimizing random write requests onto it. To achieve this goal, Bcache uses a large chunk (or bucket) of flash memory as a basic unit of replacement. The default size of a bucket is 512 kB, but it is meant to be equal to the size of an erase unit of a flash memory SSD being used as a caching device. The cache buckets can be managed by the LRU, FIFO, or Random replacement algorithm and are synced with a backing device (or a disk drive) in the write-through (default), write-back, or write-around mode. On a read miss, Bcache fetches the missed page from the backing device and hands it over to the upper layer. The fetched page is inserted into the flash cache by a background daemon. The daemon is also responsible for garbage collection of flash cache.

Bcache segregates write requests to take advantage of the locality of data. If write requests are initiated by the same process or sequentiality is detected in the logical block addresses, those write requests are inserted into the same open bucket. Otherwise, they are scattered to different flash buckets. Bcache bypasses the flash cache if data being written are unlikely to be read again shortly. For example, when a large chunk of data is written, only the first bucket is cached in flash memory while the rest is written to the backing device directly. This bypassing method prevents the flash cache from being polluted by cold data.

There is an important distinction between flash caching methods as a database bufferpool extension (e.g., *FaCE*) and those provided in the operating system block layer (e.g., Bcache). While the former can always decide whether to cache a page on entry from disk to the RAM buffer or on exit from the RAM buffer to disk, the latter can only cache a page on entry from disk to the RAM buffer unless the page is evicted dirty. If a page is evicted clean, the page is simply discarded without the block layer being notified of the fact. In other words, the flash cache in the block layer has no choice but to cache clean pages *on entry* when they are fetched from disk.

3.4 Log-structured database techniques for flash storages

Recently, there has been a widening body of work based on log-structured techniques that treat the log as the database, particularly in the field of NoSQL and distributed databases [5,7,8,34]. For example, Hyder bases itself on the log-structured database in order to make the best use of flash-based storage [7,8]. In general, random writes could be transformed into sequential writes to a log-structured database made in the append-only fashion for higher write bandwidth.

In this regard, the motivation behind the flash-based log-structured techniques is in the same vein with that of *FaCE*, although those bodies of work use SSDs as main storage instead of caching storage.

4 Flash as cache extension

The principal benefit of using flash memory as an extension to a RAM buffer is that a mix of flash memory and disk drives can be used in a unified way without manual or algorithmic intervention for data placement across the drives. When a data page is deemed cold and swapped out by the buffer manager, it may be granted a chance to stay in the flash cache for an extended period. If the data page turns out warm enough to be referenced again while staying in the flash cache, then it will be swapped back into the RAM buffer from flash memory much more quickly than from disk. If a certain page is indeed cold and not referenced for a long while, then it will not be cached in either the RAM buffer or the flash cache.

4.1 Basic framework

Figure 1 illustrates the main components and their interactions of a caching system deployed for a database system,

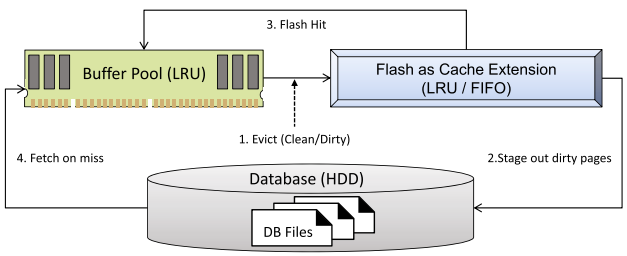

Fig. 1 Flash as cache extension: overview

when the flash cache is enabled. The main interactions between the components are summarized as follows.

- When the database server requests a data page that is not in memory (i.e., the RAM buffer), the flash cache is searched for the page. (A list of pages cached in flash memory is maintained in the RAM buffer to support the search operations.) If the page is found in the flash cache (i.e., **flash hit**), it is fetched from the flash cache. Otherwise, it is fetched from disk.
- When a page is swapped out of the RAM buffer, different actions can be taken on the page depending on whether it is clean or dirty. If it is clean, the page is discarded or **staged in** to the flash cache. If it is dirty, the page is written back to disk or **staged in** to the flash cache, or both.
- When a page is **staged out** of the flash cache, different actions can be taken on the page depending on whether it is clean or dirty. If it is clean, the page is just discarded. If it is dirty, the page is written to disk unless it was already written to disk when evicted from the RAM buffer.

Evidently from the key interactions described above, the fundamental issues are when and which data pages should be staged in to or out of the flash cache. There are quite a few alternatives to addressing the issues, and they may have profound impact on the overall performance of the database server. For example, when a data page is fetched from disk on a RAM cache miss, we opt not to enter the page to the flash cache, because the copy in the flash cache will never be accessed while the page is cached in the RAM buffer. For this reason, staging a page in to the flash cache is considered only when the page is evicted from the RAM buffer.

4.2 Design choices for *FaCE*

The *FaCE* caching scheme focuses on how exactly data pages are to be staged in the flash cache and how the flash cache should be managed. Specifically, in the rest of this section, we discuss alternative strategies toward the following three key questions and justify the choices made for the *FaCE* scheme: (1) When a dirty page is evicted from the RAM buffer, does it have to be written through to disk as well as the flash cache or only to the flash cache? (2) Which replacement algorithm is better suited to exploiting flash memory as an extended cache? (3) When a clean page is evicted from the RAM buffer, does it have to be cached in flash memory at all or given as much preference as a dirty page? These are orthogonal questions. An alternative strategy can be chosen separately for each of the three dimensions.

Note that the *FaCE* scheme does not distinguish data pages by types (e.g., index pages vs. log data) nor monitor data access patterns to separate hot pages from cold ones. Nonetheless, there is nothing in the *FaCE* framework that disallows such additional information to be used in making caching decisions. Table 3 summarizes the design choices of *FaCE* and compares them with existing flash memory caching methods. The design choices of *FaCE* will be elaborated in this section.

4.2.1 Data sync policy

When a dirty page is evicted from the RAM buffer, it may be *written through* to both disk and the flash cache. Alternatively, a dirty page may be *written back* only to the flash cache, leaving its disk copy intact and outdated until it is synchronized with the current copy when being staged out of the flash cache and written to disk.[1]

The two alternative approaches of *write-back* and *write-through* are equivalent in the effect of flash caching for read operations. However, the overhead of the *write-through* policy is obviously much higher than that of the *write-back* policy for write operations. While the *write-through* policy requires a disk write as well as a flash write for each dirty page being evicted from the DRAM buffer, the *write-back* policy reduces disk writes by staging them in the flash cache until they are staged out to disk. In effect, *write-back* replaces one or more disk writes (required for a dirty page evicted repeatedly) with as many flash writes followed by a single (deferred) disk write. For the reason, *FaCE* adopts the *write-back* policy rather than *write-through*.

Now that the flash and disk copies of a data page may not be in full sync, *FaCE* ensures that a request of the page from the database server is always served by the current page. Note that the adoption of a *write-back* policy does not make the database server any more vulnerable to data corruption, as flash memory is a nonvolatile storage medium. In fact, the *FaCE* caching scheme provides a low-cost recovery mechanism that takes advantage of the persistency of flash-resident cached data. (Refer to Sect. 5 for details.)

Modern high-capacity SSDs have started adopting TLC flash memory chips. They have very low endurance with only about a thousand P/E cycles, and they are prone to unexpected failures. In order to avoid data loss even with less reliable devices, any flash caching scheme should be equipped with protection from data loss. For this reason, we have also implemented the *write-through* policy, which synchronizes data between the flash cache and disk at all times such that the database can be recovered even in the presence of a failure of a flash caching device.

[1] The *write-around* policy and its variants are suggested for streaming and OLAP workloads to avoid polluting the cache [2]. We do not discuss them further in this paper, because our focus is on OLTP workloads.

Table 3 Comparison of flash caching methods: WT, WB, and WA denote write-through, write-back, and write-around, respectively

	Exadata	TAC	LC	Bcache	*FaCE*
When	On entry	On entry	On exit	On entry	On exit
What	Clean/Dirty	Clean/Dirty	Clean/Dirty	Clean/Dirty	Clean/Dirty
Sync	WA/WB	WT	WT/WB	WT/WB/WA	WT/WB
Replacement	LRU	LRU	LRU/2	LRU/FIFO/Random	mvFIFO+GSC
Granularity	Page	Page	Page	Bucket	Page

4.2.2 Flash cache replacement policy

The size of a flash cache is likely to much larger than the RAM buffer but is not unlimited either. The page frames in the flash cache need to be managed carefully for better utilization. The obvious first choice is to manage the flash cache by LRU replacement. With LRU replacement, a victim page is chosen at the rear end of the LRU list regardless of its physical location in the flash cache. This implies that each page replacement will incur a random flash write for an (logically) in-place replacement. Random writes are the pattern that requires flash memory to make the most strenuous effort to process, and the bandwidth of random writes is typically an order of magnitude lower than that of sequential writes with flash memory.

Alternatively, a flash cache can be managed by FIFO replacement. The FIFO replacement may seem irrational, given that it is generally considered inferior to the LRU replacement and can suffer from Belady's anomaly with respect to hit rate. Nonetheless, the FIFO replacement has its own unique merit when it is applied to a flash caching scheme. Since all incoming pages are enqueued to the rear end of the flash cache, all flash writes will be done sequentially in the append-only fashion. This particular write pattern is known to be a perfect match with flash memory [27] and helps the flash cache yield the best performance.

The *FaCE* scheme adopts a variant of FIFO replacement. This is different from the traditional FIFO replacement in that one or more different versions of a data page are allowed to be present in the flash cache simultaneously. When a flash frame is to be replaced by an incoming page, a victim frame is selected at the front end of the flash cache and the incoming page is enqueued to the rear end of the flash cache. If the incoming page is dirty, then it is enqueued unconditionally. No additional action is taken to remove existing versions in order not to incur random writes. If the incoming page is clean, then it is enqueued only when its copy does not exist in the flash cache. A data page dequeued from the flash cache is written to disk only if it is dirty and it is the latest version in the flash cache, so that redundant disk writes can be avoided. We call this approach *multi-version FIFO* (*mvFIFO*) replacement. Refer to Sect. 4.3 for the elaborate design of *FaCE* and its optimization.

Despite the previous studies reporting the limitations of LRU replacement applied to a second-level buffer cache [39, 41], we observed that LRU replacement still delivered higher hit rates in the flash cache than the *mvFIFO* replacement. This is partly attributed to the fact that the former keeps no more than a single copy of a page cached in flash memory while the latter may need to store one or more versions of a page. However, the gap in the hit rates was narrow and far outweighed by the reduced I/O cost of the *mvFIFO* replacement with respect to the overall transaction throughput.

4.2.3 Page admission policy: clean and dirty

Caching a page in flash memory will be beneficial, if the cached copy is referenced again before being removed from the flash cache and the cost of a disk read is higher than the combined cost of a flash write and a flash read, which is true for most contemporary disk drives and flash memory SSDs. The caching decision thus needs to be made based on how probable it is a cached page will be referenced again at least once before the page is removed from the flash cache.

As a matter of fact, if a page being evicted from the RAM buffer is dirty, it is always beneficial to cache the page in flash memory, because an immediate disk write would be requested for the page otherwise. In addition, by caching dirty pages, the *write-back* policy can turn a multitude of disk writes—required for repeated evictions of the same page from the RAM buffer—into as many flash writes followed by a single disk write.

On the other hand, the threshold for caching a clean page in flash memory is higher. This is because caching a clean page could cause a dirty page to be evicted from the flash cache and written to disk, while no disk write would be incurred if the clean page was simply discarded. In that case, the cost of caching a clean page, which is no less than a disk write and a flash write, will not be recovered by a single flash hit.

Therefore, especially when a *write-back* policy is adopted, dirty pages should have priority over clean ones with respect to caching in flash memory. This justifies the adoption of our new *mvFIFO* replacement that allows multiple versions of a data page to be present in the flash cache.

4.3 The *FaCE* system

In a traditional buffer management system, a *dirty* flag is maintained for each page kept in the buffer pool to indicate whether the page has been updated since it was fetched from disk. Using a single flag per page in the buffer is sufficient, because there exist no more than two versions of a data page in the database system at any moment. In the *FaCE* system, however, a data page can reside in the flash cache as well as a DRAM buffer and disk. Therefore, the number of different copies of a page may be more than two, not to mention the different versions of data pages that may be maintained in the flash cache by the *mvFIFO* replacement.

We introduce another flag called a *flash dirty* (or *fdirty* in short) to represent the state of a data page accurately. Much like a dirty flag is set for a data page updated in the DRAM buffer, if a *fdirty* flag is set for a buffered data page that is newer than its corresponding flash-resident copy. With the flags playing different roles, the *FaCE* system can determine what action needs to be taken when a page is evicted from the DRAM buffer or from the flash cache. This will be explained in detail shortly.

Both *dirty* and *fdirty* flags are reset to zero, when a page is fetched from disk (because a copy does not exist in the flash cache). They are both set to one when the page is updated in the DRAM buffer. If a page is fetched from the flash cache after being evicted from the DRAM buffer, then the *fdirty* flag must be reset to zero to indicate that the two copies in the DRAM buffer and the flash cache are synced. However, the *dirty* flag of this page must remain unaffected, since this page in the DRAM and flash cache may still be newer than its disk copy. This implies that while the *fdirty* flags are needed only for the pages in the DRAM buffer, the *dirty* flags must be maintained for the pages both in the DRAM buffer and in the flash cache.

When a page is evicted from the DRAM buffer, it is enqueued to the flash cache unconditionally if the *fdirty* flag is on. Otherwise, it is enqueued to the flash cache only when there is no identical copy already in the flash cache. If a copy is enqueued unconditionally by a raised *fdirty* flag, it will be the most recent version of the page in the flash cache. Since the conditional enqueuing ensures no redundancy in the flash cache, these enqueuing methods—conditional and unconditional—of *mvFIFO* guarantee that there will be no more than one copy of one distinct version and the latest version of a page can be considered the only valid copy among those in the flash cache.

For the convenience of disposing invalid copies, we maintain a *valid* flag for each page cached in the flash memory. The *valid* flag is set for any page entering the flash cache, which in turn invalidates the previous version in the flash cache so that there exists only a single valid copy at any moment in time. The previous version is invalidated by simply removing it from the hash table and marking it as invalid in the meta-data directory. Since the hash table and meta-data directory are memory resident, invalidating an old version does not incur any additional I/O operation until they are flushed to stable storage by meta-data checkpointing. (Refer to Sect. 5.1 for checkpointing.) When a page is dequeued from the flash cache, we will flush it to disk only if the *dirty* and *valid* flags are both on. Otherwise, it will be simply discarded.

The main operations of the *mvFIFO* replacement are illustrated in Fig. 2 and summarized in Algorithm 1.

Group second chance

The *second chance* replacement is a variant of the FIFO replacement and is generally considered superior to its basic form. The same idea of second chance can be adopted for the *mvFIFO* replacement. With a second chance given to a *valid* page being dequeued from the flash cache, if the page has been referenced while staying in the flash cache, it will be enqueued back instead of being discarded or flushed to disk.

For a DRAM buffer pool, the second chance replacement is implemented by associating a *reference* flag with each page and by having a *clock hand* point to a page being considered for a second chance. Since it does not involve copying or moving pages around, the second chance replacement can be adopted with a little additional cost for setting and resetting *reference* flags. For the flash cache being managed by the *mvFIFO* replacement, on the other hand, it is desired that data pages are *physically* dequeued from and enqueued to flash memory for efficiency of sequential I/O operations.

The negative aspect of this, however, is the increased I/O activities, as dequeuing and enqueuing a page require two I/O operations to be made to the flash cache. This will be further aggravated if more than a few valid (and referenced) pages need to be examined by the second chance replacement before a victim page is found in the flash cache. It is an ironic situation, because the more pages in the flash cache are hit by references or utilized, the more likely the cost of a replacement grows higher.

To address this concern, we propose a novel **group second chance (GSC)** replacement for the flash cache. When a page evicted from the DRAM buffer is about to enter the flash cache, a replacement is triggered to make a space for the page. The pages at the front end of the flash cache will be scanned to find a victim page, but the group second chance limits the scan depth so that the replacement cost is bounded. Though the scan depth can be set to any small constant, it will make a practical sense to set the scan depth to no more than the number of pages (typically 64 or 128) per a flash memory block.

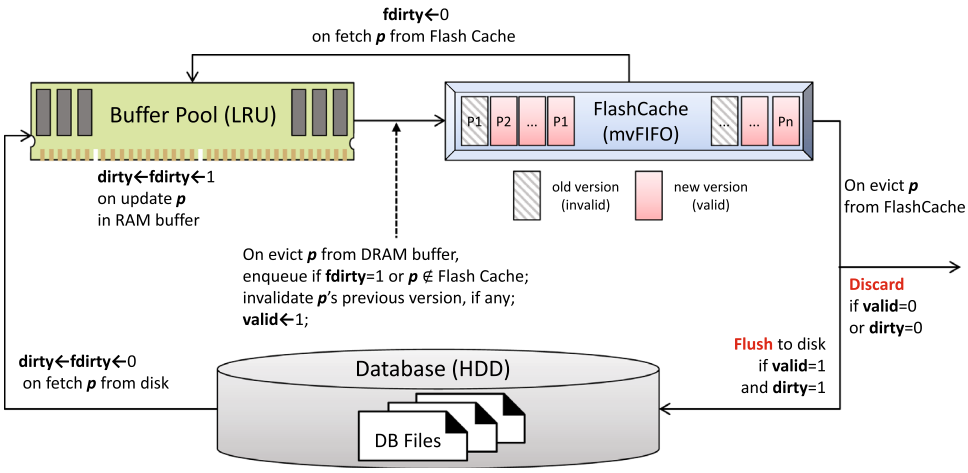

Fig. 2 Multi-version FIFO flash cache

Algorithm 1: Multi-version FIFO Replacement

On eviction of page p from the DRAM buffer:
  if $p.fdirty = true \vee p \notin flash\ cache$ **then**
    invalidate the previous version of p if it exists;
    p is enqueued to the flash cache;
    $p.valid \leftarrow true$;
  end if
On eviction of page p from the flash cache:
  if $p.dirty = true \wedge p.valid = true$ **then**
    p is written back to disk;
  else
    p is discarded;
  end if
On fetch of page p from disk:
  $dirty \leftarrow fdirty \leftarrow false$;
On fetch of page p from the flash cache:
  $fdirty \leftarrow false$;
On update of page p in the DRAM buffer:
  $dirty \leftarrow fdirty \leftarrow true$;

All the pages within the scan depth are dequeued from the flash cache in a batch. Following the basic *mvFIFO* replacement, the pages in a batch are either discarded or flushed to disk if their reference flags are down. Then, the remaining pages will be enqueued back to the flash cache. In a rare case where all the pages in the batch are referenced, the page at the very front end will be discarded or flushed to disk in order to make a space for an incoming page from the DRAM buffer. In a more typical case, the number of pages to be enqueued back will be much smaller than the scan depth. In this case, more pages are *pulled* from the LRU tail of the DRAM buffer to fill up the space in the batch. This ensures that a dequeuing or enqueuing operation will be carried out by a single batch-sized I/O operation, much more infrequently than being done for individual pages.

Pulling page frames from the DRAM buffer is analogous to what is done by the background write-back daemons of the Linux kernel or the DBWR processes of the Oracle database server. The Linux write-back daemons wake up when free memory shrinks below a threshold and write dirty pages back to disk to free memory. The Oracle DBWR processes perform a batch write to make clean buffer frames available. We expect that the effect of pulling page frames is negligible on hit rate of the DRAM buffer and the flash cache either positively or negatively.

5 Recovery in *FaCE*

When a system failure happens, it must be recovered to a consistent state such that the atomicity and durability of transactions are ensured. Two fundamental principles for database recovery are write-ahead logging and commit-time force-write of the log tail. The *FaCE* system is no different in that these two principles are strictly applied for database recovery.

When a dirty page is evicted from the DRAM buffer, all of its log records are written to a stable log device in advance. As far as the durability of transactions is concerned, once a dirty page is written to either the flash cache or disk, the page is considered persistently propagated to the database, as the flash memory drive used as a cache extension is nonvolatile. The nonvolatility of flash memory ensures that it is always possible for the *FaCE* recovery system to recover the latest copies from the flash cache after a system failure.

It is interesting to note that *FaCE* utilizes data pages stored in the flash cache to serve dual purposes, namely cache extension and recovery. Since *FaCE* ensures that data pages stored in the flash cache are always the same as or newer than the copies stored in disk, the flash cache can be used to improve the overall cache performance during the online database processing and to recover the system from a failure more quickly. In this section, we present the recovery mechanism of *FaCE* that achieves transaction atomicity and durability at a nominal cost and recovers the database system from a failure quickly by utilizing data pages survived in the flash cache.

5.1 Checkpointing the flash cache

If a database system crashes while operating with *FaCE* enabled, there is an additional consideration that needs to be given so that the standard restart steps can restore the database to a consistent state. Some of the data pages stored in the flash cache may be newer than the corresponding disk copies, and the database consistency can only be restored by using those flash copies. Even if they were fully synced, the flash copies should be preferred to the disk copies, because the flash copies will help the system restart more quickly.

The only problem in doing this though is that we must guarantee the information about data pages cached in the flash memory survives a system failure. Otherwise, with the information lost, the flash copies of data pages will be inaccessible when the system restarts, and the database may not be restored to a consistent state. Of course, it is not impossible to restore the meta-data by analyzing the entire flash cache but it will be a time-consuming process. For example, it will take about 50 s to scan a flash cache of 20 GB capacity at 400 MB/s read speed. Obviously, as the capacity of a flash cache increases, the time taken to analyze it will increase proportionally. In contrast, as will be depicted below in this section, the checkpointing mechanism of *FaCE* can limit the analysis time within 1 or 2 s regardless of the capacity of a flash cache.

One practice common in most database systems is to include additional meta-data (e.g., file and block identification numbers and `pageLSN`) in the individual page header [18]. However, adding the information about whether it is cached in the flash to the page header is not an option either, because it will incur too much additional disk I/O to update the disk-resident page header whenever a page enters or leaves the flash cache.

One remedy proposed by the temperature-aware caching (TAC) [9] is to maintain the meta-data current persistently in the flash memory. The meta-data are maintained in a data structure called a slot directory that contains one entry for each data page stored in the flash cache. TAC uses flash memory as a write-through cache and relies on an invalidation–validation mechanism to keep the flash cache consistent with disk at all times. One obvious drawback of this approach is that an entry in the slot directory needs be updated for each page entering the flash cache. This overhead will not be trivial, because updating an entry requires two additional random flash writes—one for invalidation and another for validation.

In principle, this burden will be shared by any LRU-based flash caching method that maintains meta-data persistently in the flash memory [13]. This is because it needs to update an entry in the meta-data directory for each page being replaced in the flash cache, incurring just as much overhead as TAC does. This overhead will be significant and unavoidable, because the LRU replacement selects any page in the flash cache for replacement and, consequently, updating meta-data entries will have to be done by random write operations.

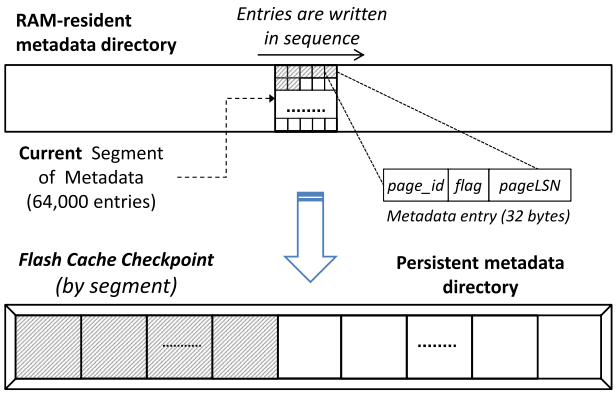

Fig. 3 Meta-data checkpointing in *FaCE*

Fortunately, however, *FaCE* has an effective way of dealing with the overhead since it relies on the *mvFIFO* replacement instead of LRU. In a similar way to how a database log tail is maintained, meta-data changes are collected in memory and written persistently to flash memory in a single large segment. This type of meta-data management is feasible with *FaCE*, because a data page entering the flash cache is always written to the rear in chronological order, and so is its meta-data entry to the directory. Therefore, meta-data updates will be done more efficiently than doing them for individual entries (typically tens of bytes each). This process is illustrated in Fig. 3.

Of course, this memory-resident portion of the meta-data directory will be lost upon a system failure, but it can be restored quickly by scanning only a small portion of the flash cache that corresponds to the most recent segment of meta-data. We call these meta-data flushing operation *flash cache checkpointing*, as saving the meta-data persistently (about one and a half MBytes each time in the current implementation of *FaCE*) is analogous to the database checkpointing. The flash cache checkpointing is triggered independently of the database checkpointing. Recovering the meta-data is described in more detail in the next section.

5.2 Database restart

When a database system restarts after a failure, the first thing to do is to restore the meta-data directory of the flash cache at the time of the failure. While the vast majority of meta-data entries are stored in flash memory and survive a failure, the entries in the current segment are resident in memory and lost upon a failure. To restore the current segment of the meta-data directory, the data pages whose meta-data entries are lost need be fetched from the flash cache. Those data pages can be found at the rear end of the flash cache maintained as

a circular queue. The front and rear pointers are maintained persistently in the database control file. The data pages in the flash cache contain all the necessary information such as page id and pageLSN in their page header.

In theory, the meta-data directory can be restored by fetching the data pages belonging only to the latest segment from the flash cache. However, this will require the database system to be quiesced while flushing the current segment of meta-data is in progress, and its negative impact on the performance will not be negligible. In the current implementation of *FaCE*, we allow a new meta-data entry to enter the current segment in memory, even when the previous segment is currently being flushed to flash memory. Considering the fact that a failure can happen even in the midst of flushing meta-data, the current implementation of *FaCE* restores the meta-data directory by fetching data pages belonging to the *two most recent segments* of the directory from the flash cache. This way we can avoid quiescing the database system and improve its throughput at minimally increased cost of restart. In fact, as will be shown in Sect. 6.8, *FaCE* can shorten the overall restart time considerably, because most of the redo recovery can be carried out by utilizing data pages cached persistently in flash memory.

6 Performance evaluation

We have implemented the *FaCE* scheme in the PostgreSQL 8.4 database server to demonstrate its effectiveness as a flash cache extension for database workloads. TPC-C benchmark tests were carried out on a hardware platform equipped with a RAID-0 array of enterprise-class 15k-RPM disk drives and consumer-grade SSD and datacenter SSD.

6.1 Prototype implementation

The *FaCE* scheme has been implemented as an addition to the buffer manager, recovery, and checkpointing modules of PostgreSQL. The most relevant functions in the buffer manager module are `buffer allocator` and `flush buffer`. The `buffer allocator` is called upon DRAM buffer misses, and it handles making free buffer for the requester. In addition, `flush buffer` is called to flush dirty buffer to the database if necessary. In the *FaCE* scheme, `flush buffer` is also used to flush a clean buffer, which was discarded when selected as victim buffer, as well as dirty one. We have modified these modules to incorporate the *mvFIFO* and its optimization strategies of *FaCE* in the buffer manager. For database checkpointing, we have modified the checkpoint module, so that all the dirty pages in the DRAM buffer are flushed to the flash cache instead of disk. A new recovery module for the mapping meta-data has been added.

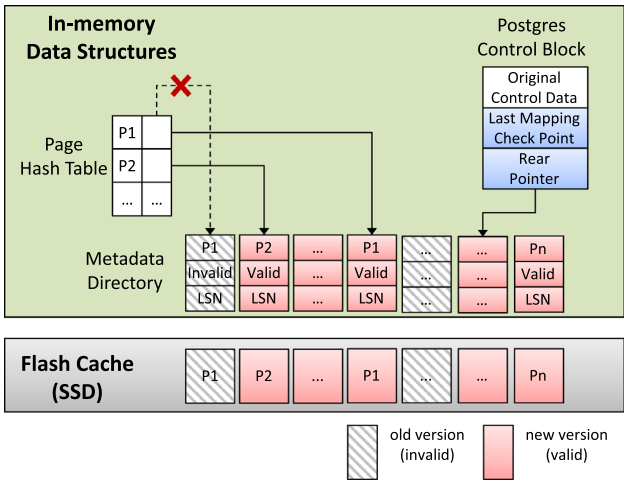

Fig. 4 In-memory data structures for *FaCE*

Besides, we have added a few data structures to manage page frames in the flash cache. Among those are a directory of meta-data maintained for all the data pages cached in flash memory and a hash table that maps a page id to a frame in the flash cache. The directory of meta-data is maintained as an array of entries (FIFO) or LRU queue depending on the choice of a management policy. There is an entry for each page cached in the flash memory, and each entry stores the page id, frame id, status flag, LSN, and checksum. Much like a hash table for the DRAM buffer frames, this hash table is used to determine whether a page is cached in flash memory and to find the frame if it is cached.

Both the meta-data directory and the hash table are resident in memory, but a persistent copy of the former is maintained in flash memory as well for recovery purposes. (See Sect. 5.1 for details.) Since the hash table is not persistent, it is rebuilt from the meta-data when the database system restarts. These in-memory data structures are illustrated in Fig. 4. The rear pointer of the flash cache is maintained in the control block of PostgreSQL and is written persistently as part of meta-data checkpointing, incurring no additional I/O operations.

6.2 Experimental setup

The TPC-C benchmark is a mixture of read-only and update intensive transactions that simulate the activities found in OLTP application environments. The benchmark tests were carried out by running PostgreSQL with *FaCE* enabled on a Linux system with two socket 2.6 GHz Intel Xeon X5650 processor. This computing platform was equipped with 74 GB DRAM, two commodity SSDs (Samsung 850 Evo 250 GB, Samsung 840 Pro 256 GB), two enterprise-class datacenter SSDs (Intel DC S3500 480 GB, Intel P3700 400 GB), and a RAID-0 disk array with 16 drives. The RAID

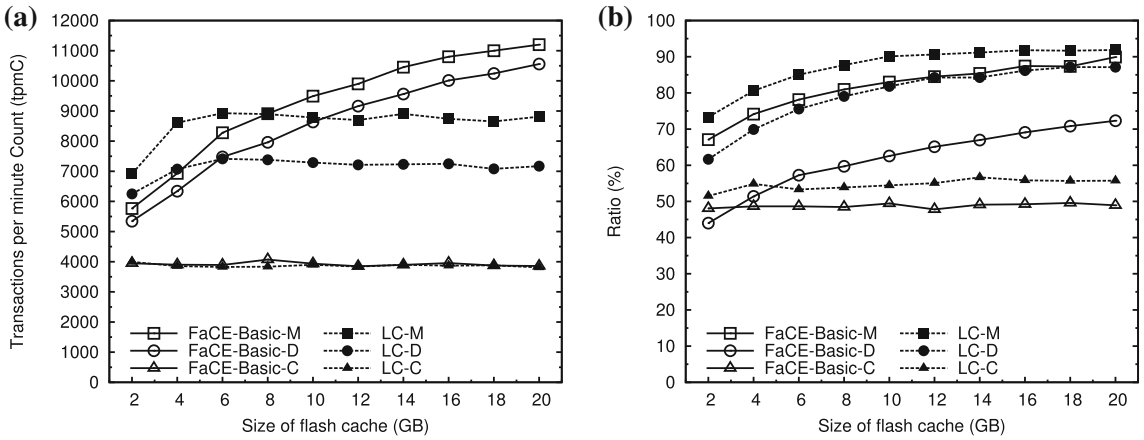

Fig. 5 Transaction throughput and hit ratio by different options for caching pages in *FaCE-Basic*: Postfix 'M,' 'C,' and 'D' denote 'Mixed,' 'CleanOnly,' and 'DirtyOnly,' respectively. **a** Transaction throughput, **b** hit ratio

controller was Intel RS2WG160 with PCIe 2.0 interface and 512 MB cache, and the disk drives were an enterprise class 15k-RPM Seagate ST3146356SS with 146.8 GB capacity each and a serial attached SCSI (SAS) interface.

The database size was set to approximately 50 GB (scale of 500, approximately 57 GB including indexes and other data files), and the DRAM buffer pool was limited to 200 MB in order to amplify I/O effects for a relatively small database. The number of concurrent clients was set to 50, which was the highest level of concurrency achieved on the computing platform before hitting the scalability bottleneck due to contention [22]. The page size of PostgreSQL was 4 kB as well as flash cache. The benchmark database and workload were created by the BenchmarkSQL tool [29].

For steady-state behaviors, all performance measurements were made after the flash cache was fully populated with data pages. Both the flash memory and disk drives were bound as a raw device, and 'Direct IO' flag was set in for the data files so that interference from data caching by the operating system was minimized.

6.3 Caching clean or dirty pages

First of all, as pointed out in Sect. 4.2, the benefit of caching a page in flash memory may vary depending on whether it is clean or dirty. To measure the differential caching effect of clean and dirty pages, we ran benchmark tests with three different options for caching pages in flash memory: (1) clean pages only, (2) dirty pages only, and (3) mixed of clean and dirty pages.

The performance trends observed in this set of experiments are shown in Fig. 5a. Not surprisingly, for both *FaCE* and LC, the best transaction throughput was yielded by the third option `mixed`, which was followed by the second option `dirty only` and then by the first option `clean only`. Note that *FaCE* was used with the GSC optimization disabled in order to isolate the effect of caching different types of pages. The throughput crossover between *FaCE* and LC occurred at around 8 GB of flash cache only because the GSC optimization was disabled. When GSC was enabled, as shown in Fig. 8b, *FaCE-GSC* always outperformed LC in all cache sizes. This trend in transaction throughput can be in part accounted for by another similar trend observed for the three options with respect to flash hit rates, as shown in Fig. 5b.

Table 4 Disk write reduction

(Measured in %)	Flash cache size				
	4 GB	8 GB	12 GB	16 GB	20 GB
Mixed	51	61	66	71	75
Dirty only	53	62	67	71	74

When it comes to *write reduction*, however, the outcome was quite intriguing. By write reduction, we mean disk writes eliminated by the flash cache that would be required for all dirty pages being evicted from the buffer pool without it. Formally, it is computed by $1 - \frac{W_{\text{disk}}}{D_{\text{flash}}}$, where W_{disk} denotes the number of pages written to disk and D_{flash} denotes the number of dirty pages evicted from the buffer pool. Obviously, a higher write reduction is expected to improve transaction throughput, because costly random writes to disk can be avoided.

Table 4 shows that the two options, *Dirty Only* and *Mixed*, were almost identical with respect to write reduction. Given that a disk write can be eliminated only when a dirty page cached in flash memory is overwritten or invalidated by an incoming copy, there is only one way this seemingly unintuitive result can be explained. Despite the reduced population of dirty pages in the flash cache, it was still large enough to be hit again once or more by highly skewed writes common

Table 5 Percentage of flash used to store old versions

Database size	Flash cache size				
	4 GB	8 GB	12 GB	16 GB	20 GB
108 GB	8.3	12.3	14.7	18.1	22.2
57 GB	10.0	15.5	23.9	35.3	44.9
21 GB	19.2	29.7	47.8	56.9	58.8

in the TPC-C workload [19]. Consequently, caching clean pages in addition to dirty ones in flash memory improved hit rates without sacrificing write reduction significantly.

The other option *Clear only* is not included in Table 4, as caching clean pages only does not reduce the amount of disk writes at all. This is because all dirty pages evicted from the RAM buffer were written to disk without being staged in the flash cache. For the reasons, *FaCE* was run with the *Mixed* option to cache both clean and dirty pages for all the experiments reported in this section.

The *mvFIFO* of *FaCE* allows multiple versions of the same page to be stored in the flash cache. To understand how detrimental this might be to storage utilization, we measured the portion of the flash cache capacity being used to store old versions. The measurements given in Table 5 are the fractions of varying cache capacities consumed to store old versions of cached pages. The measurements were obtained for three different database sizes when *FaCE* was run with the GSC optimization. The databases of different sizes were created by setting the number of warehouses to 1000, 500 (the default configuration), and 200, respectively. As given in Table 5, as the size of a database decreased, the percentage of old version pages increased in the flash cache. Consequently, the storage utilization of *FaCE* was not as high as a caching method (e.g., LC) that stores only a single copy of each cached page. However, as will be shown in Sect. 6.4, *FaCE* achieves much higher I/O throughput than LC.

6.4 Transaction throughput: *FaCE* versus LC

In this subsection, we analyze the performance impact of *FaCE* with respect to transaction throughput as well as hit rate and I/O utilization. We measured I/O utilization by the `iostat` utility on the Linux platform. In particular, we collected the `%util` statistics reported from `iostat`, which means 'percentage of CPU time during which I/O requests were issued to the device (bandwidth utilization for the device).'

To demonstrate its effectiveness, we compare *FaCE* with the Lazy Cleaning (LC) method [14], which is one of the most recent flash caching strategies based on LRU replacement. In order to evaluate the effect of group second chance (GSC) optimization, we present the performance measurements of *FaCE* under two configurations, with and without it. They are denoted by *FaCE-Basic* and *FaCE-GSC*, respectively, in this section. For *FaCE-GSC*, the write batch size was set to 256 kB (i.e., 64 4 kB pages).

Besides, in order to analyze the effect of different types of flash memory SSDs, we used both commodity (Samsung 840 Pro) and datacenter (Intel DC S3500) SSDs in the experiments. (Refer to Table 1 for the characteristics of the SSDs.) Both LC and *FaCE* built into the PostgreSQL database server implement the *write-back* policy by copying dirty pages to disk when they are evicted from the flash cache and cache both clean and dirty pages evicted from the DRAM buffer.

6.4.1 Read hit rate and write reduction

Figure 6 compares LC and *FaCE* with respect to read hits and write reductions observed in the TPC-C benchmark using the Samsung 840 Pro SSD.

As the size of the flash cache increased, both the hit rates and write reductions increased in all cases, because it became increasingly probable for warm pages to be referenced again, overwritten or invalidated before they were staged out of the flash cache.

Figure 6a shows the hit rate of *FaCE-GSC* was lower than that of LC by no more than 3 % due to the locality of references in the TPC-C workloads.

Not surprisingly, write reduction in LC shown in Fig. 6b were approximately 3–15 % higher than those of *FaCE* consistently over the entire range of the flash cache sizes tested. This was because, when the flash cache is managed by LC, the flash cache keeps no more than a single copy for each cached page, and the copy is always up-to-date. Thus, LC could utilize the space of the flash cache more effectively than *FaCE*, which could store more than a single copy for a cached page. Despite such duplicate pages, however, as Fig. 6a shows, the hit rate of *FaCE* was not lower than that of LC by more than 10 % due to the locality of references in the TPC-C workloads. Another point to note is that the group second chance (GSC) improved not only read hits but also write reductions for *FaCE* by giving dirty pages a second chance to stay and to be invalidated in the flash cache rather than being flushed to disk.

From Fig. 6, it is clear that the effect of GSC optimization is significant. The group second chance (GSC) improved not only read hits but also write reductions for FaCE by giving dirty pages a second chance to stay and to be invalidated in the flash cache rather than being flushed to disk. Besides, the batch write of 64 4 kB pages in *FaCE-GSC* is quite effective in improving the device utilization and I/O throughput.

6.4.2 Storage I/O utilization and throughput

Keeping a single copy for each cached page by LRU often requires overwriting an existing copy—either an old version

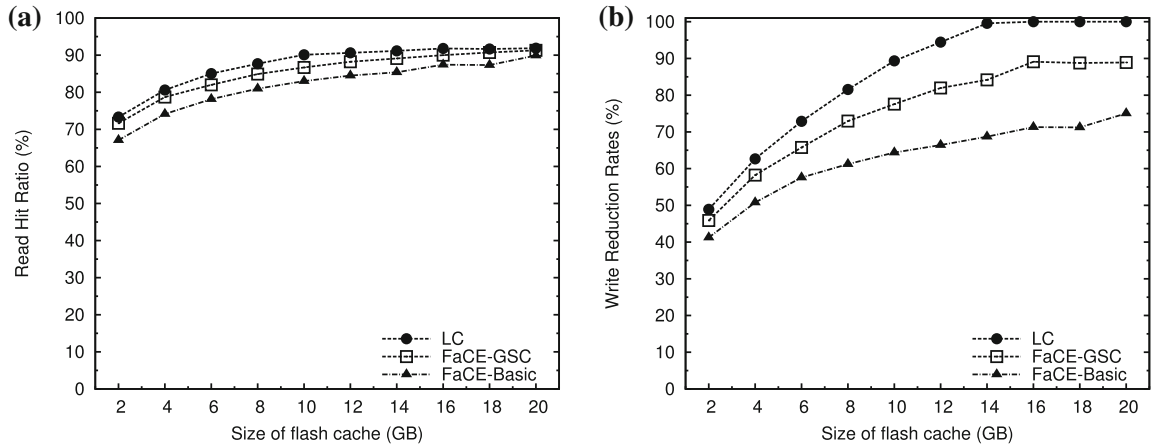

Fig. 6 Read hit and write reduction rates in flash cache (Samsung 840 Pro SSD)

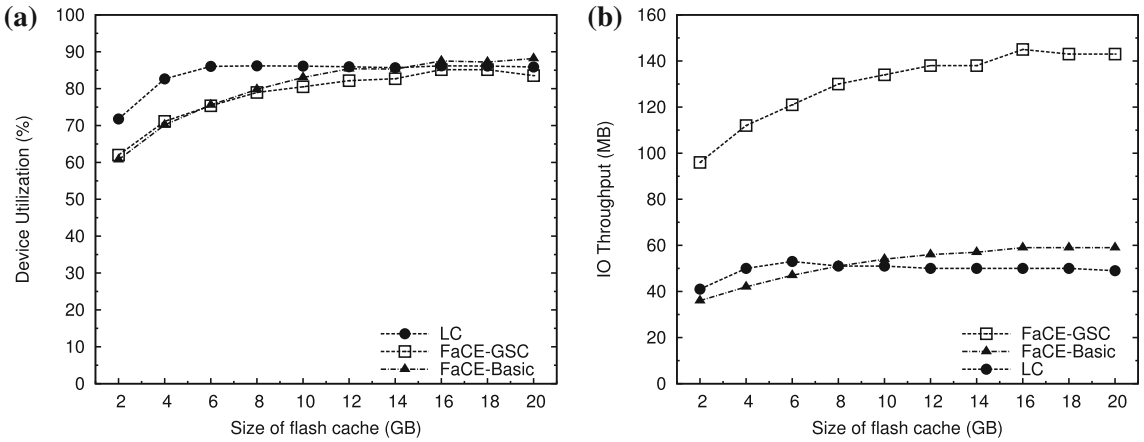

Fig. 7 Storage I/O utilization and throughput (Samsung 840 Pro SSD)

or a victim page—in the flash cache. Either way overwriting a page in the flash cache will incur a random flash write. Figure 7 compares LC and *FaCE* with respect to average utilization and I/O throughput of the flash cache. Figure 7a shows that LC raised the device saturation of a flash cache quickly to 80 % or higher. As the size of flash cache increased in *FaCE*, the average utilization was also linearly increased.

Figure 7b compares LC and *FaCE* with respect to throughput carried out per second. The results given in figure clearly reflect the difference between LC and *FaCE* in device saturation of the flash cache. *FaCE* processed I/O operations more efficiently than LC by more than a factor of two when the size of flash cache was 10 GB. This was because the write operations were dominantly random by LRU while they were dominantly sequential by *FaCE*. Another important point to note is that the I/O throughput of *FaCE* improved consistently and considerably as the size of flash cache increased. However, the I/O throughput of LC does not improve as the size of flash cache increased.

6.4.3 Impact on transaction throughput

Figure 8 compares LC and *FaCE* with respect to transaction throughput measured in the number of transactions processed per minute (tpmC). In order to understand the scope of performance impact by flash caching, we included the cases where the database was stored entirely on either a flash memory SSD (denoted by `SSD-only`) or a disk array (denoted by `HDD-only`).

Figure 8a shows the transaction throughput obtained by using a low-grade commodity SSD, Samsung 850 Evo, as a caching device (or as a main storage medium for the `SSD-only` case). The throughput of `SSD-only` was merely about 70 % higher than that of `HDD-only`. This is because, as given in Table 1, the low random write performance of the TLC-based SSD became a bottleneck in transaction processing. The LC method, which relies on random write operations, also yielded almost the same performance trend for the same reason. On the other hand, *FaCE-Basic* or *FaCE-GSC* yielded much higher throughput

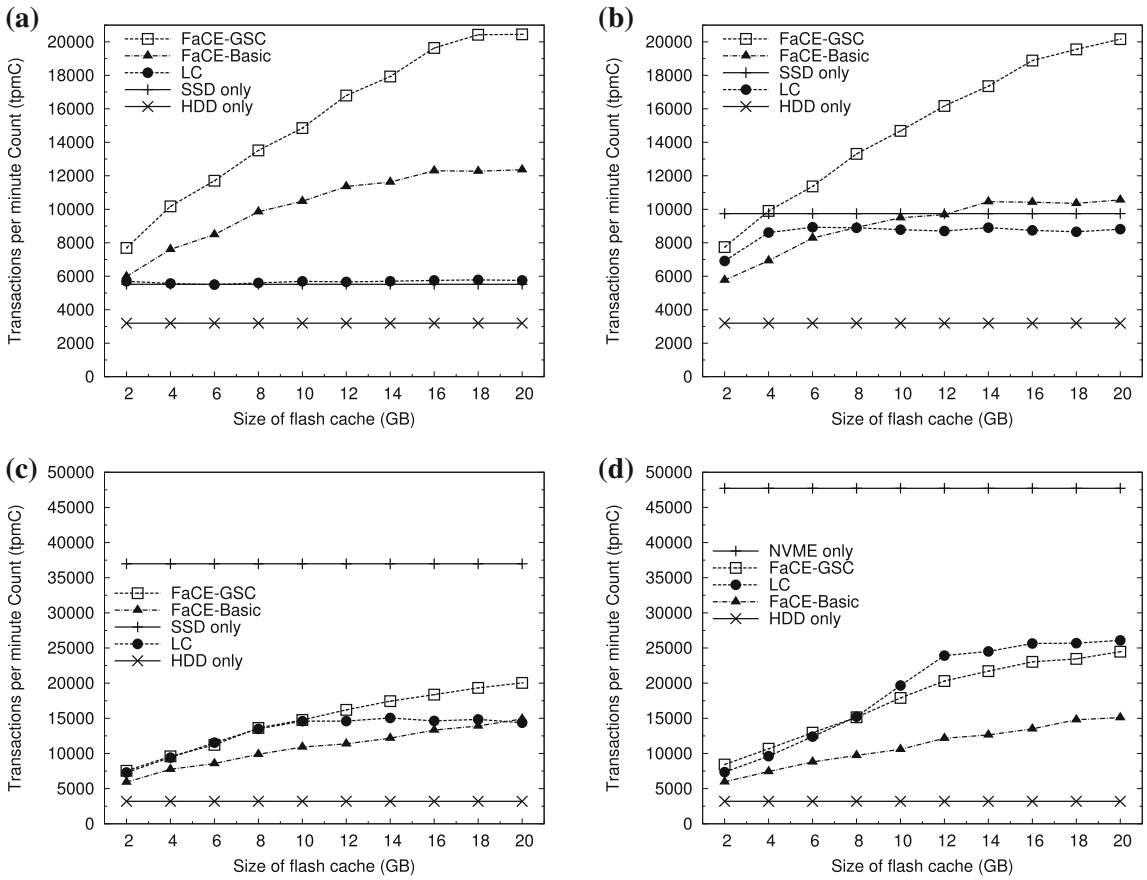

Fig. 8 Transaction throughput: *FaCE* versus LC. **a** Samsung 850 EVO SSD, **b** Samsung 840 Pro SSD, **c** Intel DC S3500 SSD, **d** Intel NVMe P3700

consistently over the entire range of a flash cache size without being limited by the low random write throughput of the caching device. This result demonstrates that *FaCE* could take advantage of a small amount of flash memory to deliver even higher throughput than using a large amount of flash memory to store the entire database tables.

Figure 8b shows the transaction throughput obtained by using another commodity SSD, Samsung 840 Pro, as a caching device. Under the LC method, the transaction throughput remained flat without any significant improvements with increases in the size of flash cache after 4 GB. This is because the commodity SSD was already utilized at its maximum (70–86 % as shown in Fig. 7a), and the saturation in the flash caching device became a performance bottleneck quickly as more data pages were stored in the flash cache. Under the *FaCE* method, in contrast, the utilization of flash caching device was lower than LC method consistently. As the size of the flash cache increased, the transaction throughput continued to improve over the entire range without being limited by the I/O throughput of the flash caching drive.

With a datacenter SSD, as shown in Fig. 8c, almost identical trend was observed in transaction throughput by *FaCE*. The LC method, on the other hand, improved transaction throughput and reduced the performance gap considerably. This was due to the higher random write throughput of the datacenter SSD. With the large amount of flash cache, however, *FaCE* still outperformed LC about 40 %. As pointed out in Sect. 2, the bandwidth disparity between random writes and sequential writes still exists even in the datacenter-type SSDs, and *FaCE* is apt to take advantage of it better by turning random writes into sequential ones.

Another point to note in Fig. 8 is that the performance gap between *FaCE-Basic* and *FaCE-GSC* widened as the size of flash cache increased. When the size of flash cache was 20 GB, *FaCE-GSC* delivered approximately twice higher throughput than *FaCE-Basic*. The significant effect of GSC optimization was due to the combination of high hit ratio, high write reduction (Fig. 6), high device utilization, and high I/O throughput (Fig. 7). LC performed comparably with or slightly better (within 10 %) than *FaCE-GSC* only for Intel P3700 (Fig. 8d), which is a high-end NVMe SSD released recently. This is because Intel P3700 processes random writes almost as fast as sequential writes.

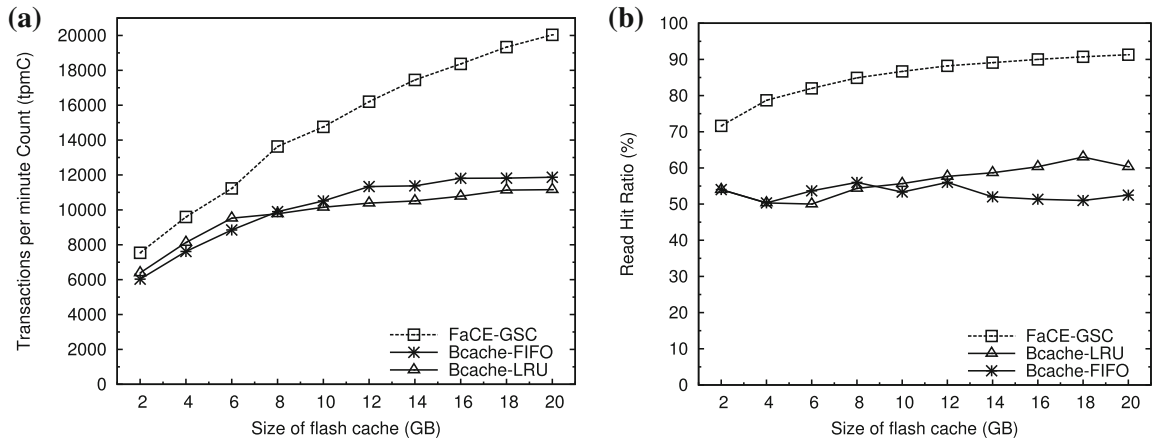

Fig. 9 Transaction throughput: *FaCE* versus Bcache (Samsung 840 Pro SSD). **a** Transaction throughput, **b** cache hit ratio

In summary, while LC achieved hit rates and write reductions better than *FaCE* and reduced the amount of traffic to disk drives, *FaCE* was superior to LC in utilizing the flash cache efficiently for higher I/O throughput. Despite the trade-off, it turned out the benefit from higher I/O bandwidth of flash cache outweighed the benefit from reduced I/O operations of disk. Overall, *FaCE* outperformed LC in transaction throughput considerably irrespective of the type of a flash memory device used.

Table 6 Sequentiality and average write size by different caching schemes: cache size 10 GB

Caching scheme	Sequentiality (%)	Avg write size (kB)
FaCE	20.59	256.3
LC	1.36	4.0
Bcache (FIFO)	10.89	4.6
Bcache (LRU)	8.83	5.2

6.5 Transaction throughput: *FaCE* versus Bcache

This section evaluates the performance of *FaCE* in comparison with Bcache described in Sect. 3.3. Bcache was run in the write-back sync mode, and both LRU and FIFO were tested as a cache replacement policy. For fair comparison with *FaCE*, the cache device was mounted to the data directory of PostgeSQL so that only data pages would be cached. From this section on, the cache device was a Samsung 840 Pro SSD, and the backing device was a RAID-0 array with 16 disk drives.

Figure 9 compares *FaCE* and Bcache with respect to transaction throughput and cache hit ratio. In Fig. 9a, *FaCE* consistently outperformed Bcache in terms of tpmC regardlessly of cache sizes and replacement policies of Bcache. When the cache size was 20 GB, *FaCE* yielded about 68 % higher throughput than Bcache. This wide performance gap was attributed to different write patterns produced by *FaCE* and Bcache. Although they share the same goal of minimizing random writes, Bcache did not fulfill it satisfactorily and had to deal with many random writes for OLTP workloads. This will be explained later in more detail.

Another reason for the performance gap was due to the difference in the cache hit ratio. As shown in Fig. 9b, *FaCE* achieved about 20 % higher hit ratio than Bcache. Cache hit ratio was measured in the buffer manager of PostgreSQL for *FaCE*, while it was measured in the block layer of OS kernel for Bcache. We believe that the on-entry caching policy of Bcache is responsible for the low hit ratio. Bcache stores a clean page in the flash cache when the page is fetched from the backing device to the RAM buffer. The page cached in the flash memory will not be in use until the page frame gets evicted from the RAM buffer. If the page is dirty on eviction, the clean copy in the flash cache will be invalidated, because the dirty but newer version of the page will enter the flash cache. The clean copy cached in flash memory will get used only when the page is evicted from the RAM buffer as clean. This will lower the chance of utilization for the pages cached in flash memory.

To understand the write patterns made into the flash cache, we analyzed the I/O traces produced by each caching scheme by using the blktrace tool [3]. Table 6 compares the sequentiality and average write sizes of the caching schemes. The write sequentiality was measured by counting the number of write operations requested in a consecutive region in the address space. As given in Table, *FaCE* achieves considerably higher sequentiality than the other caching schemes as well as almost 50 times larger write size.

Bcache maintains up to six active buckets by default at any moment in time. When each bucket is considered separately, all writes are made in the buckets sequentially. However, when all six buckets are considered altogether, the pattern of

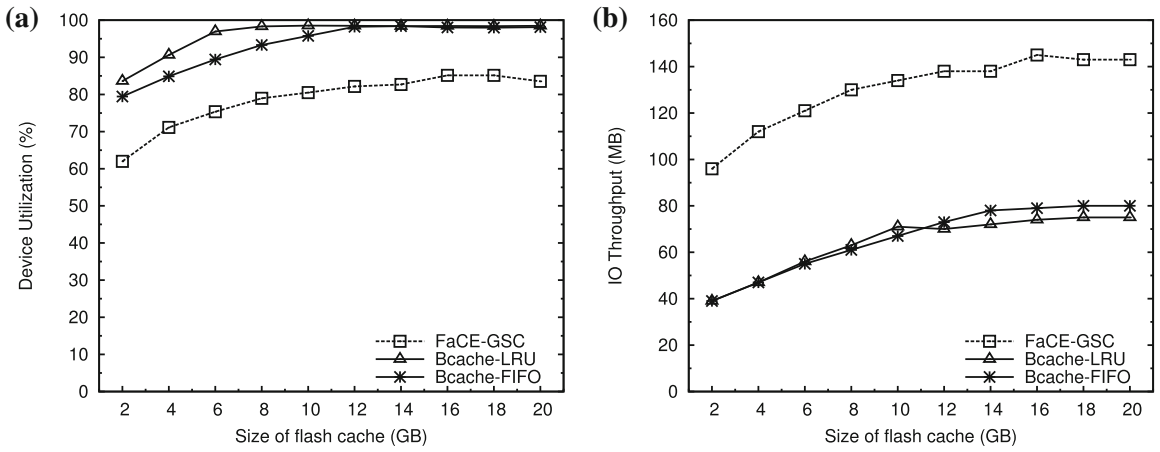

Fig. 10 Storage IO utilization and throughput: *FaCE* and Bcache (Samsung 840 Pro SSD)

writes is not so sequential, as the writes in the six streams will be interspersed with one another. The average size of a write was rather small at about 4.6 kB.

In stark contrast, *FaCE* produced a sequential write pattern, as is clearly shown in Table 6. This is because *FaCE* carries out writing multiple pages in a large batch. The average size of a write was about 256 kB for *FaCE*. In the case of LC, randomness was even worse and the write pattern was completely random.

Figure 10a, b compares *FaCE* and Bcache with respect to I/O utilization and I/O throughput. As the size of flash cache increased, the processing capacity of an SSD being used as a flash cache was consumed much more quickly with Bcache and maxed out eventually. With *FaCE*, the utilization of the SSD stayed comfortably below 90 % at all times. This observation concurs well with the I/O throughput of *FaCE*, which was almost twice higher than that of Bcache.

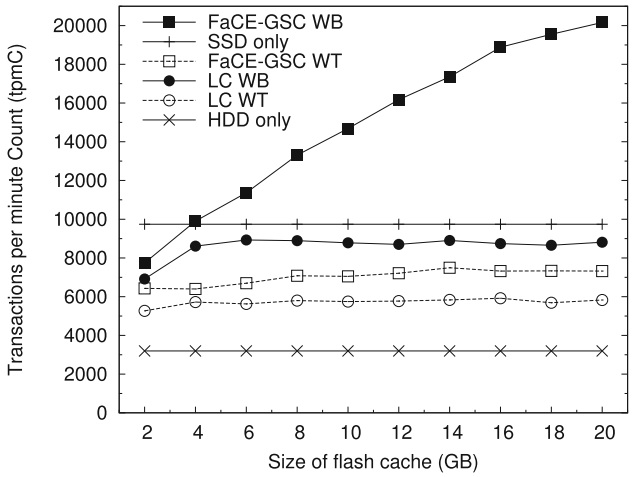

Fig. 11 Transaction throughput: Write-Back(WB) versus Write-Through (WT)

6.6 Write-through sync policy for reliability

This subsection analyzes the impact of sync policies, described in Sect. 4.2, on the transaction throughput. If write-through is chosen as a sync policy, a dirty page evicted from the DRAM buffer pool is written to a caching device (i.e., SSD) and a backing device (i.e., disk drive) simultaneously using libaio's `io_submit` system call. The requesting thread waits until the both operations are finished.

The main advantage of the write-through sync policy is that the consistent state of database can be maintained in the presence of caching devices. However, it is also evident that the write-through sync policy entails non-trivial overhead of syncing data between the caching and backing devices at all times. As shown in Fig. 11, the throughput from both *FaCE-GSC* and LC degenerates significantly when the sync policy was switched from write-back to write-through. Even worse, the throughput increase saturates in the early stage, as the overall system becomes bounded by the slow I/O performance of a backing device (i.e., disk) that must flush every single dirty page evicted from the buffer pool.

Even with the write-through sync, *FaCE* outperformed LC approximately 20 % in terms of transaction throughput. This is mainly because *FaCE* achieves higher write performance by group replacement, which enables evicting multiple pages from the DRAM buffer pool by a single system call (`io_submit`).

6.7 Cost-effectiveness and scalability

This subsection evaluates empirically the cost-effectiveness of the flash cache and its impact on the scalable throughput of a disk array.

6.7.1 Write amplification under OLTP workload

In order to evaluate the effect of caching methods on the durability of SSDs, we measured the wearout of SSDs while

Table 7 Comparison of WAF according to caching scheme

Storage media	*FaCE-GSC*	LC
Samsung 840 Pro SSD	1.02	5.13
Intel DC S3500	1.08	2.47

running the TPC-C benchmark with *FaCE-GSC* and LC. The size of a flash cache was set to 10 GB, and the WAF values were collected every 10 s during the benchmark, which was run for 24 h. The benchmark was repeated with the two caching methods using two different commodity SSDs.

The average WAF values in Table 7 clearly show that *FaCE-GSC* wore out the SSDs more slowly than LC by about 5 times with Samsung 840 Pro and about 2.3 times with Intel DC S3500. Note that the WAF values shown in Table 7 are surprisingly close to those in Table 2 of Section 2. This is because the average write size of *FaCE-GSC* was 256.3 kB while that of LC was 4.0 kB. Refer to Table 6 for the average write sizes of the two caching methods.

6.7.2 More DRAM or more flash

A database buffer pool is used to replace slow disk accesses with faster DRAM accesses for frequently accessed database items. In general, investing resources in the DRAM buffer is an effective means to improve the overall performance of a database system, because a larger DRAM buffer pool will reduce page faults and increase the throughput. As the size of a DRAM buffer increases, however, the return on investment will not be sustained indefinitely and will eventually saturate after passing a certain threshold. For example, with TPC-C workloads, the DRAM buffer hit rate is known to reach a knee point when its size is quite a small fraction of the database size due to skewed data accesses [38].

As the analysis given in Sect. 2.3 indicates, the return on investment is likely to be much higher with flash memory than DRAM given the persistent and widening price gap between them. To demonstrate the cost-effectiveness of flash memory as a cache extension, we measured the performance gain that would be obtained by making the same amount of monetary investment to DRAM and flash memory.

Assuming that the price per gigabyte of commodity SSD is approximately ten times lower than DRAM [15], we evaluated the cost-effectiveness by measuring the throughput increment obtained from each 2 GB of flash memory or alternatively each 200 MB of DRAM added to the basic system configuration described in Sect. 6.2. As shown in Fig. 12, the transaction throughput (or return on investment) was consistently higher with a wide margin when more resources were directed to flash memory rather than DRAM.

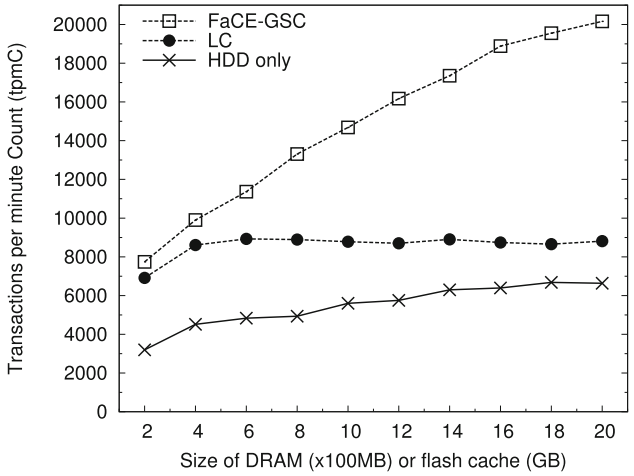

Fig. 12 16 HDDs versus larger DRAM versus *FaCE*

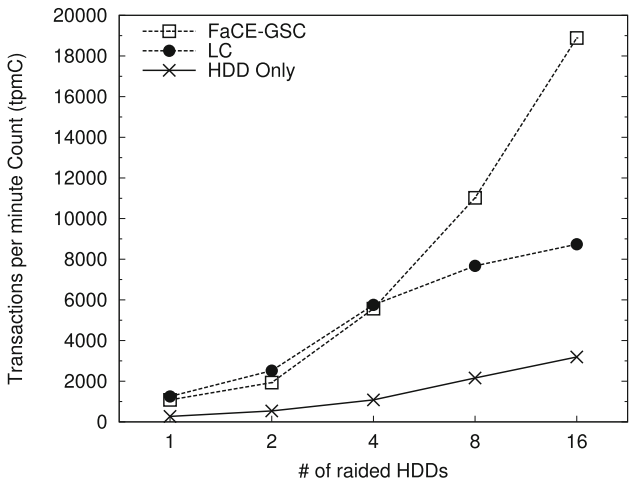

Fig. 13 Effect of a disk array size: cache size 16 GB

6.7.3 Scale-up with more disks

No matter how large a DRAM buffer or a flash cache is, there will be cache misses if neither is as large as the entire database. For the reason, the I/O throughput of disk drives will always remain on the critical path of most database operations. For example, when a data page is about to enter the flash cache, another page may have to be staged out of the flash cache if it is already full. In this case, a page evicted from the DRAM buffer will end up causing a disk write followed by a flash write. This acutely points out the fact that the system throughput may not be improved up to its full potential by the flash cache alone without addressing the bottleneck (i.e., disks) on the critical path.

Using a disk array is a popular way of increasing disk throughput (measured in IOPS). Figure 13 compares LC and *FaCE* with respect to transaction throughput measured with a varying number of disk drives from one to sixteen. The

flash cache size was set to 16 GB for both LC and *FaCE*, and `HDD-only` was also included in the scalability comparison.

For *FaCE* and `HDD-only`, transaction throughput increased consistently as the number of disk drives increased. This confirms that disk drives were on the critical path, and increasing disk throughput was the key to improving the transaction throughput. (Note that the same database was distributed across all available disk drives). It also confirms that, with *FaCE*, the flash cache was never a limiting factor in transaction throughput even for the largest configuration we tested (i.e., 16 disk drives in the RAID-0 array). In contrast, throughput increase in LC started slowing down earlier at eight disk drives.

When the number of disk drives was very small (i.e., one or two), *FaCE* was outperformed by LC in transaction throughput, due mainly to the lower utilization of flash caching device. This result is consistent with the observations reported in the study of CAC [28], where only one disk drive was used as a backing device storing the entire database. However, it should be noted that, as clearly shown in Fig. 13, a single disk configuration is the most unfavorable setting for *FaCE*, which scales better than any existing flash caching method as the size of a disk array increases.

6.8 Performance of recovery

In order to evaluate the recovery system of *FaCE*, we crashed the PostgreSQL server using the Linux `kill` command and measured the time taken to restart the system. When database checkpointing was turned on, the `kill` command was issued at the midpoint of a checkpoint interval. For example, if the checkpoint interval was 200 s, the `kill` command was issued 100 s after the most recent checkpoint. In the case of *FaCE*, the *FaCE* (GSC) was chosen for the flash cache management, and the restart time included the time taken to restore the meta-data directory. The flash cache size was set to 10 GB.

Table 8 presents the average recovery time taken by the system when it was run with different checkpoint intervals 100, 200, and 300 s. For each checkpoint interval, we took the average of restart times measured from five separate runs. Across all three checkpoint intervals, *FaCE* reduced the restart time considerably—from about 1.7 to 9 times depending on the checkpoint intervals—over the `HDD-only` system without flash cache. Such significant reduction in restart time was possible because the redo recovery could be carried out by utilizing data pages cached persistently in flash memory. We observed in our experiments that more than 98 % of data pages required for recovery were fetched from the flash cache instead of disk. Another intriguing point is that *FaCE* recovers slightly faster than even the `SSD-only` system. This is because the random writes performed by a Postgres recovery thread are slower than the sequential writes performed by *FaCE*.

Table 8 Time taken to restart the system

(Measured in second)	Checkpoint intervals		
	100	200	300
FaCE-GSC	34	50	59
HDD-only	58	201	525
SSD-only	41	54	66

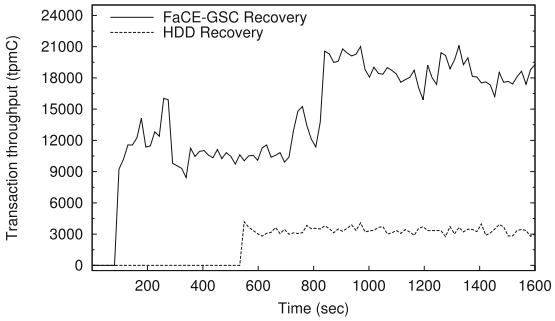

Fig. 14 Transaction throughput after the system restart

The restart times given in Table 8 for *FaCE* include the time taken to restore the meta-data directory, which was approximately 1.5 s on average regardless of the checkpoint interval. The meta-data directory can be restored by fetching the persistent portion of the directory from flash memory and by scanning as many data pages as two segments worth of meta-data entries from the flash cache. The latter is required to rebuild the most recent segment of the directory. In our experiments, the persistent portion of the meta-data directory was 80 MB, and the amount of data pages to read from the flash cache was 512 MBytes.[2] The sequential read bandwidth of the flash memory SSD used in our experiments was high enough—at least 400 MB/s—to finish this task within 1.5 s on average.

Figure 14 shows the time-varying transaction throughput measured immediately after the system was recovered from a failure. This figure clearly demonstrates that, when *FaCE* was enabled, the system resumes normal transaction processing much more quickly and maintains higher transaction throughput at all times. The checkpoint interval was set to 300 s in this experiment.

Figure 14 also shows that transaction throughput slumped to around 10,000 tmpC shortly after restart for about 400 s before reaching a steady and higher level of throughput. This

[2] Each segment contains 64,000 meta-data entries of 32 bytes each. Among the 40 segments required for a 10 GB flash cache, 38 of them are fetched directly from flash memory and the rest are rebuilt when the checksum of the data page and stored checksum in the meta-data directory are the same.

is because all the reference information of pages cached in flash memory was lost when the system crashed and *FaCE* had to run without the GSC optimization until it collected sufficient page references.

We also measured normal shutdown and restart times. When the database server shut down, *FaCE* took between 7 and 9 s to flush all the dirty pages from the 200 MB DRAM buffer to flash cache. In addition, *FaCE* needed to store the meta-data mapping information, but it took less than a second. When the database server restarted, *FaCE* helped it reach a steady state more quickly for the same reason presented for recovery.

7 Conclusion

This paper presents a low-overhead caching method called *FaCE* that utilizes flash memory as an extension to a DRAM buffer for a recoverable database. *FaCE* caches data pages in flash memory on exit from the RAM buffer. By basing its caching decision solely on the RAM buffer replacement, the flash cache is capable of sustaining high hit rates without incurring excessive run-time overheads for monitoring access patterns, identifying hot and cold data items, and migrating them between flash memory and disk drives. We have implemented *FaCE* and its optimization strategies within the PostgreSQL open-source database server and demonstrated that *FaCE* achieves a significant improvement in the transaction throughput.

We have also made a few important observations about the effectiveness of *FaCE* as a flash caching method. First, *FaCE* demonstrates that adding flash memory as a cache extension is more cost-effective than increasing the size of a DRAM buffer. Second, the optimization strategies (i.e., GSC) of *FaCE* indicate that turning small random writes to large sequential ones is critical to maximizing the I/O throughput of a flash caching device so as to achieve scalable transaction throughput. Third, the *mvFIFO* replacement of *FaCE* enables efficient and persistent management of the meta-data directory for the flash cache and allows more sustainable I/O performance for higher transaction throughput. Fourth, *FaCE* takes advantage of the nonvolatility of flash memory to minimize the recovery overhead and accelerate the system restart from a failure. Since most data pages needed during the recovery phase tend to be found in the flash cache, the recovery time can be shortened significantly.

Acknowledgments Sang-Won Lee and Bongki Moon are the corresponding authors of this paper. This research was supported in part by Institute for Information & communications Technology Promotion (IITP) (R0126-15-1088) and the IT R&D program of MKE/KEIT [10041244, SmartTV 2.0 Software Platform]. This work was also partly supported by the National Research Foundation of Korea (NRF) Grant (No. 2015R1A5A7037372) funded by the Korean Government (MSIP).

References

1. Ashdown, L., Kyte, T.: Oracle Database Concepts 11g Release 2. Oracle Corporation. https://docs.oracle.com/cd/E11882_01/server.112/e40540.pdf (2015)
2. Athanassoulis, M., Chen, S., Ailamaki, A., Gibbons, P.B., Stoica, R.: MaSM: efficient online updates in data warehouses. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pp. 865–876 (2011)
3. Axboe, J.: blktrace: Block Layer IO Tracing Tool. http://git.kernel.org/cgit/linux/kernel/git/axboe/blktrace.git
4. Axboe, J.: FIO (Flexible IO Tester). https://github.com/axboe/fio.git
5. Balakrishnan, M., Malkhi, D., Wobber, T., Wu, M., Prabhakaran, V., Wei, M., Davis, J.D., Rao, S., Zou, T., Zuck, A.: Tango: distributed data structures over a shared log. In: Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, SOSP '13, pp. 325–340 (2013)
6. Belady, L.A., Nelson, R.A., Shedler, G.S.: An anomaly in space–time characteristics of certain programs running in a paging machine. Commun. ACM **12**(6), 349–353 (1969)
7. Bernstein, P.A., Das, S., Ding, B., Pilman, M.: Optimizing optimistic concurrency control for tree-structured, log-structured databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, pp. 1295–1309 (2015)
8. Bernstein, P.A., Reid, C.W., Das, S.: Hyder—a transactional record manager for shared flash. In: CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9–12, Online Proceedings, pp. 9–20 (2011)
9. Bhattacharjee, B., Ross, K.A., Lang, C., Mihaila, G.A., Banikazemi, M.: Enhancing recovery using an SSD buffer pool extension. In: Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN '11, pp. 10–16 (2011)
10. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: An object placement advisor for DB2 using solid state storage. Proc. VLDB Endow. **2**(2), 1318–1329 (2009)
11. Canim, M., Mihaila, G.A., Bhattacharjee, B., Ross, K.A., Lang, C.A.: SSD bufferpool extensions for database systems. Proc. VLDB Endow. **3**(1–2), 1435–1446 (2010)
12. Chen, F., Lee, R., Zhang, X.: Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In: Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, HPCA '11, pp. 266–277 (2011)
13. Do, J., Zhang, D., Patel, J.M., DeWitt, D.J.: Fast peak-to-peak behavior with SSD buffer pool. In: 29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8–12, pp. 1129–1140 (2013)
14. Do, J., Zhang, D., Patel, J.M., DeWitt, D.J., Naughton, J.F., Halverson, A.: Turbocharging DBMS buffer pool using SSDs. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, SIGMOD '11, pp. 1113–1124 (2011)
15. DramExchange: Price Quites. http://www.dram-exchange.com/Price/NationalDramDetail.aspx
16. Fusion-IO: IOTurbine. http://www.fusionio.com/products/ioturbine
17. Gray, J., Fitzgerald, B.: Flash disk opportunity for server applications. Queue **6**(4), 18–23 (2008)
18. Gray, J., Reuter, A.: Transaction Processing: Concepts and Technique. Morgan Kaufmann, Burlington (1993)
19. Hsu, W.W., Smith, A.J., Young, H.C.: Characteristics of production database workloads and the TPC benchmarks. IBM Syst. J. **40**(3), 781–802 (2001)

20. Intel: Intel Cache Acceleration Software. http://www.intel.com/content/www/us/en/software/intel-cache-acceleration-software-performance.html
21. Intel: Solid-State Drives in Server Storage Applications. http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ssd-server-storage-applications-paper.pdf
22. Johnson, R., Pandis, I., Hardavellas, N., Ailamaki, A., Falsafi, B.: Shore-MT: a scalable storage manager for the multicore era. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology, EDBT '09, pp. 24–35 (2009)
23. Kang, W.H., Lee, S.W., Moon, B.: Flash-based extended cache for higher throughput and faster recovery. Proc. VLDB Endow. **5**(11), 1615–1626 (2012)
24. Kang, W.H., Lee, S.W., Moon, B., Kee, Y.S., Oh, M.: Durable write cache in flash memory SSD for relational and NoSQL databases. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, pp. 529–540 (2014)
25. Koltsidas, I., Viglas, S.D.: Flashing up the storage layer. Proc. VLDB Endow. **1**(1), 514–525 (2008)
26. Lee, S.W., Moon, B., Park, C.: Advances in flash memory SSD technology for enterprise database applications. In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data, SIGMOD '09, pp. 863–870 (2009)
27. Lee, S.W., Moon, B., Park, C., Kim, J.M., Kim, S.W.: A case for flash memory SSD in enterprise database applications. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data, SIGMOD '08, pp. 1075–1086 (2008)
28. Liu, X., Salem, K.: Hybrid storage management for database systems. Proc. VLDB Endow. **6**(8), 541–552 (2013)
29. Lussier, D., Martin, S.: The BenchmarkSQL Project. http://benchmarksql.sourceforge.net
30. Oracle Corporation: Oracle TPC Benchmark C Full Disclosure Report. http://c970058.r58.cf2.rackcdn.com/fdr/tpcc/Oracle_SPARC_SuperCluster_with_T3-4s_TPCC_FDR_120210.pdf (2010)
31. Overstreet, K.: bcache. http://bcache.evilpiepirate.org
32. Samsung: Samsung Solid State Drive White Paper. http://www.samsung.com/global/business/semiconductor/minisite/SSD/us/html/whitepaper/whitepaper.html
33. SanDisk: FlashSoft. http://www.sandisk.com/enterprise/flashsoft
34. Sears, R., Ramakrishnan, R.: bLSM: a general purpose log structured merge tree. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, pp. 217–228 (2012)
35. Srinivasan, M., Saab, P.: FlashCache. https://github.com/facebook/flashcache
36. STEC: EnhanceIO SSD Caching Software. https://github.com/stec-inc/EnhanceIO
37. Subramaniam, M.: Exadata Smart Flash Cache Features and the Oracle Exadata Database Machine. Oracle Corporation. http://www.oracle.com/technetwork/database/exadata/exadatasmart-flash-cache-366203.pdf (2013)
38. Tsuei, T.F., Packer, A.N., Ko, K.T.: Database buffer size investigation for OLTP workloads. In: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data, SIGMOD '97, pp. 112–122 (1997)
39. Willick, D., Eager, D., Bunt, R.: Disk cache replacement policies for network fileservers. In: Proceedings of the 13th International Conference on Distributed Computing Systems, pp. 2–11 (1993)
40. Zhao, M.: DM-cache cache target for device-mapper. https://github.com/mingzhao/dm-cache
41. Zhou, Y., Philbin, J., Li, K.: The multi-queue replacement algorithm for second level buffer caches. In: Proceedings of the General Track: 2001 USENIX Annual Technical Conference, pp. 91–104 (2001)