TDT4225 – Assignment 3

Kristoffer Dalby

1.

The ZFS Adaptive replacement cache (ARC) uses a combination of Most frequently used (MFU) and most recently used (MRU).

ARC consists of four lists:

- A list of the MRU pages
- A list of the MFU pages

It also includes two ghost lists, one for each list above. These lists are special in that they contain recently dropped pages from the other two lists.

- A list of the recently evicted pages from MRU
- A list of the recently evicted pages from MFU

Before we start, it is important to note that all the lists only contain pointers to the blocks in cache.

When the file system first reads a page/block from the file system it if placed in the MRU list and the cache. When we then read a different block, it is also put in the MRU and the cache.

If the first block is now re-read, it is moved over to the MFU list. As long as there is still space in the MRU, the block will still reside in both MRU and MFU.

To explain the ghost lists, we need to imagine that the cache and both the main lists are full.

When both lists are full, and we read a new block, which is not a MFU block, the Least recently used block in the MRU list will then be evicted and moved to the ghost list and out of the cache. The same thing happens when the least frequently used block is changed for a block with higher usage.

This makes the file system adapt to the change in type of workload, for example, if the system has a load that needs a lot of recently used files, the system will adapt to the challenge and use the MRU list for everything it is worth. If the workload is using frequently used files, the system will adapt to this.

In addition, the ZFS ARC expands the original ARC by using available memory for caching and it implements the possibility to lock pages/blocks to the cache so they wont be evicted.

2.

A Bloom filter is a data structure which is both probabilistic and space-efficient. The interesting property of the Bloom filter is that it can efficiently check if a element is in NOT in a set. In other words, one can check a Bloom filter and get the response “possibly in set” or “definitely not in the set”.

Example usage can be for example to filter out data that does not exist before accessing a data store. Let say we have a long list of social security numbers, where not all numbers is available in the database, and the cost to fetch information from the database is high, we can use a Bloom filter to remove all the numbers that are not of interest before actually querying the database.

3.

When a *Put* operation is issued to LevelDB, the program will use three main components (log, Memtable and SSTable). First, the data that is to be written is logged to disk. This is so the information can be recovered if an error or crash occur. As the Memtable is in volatile memory, it will not survive a power outage. Right after the data is written to log(disk) the data is put into the Memtable. Here it is available until the Memtable is to be written to disk. A typical reason for this is because the Memtable has met its max size limit. The process of writing the Memtable to disk is called Compaction. This is the process where the Memtable is converted to a SSTable. While the Memtable is written to disk, it is available as a “immutable” Memtable which cannot be changed, but is available for reads. When the Memtable is written, the memory which contains the “immutable” Memtable can be freed. The written log file that belongs to the Memtable can also be deleted.

When a *Get* operation is issued to LevelDB, the program will need to look into the Memtable and the SSTable.
First LevelDB will look into the Memtable, and if a “immutable” Memtable exists, this is also checked. Secondly, the SSTables are checked, first from the highest level (level 0) and then deeper (level 1) and deeper (level n). By doing lookups in this order, we can stop searching immediately when we find a matching key. This is because the first entry found will have the highest sequence number and therefor be the newest entry with this key.

4.

When a *Put* operation is issued to LMDB, the program will create a write-transaction. A write-transaction asks the LMDB to prepare the needed information so that the current B+tree can be copied into a new version. When a new version is ready, the changes to the tree is made. When the change/write process is complete, the transaction is committed, which writes all changed pages to disk and updates the database so all the new transactions will work on the new tree.

When a *Get* operation is issued to LMDB, the program will pretty much register a read-transaction and do a B+tree lookup. It is also important to note that the read must be pointed to the current version of the database.

5.

tHDD = tr + (2KB / 800KB) * tr = tr(1 + 2KB / 800KB)
tSSD = ta + tb = 0.1 + (2KB / 250MB/s) = 0.108 ms

tSSD = tHDD

0.108 ms = tr(1 + 2KB / 800KB)

0.108 ms = tr(1.0025)

tr = 0.108/1.0025 = 0.1077306733 ms

RPM = tr * 1000 * 60
RPM = 6463.8403992 ≈ 6464

6.
n = 200 000 000
l = 100 byte
V = n*l = 20GB = 2 * 10 ** 10
bSSD = 4KB
tSSD = 0.1 ms = 10** -4
M = 200 MB
N = 20 000 / 2*200 = 50
m = 2*10**8/100 = 2*10**6
t = 10**-6
hi = log2m = log2 2 *10**6 = 21
hm = log2N = log2 50 = 6

tCPUi = n(hi+1)t = 2 * 10 ** 8 * (21+1) * 10 ** -6 = 4400 seconds
tCPUm = n*hm*t = 2 * 10 ** 8 * 6 * 10 ** -6 = 1200 seconds

tIOSSD = (4 * V * tSSD)/ bSSD = (((4 * 2 * 10 ** 10) * 10 ** -4) / 4) / 1000 = 2000 seconds

a) Time to do the initial sort: 4400 seconds
b)
max(4400, 1000) + max(1200, 1000) = 5600 seconds

7.
a)
Run 1 (delfiler): 3, 5, 10, 32, 43, 44, 64, 79, 98
Run 2 (delfiler): 3, 5, 8, 35
Run 3 (delfiler): 2

b)

N = 370
p = 90

y = Ceil((370-1)/(90-1)) = 5

x = y(90-1) + 1 – 370

x = 76
You will need to add 76 files to the first merge round for an optimal tree.

c)

N = 370
p = 90
y = 5
x = 76

2(p-x) + 2p + 2p + 2p + 2n = 2(p-x) + 6p + 2n
2(14) + 6*90 + 2 * 370 = 1508 delfiler

8.

Table Data	R	A	B	WS
Key length (bytes)	8	8	8	
Record length (bytes)	200	600	300	
Number of records	250 000	300 000	1 200 000	
Tabel volume (megabytes)	50	180	360	10
Result records (bytes)		100	100	
Temp data file (bytes)		30	120	

Nested loop:
N = Ceil(V_a / M) = 30 / 10 = 3

$V^J_{NL} = V_A + nV_B + V_R$ = 180 + (3 * 360) + 50 = 1310 MB

Nested loop, with temp file:
$V = V_a + V_B$ + Write T_b + Read T_b + Read T_b +V_r
V = 180 + 360 + 120 + 120 + 120 + 50 = 950 MB

Partitioning:
$V = V_a$ + Write T_a + V_b + Write T_b + Read ($T_a + T_b$) + V_r
V = 180 + 30 + 360 + 120 + 150 + 50 = 890 MB

9.

People = 9 * 10 ^ 6
Names = 10 ^ 5
Rows = 150 bytes
Projection = 30 bytes
WS = 50 megabytes
M = WS

V_a = 150 * 9 * 10 ^ 6 = 1350 Megabyte
V_r = 10 ^ 5 * 30 = 3 Megabyte

Without guard:
$V_{nl}^{P} = ((V_a (V_a + M) / 2M) + V_r$
$V_{nl}^{P} = ((1350 (1350 + 50) / 2*50) + 3 = 18903$

With guard:
n = Ceil(V_{aK}/Ma) = Ceil((1350 * 30)/(50 * 150)) = 6

$V_{nl}^{P}{}_{w} = V_a (1 + (((n – 1)p)/a)) + V_r$

$V_{nl}^{P}{}_{w} = 1350(1 + (((6 – 1)30)/150)) + 3 = 2703$