Using Intel® Optane™ memory to expand the capacity of a typical realtime kdb market data system

by Eoin Cunning and Nick McDaid

Abstract

- Wrote a test suite to examine the latency of a typical real time kdb market data system bench marked against one of the busiest day for US equities in 2020
- Wrote tools to easily configure the splitting of real time data between dram and filesystem backed memory
- Compared the performance of storing table in DRAM vs Optane Memory via appDirect.
- Found that Optane is a viable solution to augment the realtime capacity of a kdb system and allows better utiliation of existing hardware where DRAM was previously a limiting factor

Table of Contents

Background

While there has been research published in the kdb community showing the effectiveness of Optane as an extremely fast disk, "performing up to almost 10 times faster than when run against high-performance NVMe", there as yet, has been no published research using Optane as a volatile memory source. Traditionally the bottle neck of market data platforms has been the amount of DRAM on a server, which in turn determines how many RDB's a server can host to serve queries for the most recent days data. This blog looks at wether Optane can be used to help solve this problem when mounted in DAX enabled App Direct Mode. It also provides a few useful utilities for moving your data into Optane with minimal effort, and documents the observed performance.

It is assumed the reader already has some knowledge of kdb, Optane persistent memory and how kdb has been adapted to interact with it. More information on this here. A good tech talk is also available from AquaQ.

Filesystem backed memory

Filesystem backed memory is available in kdb by use of the -m command line option and the .m namespace. To move a table or some columns into filesystem backed memory,such as Optane, you simply define them in the .m namespace without any other codes changes required. This can be accomplished by using the following two functions from the mutil.q script.

```
.mutil.colsToDotM:{[t;cs] // ex .mutil.colsToDotM[`trade;`price`size]
*     / enusre cs is list
    cs,:();
    / pull out columns to move an assign in .m namespace
    (` sv `.m,t) set ?[t;();0b;cs!cs];
    / replace columns in table
    t set get[t],'.m[t];
    }

.mutil.tblToDotM:.mUtil.colsToDotM[;()]
```

The only difference between running a DRAM and Optane rdb is starting it with -m option pointing to filesystem backed memory, and running the following command after loading the schema into a process:

```
.mutil.tblToDotM each tables[];
```

Using .mutil.colsToDotM rather than .mutil.tblToDotM allows users to move a subset of columns into Optane, giving a hybrid model which allow users to keep their most accessed data in DRAM and less frequently accessed columns in Optane.

Testing Framework

The Framework was designed to mimic the flow of a typical kdb market data capture system and then test the performance of storing the data in DRAM vs Optane. The capture section of the framework was largely base on the standard tick architecture tick setup

o ensure a sufficient stress test, the system simulates the volume and velocity of market data processed during the market crash on March 9th 2020. We prestressed the RDB with 65 million records, and then sent 48,000 updates per second for the next 30 minutes, split between trades and quotes in a 1:10 ratio. A ticker-plant consumed this feed and on a 50ms timer, disturbed the messages to an aggregation engine, which generated minute / daily aggregations and published these back to the ticker-plant. The RDB consumed all incoming messages. Every 2 seconds our monitor process would query the RDB and measure:

- Max trade / quote latency (time between the tick being generated in the feed process and the tick being accessible via an RDB query)
- Max latency on aggregated trade / quote message (as above - but also including the time for the aggregation engine to do it's calculations and it's additional messaging hops)
- Time for an asof join of all trades seen for a single symbol and their prevailing quote information
- Various memory stats such as Percentage DRAM used and Percentage Optane memory used

The above was considered a "stack". We ran four stacks concurrently for our testing. Two fully hosted in DRAM and two with their RDBs hosted in Optane.

A more detailed description of the architecture can be found in the appendix.

Findings

Test 1

Comparing 2 dram rdbs versus 2 appDirect rdbs were run at all at the same time. Figures are comparing single rdb service in dram versus single service.

	Monitor Timer	Tp Timer	DRAM	AppDirect	Comparison*
Max Latency	300	50	00:00.036520548	00:00.059900059	0.6097
Bytes in Memory	300	50	171893717664	5765504	29814
Average Query Time	300	50	00:00.047099680	00:00.067810036	0.6946
Queries per minute	300	50	200	200	1
Max Latency	50	50	00:00.125182124	04:45.203363517	7.320505e-06
Bytes in Memory	50	50	171893717648	5765488	29814

	Monitor Timer	Tp Timer	DRAM	AppDirect	Comparison*
Average Query Time	50	50	00:00.050620035	00:00.069780700	0.725416
Queries per minute	50	50	880	600	0.6818182
Max Latency	20	50	01:17.585672136	14:29.525561499	0.08861793
Bytes in Memory	20	50	171888597648	5765488	29813
Average Query Time	20	50	00:00.050031983	00:00.070396229	0.7107196
Queries per minute	20	50	1000	700	0.7
Max Latency	50	1000	00:01.054148932	00:01.037875019	0.984562
Bytes in Memory	50	1000	171893717392	5765488	29814.25
Average Query Time	50	1000	00:00.047581782	00:00.067816369	0.7016268
Queries per minute	50	1000	1000	1000	1

*Comparison is factor. Higher is better. Factor of 1 = same performance. Factor of 2 = 200% faster or half memory used.

Max latency is slightly larger in AppDirect rdbs. As we increase frequency with which we query rdb appDirect starts to struggle with latency more than dram. When using a larger tp frequency of 1 second. The appDirect rdb is able to main similar latency to rdb and service same number of queries.

Test 2

Comparison of 2 Dram rdbs running compared to 10 AppDirect Rdbs. Run with 1 second tp timer.

	2 DRAM Rdbs	10 AppDirect Rdbs	Comparison*
Max Latency	00:01.042379141	00:01.040085588	1.000038
% Memory of Dram used on server	63.2989	2.25653	28.05143
Average Query Time	00:00.046716868	00:00.067810036	0.6889374
Total Queries per minute	2000	10000	5

*Comparison is factor. Higher is better. Factor of 1 = same performance. Factor of 2 = 200% faster or half memory used.

Latency of two systems is comparable. % memory usage is much smaller for 10 appDirect rdbs than 2 DRAM rdbs. Average query time is slightly slower in appDirect but can runs more total queries because of extra services.

Discussion

Test 1 - Optane rdbs can run using significantly less memory with correct settings

We can see here there is a small trade off for latency and query performance but a massive saving in memory usage. When we increase the frequency with which we query the process we see that the fact that the query time takes longer means when tables are stored in appDirect means we can service less queries and can impact latency as rdb falls behind processing updates. However as we increase the frequency further to every 0.2 seconds we see that this is also true for the regular dram rdb as its latency starts to build.

This kind of impact depends on the system its being applied to. We see that when we increased the tp timer to every second and are doing bigger bulk inserts then we can handle the higher query rate in appDirect. The Max latency goes up to a second which is expected because of tp timer.

Test 2 - We can run many more appDirect rdbs on a single server than dram rdbs

We are able to run x5 more rdb services on the exact same hardware just with the addition of optane mounts (Mounts were 85% full at the end testing period). While only using less than 3% of the memory of the box. Increasing our ability to services 5x the volume of queries being run on the rdbs. We do however have to be wary of trying to access too much of the data in optane at the same time. Similar to how it standard set up we need to ensure we have enough memory to do calculations. If we need code to access more memory than is available this can also use appDirect memory. See this section in appendix for more details.

Queries take longer so make sure extra services are worth it

Further exploration into the query performance of the Optane rdbs should be looked at. could be 2-5x slower for raw pulling from dram compared to PMEM but seeing how this would actually affect a api and if this slow down is completely offset by ability to run more rdbs.

File system backed memory seems to be better suited to less volatile data storage

As well as testing storing our rdb data in Optane we also tried moving the table in the aggEngine into Optane. We found that the aggEngine then started to struggle to keep up and we saw latency grow. Most likely we believe this is due to the nature of the aggEngine code constantly reading from and writing to its cache of data these constant look up and re writing of data does not be the optimal use case for the technology and doesn't even give us a big memory saving benefit.

Read versus write

The general consensus was that PMEM reads are much closer to DRAM performance that writes however we ran into more issues with the reads when querying the rdb than writing the data. We believe this is

because out typical market data system will be writing small inserts to the optane mounts on each update where as queries can attempt to read the entire data that is stored there. .e.g `exec max time from quote`. This is no different to how kdb behaves with DRAM it is just more noticeable as queries are slightly slower.

Middle ground

Steps for further investigation or post could involve

- enhancing query testing, have load balancer in-front of rdbs and compare replacing 1 dram rdb with 3 Optane rdbs. Ensure latency not an issue due to write contention and confirm that query impact not affect slow down mitigated by extra services available to run them.
- using Optane for large static tables that take up a lot of room in memory but are queried so often take to long to have written to disk. .e.d flat-file table you load into hdbs or reference data tables saved in gateways
- replay performance - we have seen write contention so be interesting to explore how replay would work for PMEM rdbs. This would probably stress test the write performance to the PMEM mounts. May need to batch updates and and write to dram as intermediary

Conclusion

- It is possible to run rdbs with data stored in Optane instead of dram. Code required for such changes is outlined.
- There is a trade off in performance for querying from file system backed memory. Can be seen to be 2-5x slower for raw data pull. But as you move to more aggregated cpu bound type queries this can become almost negligible.
- Possible that just trying to convert a working rdb to Optane. This slow down in read speed will cause queries to take longer and could result in rdb delaying processing of of new messages for tp.
- If a rdb is already seeing very high query volumes may not be advisable to move everything to Optane memory.
- More realistic use case is if you have columns that are seldom queried that subset of columns could be moved to .m namespace freeing up ram for more rdbs.
- Not only is same amount of memory cheaper in optane compared to DRAM but the max size optane chips is significantly larger than for DRAM. Meaning the memory limit of single server is significantly increased.
- While it isn't primarily marketed as a DRAM replacement technology, we found it was a very helpful addition in augmenting the volatile memory capacity of a server hosting realtime data.
- Reduce cost of infrastructure running with less servers and DRAM to support data processing and analytic workloads

About the authors

Eoin Cunning is a kdb+ consultant, based in London, who has worked on several KX solutions projects and currently works on a market data system for a leading global market maker firm.

Nick McDaid is a kdb+ consultant, based in London who has worked as a developer in multiple top tier banks and currently works as a developer in a European hedge fund.

Appendix

KDB 4.0 Release notes regarding Filesystem backed memory

```
2019.10.22
NUC
memory can be backed by a filesystem, allowing use of dax-enabled
filesystems (e.g. AppDirect) as a non-persistent memory extension for
kdb+.
cmd line option -m path to use the filesystem path specified as a separate
memory domain. This splits every thread's heap in to two:
  domain 0: regular anonymous memory (active and used for all allocs by
default)
  domain 1: filesystem-backed memory
 .m namespace is reserved for objects in domain 1, however names from
other namespaces can reference them too, e.g. a:.m.a:1 2 3
 \d .m changes current domain to 1, causing it to be used by all further
allocs. \d .anyotherns sets it back to 0
 .m.x:x ensures the entirety of .m.x is in the domain 1, performing a deep
copy of x as needed (objects of types 100-103h,112h are not copied and
remain in domain 0)
 lambdas defined in .m set current domain to 1 during execution. This will
nest since other lambdas don't change domains:
c  q)\d .myns
  q)g:{til x}
  q)\d .m
  q)w:{system"w"};f:{.myns.g x}
  q)\d .
  q)x:.m.f 1000000;.m.w` / x allocated in domain 1
 -120!x returns x's domain (currently 0 or 1), e.g 0 1~-120!'(1 2 3;.m.x:1
2 3)
 \w returns memory info for the current domain only:
  q)value each ("\\d .m";"\\w";"\\d .";"\\w")
-w limit (M1/m2) is no longer thread-local, but domain-local; cmdline -w,
\w set limit for domain 0
mapped is a single global counter, same in every thread's \w
```

Hardware

Server details

```
[root@clx4 ~]# lscpu
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              56
On-line CPU(s) list: 0-55
Thread(s) per core:  1
Core(s) per socket:  28
Socket(s):           2
```

```
NUMA node(s):        2
Vendor ID:           GenuineIntel
CPU family:          6
Model:               85
Model name:          Genuine Intel(R) CPU 0000%@
Stepping:            6
CPU MHz:             2538.054
CPU max MHz:         4000.0000
CPU min MHz:         1000.0000
BogoMIPS:            4400.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            1024K
L3 cache:            39424K
NUMA node0 CPU(s):   0-27
NUMA node1 CPU(s):   28-55
Flags:               fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall
nx pdpe1gb rdtscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl
xtopology nonstop_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds_cpl
vmx smx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid dca sse4_1 sse4_2
x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm
abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 invpcid_single ssbd mba
ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid
fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm cqm mpx rdt_a
avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd
avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_llc cqm_occup_llc
cqm_mbm_total cqm_mbm_local dtherm ida arat pln pts hwp hwp_act_window
hwp_epp hwp_pkg_req pku ospke avx512_vnni md_clear flush_l1d
arch_capabilities
[root@clx4 ~]# free -h
              total        used        free      shared  buff/cache
available
Mem:          374Gi       5.0Gi       352Gi        29Mi        17Gi
367Gi
Swap:         4.0Gi       0.0Ki       4.0Gi
[root@clx4 ~]#
[root@clx4 ~]# df -h
Filesystem           Size  Used Avail Use% Mounted on
devtmpfs             188G     0  188G   0% /dev
tmpfs                188G     0  188G   0% /dev/shm
tmpfs                188G   27M  188G   1% /run
tmpfs                188G     0  188G   0% /sys/fs/cgroup
/dev/mapper/cl-root   50G  8.0G   43G  16% /
/dev/mapper/cl-home  690G   84G  607G  13% /home
/dev/sda1            976M  257M  653M  29% /boot
tmpfs                 38G   20K   38G   1% /run/user/42
tmpfs                 38G     0   38G   0% /run/user/1000
/dev/pmem0           732G   73M  695G   1% /mnt/pmem0
/dev/pmem1           732G   73M  695G   1% /mnt/pmem1
[root@clx4 ~]#
[root@clx4 ~]# ipmctl show -topology
 DimmID | MemoryType                  | Capacity    | PhysicalID|
```

```
DeviceLocator
==========================================================================
======
 0x0001 | Logical Non-Volatile Device | 126.375 GiB | 0x0026    |
CPU1_DIMM_A2
 0x0011 | Logical Non-Volatile Device | 126.375 GiB | 0x0028    |
CPU1_DIMM_B2
 0x0021 | Logical Non-Volatile Device | 126.375 GiB | 0x002a    |
CPU1_DIMM_C2
 0x0101 | Logical Non-Volatile Device | 126.375 GiB | 0x002c    |
CPU1_DIMM_D2
 0x0111 | Logical Non-Volatile Device | 126.375 GiB | 0x002e    |
CPU1_DIMM_E2
 0x0121 | Logical Non-Volatile Device | 126.375 GiB | 0x0030    |
CPU1_DIMM_F2
 0x1001 | Logical Non-Volatile Device | 126.375 GiB | 0x0032    |
CPU2_DIMM_A2
 0x1011 | Logical Non-Volatile Device | 126.375 GiB | 0x0034    |
CPU2_DIMM_B2
 0x1021 | Logical Non-Volatile Device | 126.375 GiB | 0x0036    |
CPU2_DIMM_C2
 0x1101 | Logical Non-Volatile Device | 126.375 GiB | 0x0038    |
CPU2_DIMM_D2
 0x1111 | Logical Non-Volatile Device | 126.375 GiB | 0x003a    |
CPU2_DIMM_E2
 0x1121 | Logical Non-Volatile Device | 126.375 GiB | 0x003c    |
CPU2_DIMM_F2
 N/A    | DDR4                        | 32.000 GiB  | 0x0025    |
CPU1_DIMM_A1
 N/A    | DDR4                        | 32.000 GiB  | 0x0027    |
CPU1_DIMM_B1
 N/A    | DDR4                        | 32.000 GiB  | 0x0029    |
CPU1_DIMM_C1
 N/A    | DDR4                        | 32.000 GiB  | 0x002b    |
CPU1_DIMM_D1
 N/A    | DDR4                        | 32.000 GiB  | 0x002d    |
CPU1_DIMM_E1
 N/A    | DDR4                        | 32.000 GiB  | 0x002f    |
CPU1_DIMM_F1
 N/A    | DDR4                        | 32.000 GiB  | 0x0031    |
CPU2_DIMM_A1
 N/A    | DDR4                        | 32.000 GiB  | 0x0033    |
CPU2_DIMM_B1
 N/A    | DDR4                        | 32.000 GiB  | 0x0035    |
CPU2_DIMM_C1
 N/A    | DDR4                        | 32.000 GiB  | 0x0037    |
CPU2_DIMM_D1
 N/A    | DDR4                        | 32.000 GiB  | 0x0039    |
CPU2_DIMM_E1
 N/A    | DDR4                        | 32.000 GiB  | 0x003b    |
CPU2_DIMM_F1
[root@clx4 ~]#
[root@clx4 ~]# ipmctl show -memoryresources
 MemoryType   | DDR         | PMemModule   | Total
```

```
============================================================
 Volatile     | 381.500 GiB | 0.000 GiB    | 381.500 GiB
 AppDirect    | -           | 1512.000 GiB | 1512.000 GiB
 Cache        | 0.000 GiB   | -            | 0.000 GiB
 Inaccessible | 2.500 GiB   | 5.066 GiB    | 7.566 GiB
 Physical     | 384.000 GiB | 1517.066 GiB | 1901.066 GiB
```

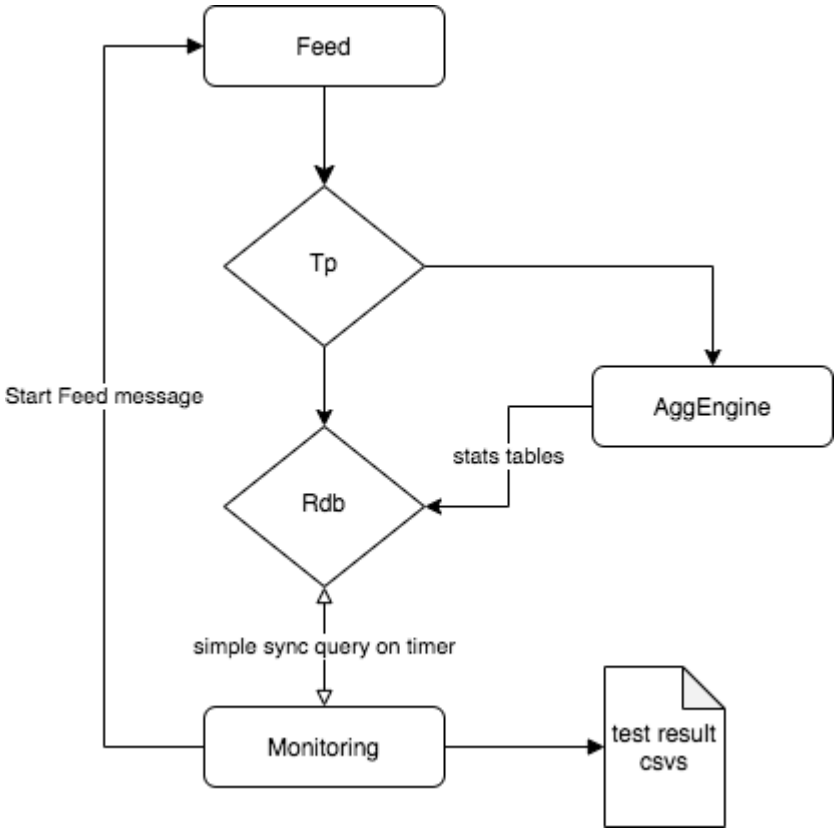
Server set up for appDirect

```
# set all Optane to appDirect either via command below of BIOs settings
(required reboot)
ipmctl create -goal persistentmemorytype=appdirect

# creating namespace from persistent namespace (align good for pages)
ndctl create-namespace --mode=fsdax --region=0 --align=2M
ndctl create-namespace --mode=fsdax --region=1 --align=2M

# make a file system
mkfs -t xfs /dev/pmem0
mkfs -t xfs /dev/pmem1

# mount file system make writeable
mkdir /mnt/pmem0
mkdir /mnt/pmem1

# mount in direct access mode
mount -o dax /dev/pmem0 /mnt/pmem0/
mount -o dax /dev/pmem1 /mnt/pmem1/

# permissions mounts
chmod 777 /mnt/pmem0
chmod 777 /mnt/pmem1
```

Numa settings

The standard recommendation when using numa is to set --interleave=all `numactl --interleave=all q` but found slightly better performance in aligning the numa nodes with the persistent memory namespaces `numactl -N 0 -m 0 q -q -m /mnt/pmem0/` and `numactl -N 1 -m 1 q -q -m /mnt/pmem1/`

Testing Framework Architecture



Feed

Data arrives from a feed. Normally this would be a feed-handler publishing data from exchanges or vendors. For consistent testing we have simulated this feed from another q process. This process generates random data and publishes down stream. For the rate of data to send we looked and the largest day of market activity in 2020. which during its last half hour of trading before the close consisted of 80,000,000 quote msgs and 15,000,000 trades. Code can be viewed here

Tp

Standard kdb tp running in batch mode. Code can be viewed here

AggEngine

This process is the main departure for standard tick set up. This process subscribes to standard trade and quote tables and calculates running daily and minute level stacks for all symbols. These aggregated tables are then published to the rdb from which they could then be queried. This process was added in order to have some kind of more complex event process as well as standard rdb. This process will constantly have to read and write to memory. where generally only has to write as it appends data and only read for queries) Code can be viewed here

Rdb

Standard rdb subscribes to tables from the tp. We also added option to prestress the memory before our half hour of testing again looking at the market data on 2020.03.02 there were 650,000,000 quote msgs and 85,000,000 trades at 15:30 so we insert these volumes into the rdb at start up. This aims to ensure that the ram is already somewhat saturated. Full code availablehere

Monitor

The Monitor process connects to the rdb and collect performance stats on a timer. Main measurements are for latency of quote table this will track if messages getting queued from the tp, quote stats table if this falls behind indicates issue in aggEngine and the query time which measures how long it takes to run some typical rdb queries .e.g aj On start up this process also kicks off the feed once having successfully connected to rdb to start testing run. Once endTime has been reached the stats collected are aggregated and written to csv file. Again link for full code available here

Defining functions to use file system backed memory

We also need to be aware that it is not just data stored in tables that uses memory. Queries and functions that we run also need to assign memory and which domain they use will depend on how the function was defined.

Here for example we compare the same functionality defined in both the root and .m namespace.

```
q)n:1000000
q)tab:([]t:n?.z.n;s:n?`4;n?100f;n?100)
q)f:{`s`t xasc x}
q)\d .m
q.m)f:{`s`t xasc x}
q.m)\d .
q)\ts f tab
133 41944096
q)\ts .m.f tab
351 512
q)f[tab]~.m.f[tab]
1b
```

A full copy of the data is required for the sort listed above. When running the function defined in the .m namespace, that copy is assigned to the filesystem backed memory. While this results in a drop off in performance, it does provide a huge saving of DRAM memory. This could be very useful in instances where users need to run multiple concurrent back-loaders or are struggling to perform end of day sorts with all the data in memory.

We did explore this functionality and some utility functions are available in mutil.q script to redefine functions and namespaces to allocate memory to file system back memory instead of DRAM but exploring uses of this further was deemed to be out of scope for this post.

We do however have to be aware of it as if we aren't defining code in the .m namespace then we will be constrained to the amount of DRAM we have. If we have more memory stored in appDirect that there is DRAM available and have queries that copies a lot of the data we could run out of DRAM.

So if leaving an existing api/query code as is there will be an initial over head for pulling the data from appDirect instead of dram. After that once we cause kdb to assign data to new memory space it will get assigned in dram and everything from that point on should be as fast as it was in a dram rdb.