Data Engine for NoSQL - IBM Power Systems™ Edition

White Paper and Technical Reference

Brad Brech – Juan Rubio – Michael Hollinger
IBM Systems Group

June 2015

Contents

1 Executive Summary

The use of NoSQL has exploded in recent years as new customer facing applications require unprecedented response time and scale to meet mobile user expectations. Typical NoSQL implementations run entirely in-memory, or rely heavily on memory as a cache, and therefore become expensive and hard to scale and the latency to traditional I/O attached storage does not meet the application requirements. A new solution is needed to support the growth in NoSQL based applications.

IBM Data Engine for NoSQL - Power Systems Edition creates a new tier of memory by attaching up to 57 Terabytes of auxiliary flash memory to the processor without the latency issues of traditional I/O storage. While not as fast as DRAM, the latency is within the acceptable limit of most applications especially when data is accessed over the network. Flash is also dramatically less expensive than DRAM, and helps reduce the deployment and operational cost for delivering the customer solution. Customers, MSPs, and ISPs all benefit from application of this new technology in the NoSQL application space. Exploiting hardware and software built-in to IBM's flagship POWER8 open architecture means that clients no longer much choose between "big" or "fast" for their solutions.

2 Business Problem

Because the SQL database tier has always been a critical component of application response time, the industry has expended significant effort optimizing it. However, the massive scale and growth of mobile applications built around cloud solution architectures have driven the adoption of NoSQL databases for their scale, resiliency, and simplicity.

The total cost of ownership for NoSQL databases has been very high due to relatively high cost of system memory[1],and the number of scale out nodes needed to hold the DRAM. This high deployment cost has traditionally limited the adoption of NoSQL implementations like Redis to applications that have relatively small datasets, or to only those parts of the application that absolutely need super-fast performance.

IBM and Redis Labs have joined forces to address this issue, recognizing a unique opportunity to leverage flash attached to IBM's open POWER8 processor via the CAPI (Coherent Accelerator Processor Interface) interface. NoSQL solutions utilizing flash as solid-state-drives (SSDs) on the I/O bus do perform better than spinning disk, however they still are not able to meet the latency times required because of I/O overheads compared to RAM. Flash DIMMs likewise have size, resiliency, and other performance limitations. By using a combination of DRAM and CAPI-attached Flash, IBM and Redis Labs are able to provide a solution that delivers a significant reduction in both deployment and operational costs while providing the ability to power faster, larger, and more scalable applications.

For a 12 TB database, the deployment (TCA) cost is 1/3 the traditional deployment, and by reducing the nodes required for the solution by up to 24X, there is a dramatic reduction in the operations costs (TCO) for networking, space-power-cooling and operations overhead.

3 IBM Solution

The IBM Data Engine for NoSQL builds on the new Coherent Accelerator Processor Interface (CAPI) on POWER8 systems, which provides a high-performance solution for the attachment of devices to the

[1] Typical DRAM cost is 250X the cost of disk and 10X the cost of flash, on a per-gigabyte basis.

processor. In this section of the paper, we will describe CAPI, FLASH, REDIS SW and how together we created a unique and valuable solution that addresses the scaling issues of typical NoSQL deployments.

3.1 Coherent Accelerator Processor Interface (CAPI) Overview

A key innovation in POWER8's open architecture is the Coherent Accelerator Processor Interface (CAPI). CAPI provides a high bandwidth, low latency path between partner devices, the POWER8 core, and the system's open memory architecture. CAPI adapters reside in regular PCIe x16 slots, and use PCIe Gen 3 as an underlying transport mechanism. However, similarities with other IO cards and accelerators end here. CAPI-capable devices can replace either application programs running on a core or provide custom acceleration implementations. CAPI removes the overhead and complexity of the I/O subsystem, allowing an accelerator to operate as part of an application. This results in a code path reduction, since applications are able to interact with the accelerator directly without going through the kernel. IBM's solution enables higher system performance with a much smaller programming investment, allowing hybrid computing to be successful and accessible across a much broader range of applications.

In the CAPI paradigm, the specific algorithm for acceleration is contained in a unit on the FPGA called the accelerator function unit (AFU or accelerator). The purpose of an AFU is to provide applications with a higher computational unit density for customized functions to improve the performance of the application and offload the host processor. Using an AFU for application acceleration allows for cost-effective processing over a wide range of applications. A key innovation in CAPI is that the POWER8 system contains custom silicon that provides the infrastructure to treat the client's AFU as a coherent peer to the POWER8 processors.

Each CAPI accelerator processor is a "first-class" citizen within the system, working with the same memory and address space used by the POWER8 processors in the server. IBM provides a durable service and abstraction layer to each accelerator that simplifies management of the accelerator device, allowing solution designers to focus more on addressing application-specific challenges. This means the accelerator, for example, can participate in locks just like any other POWER8 thread, and greatly-lowers the overhead of communication to the device. Additionally, simplified addressing makes the accelerator easy to use, and easy to program. Applications that are well-suited for CAPI include Monte Carlo algorithms, key-value stores, and financial and medical algorithms.

CAPI can also be used as a foundation for flash memory expansion, as is the case for the IBM Solution for Flash Optimized NoSQL. This innovative product gives the system access to 56 TB of data through the CAPI-connected solution.

The overall value proposition of CAPI is that it significantly reduces development time for new algorithm implementations and improves performance of applications by connecting the processor to hardware accelerators and allowing them to communicate in the same language (eliminating intermediaries such as I/O drivers). For additional detail on CAPI, see the CAPI white paper available at: http://www-304.ibm.com/support/customercare/sas/f/capi/home.html .

3.2 Redis Overview

Redis is an open source, BSD licensed, advanced **key-value cache** and **store.** Redis is not a *plain* key-value store, it is a *data structures server*, supporting different kinds of values. In traditional key-value stores users associate string keys to string values. In Redis the value is not limited to a simple string, but can also hold more complex data structures. The following is the list of all the data structures currently supported by Redis:

Data Structure	Description
Binary-Safe Strings	
Lists	Collections of string elements sorted according to the order of insertion
Sets	Collections of unique, unsorted string elements
Sorted sets	Similar to Sets but where every string element is associated to a floating number value, called *score*
Hashes	Maps composed of fields associated with values, where the field and the value are strings (these are very similar to Ruby or Python hashes)
Bit arrays (or simply bitmaps)	String values represented as arrays of bits manipulated with special commands; users can set and clear individual bits, count all the bits set to 1, find the first set or unset bit, etc.
HyperLogLogs	Probabilistic data structure which is used in order to estimate the cardinality of a set

Table 1: Data structures supported by Redis

3.3 Flash Optimized NoSQL Hardware

The IBM Data Engine for NoSQL leverages the ability to provide a high throughput, low latency connection to a Flash memory array to create a unique memory tier that addresses the scaling issues of NoSQL deployments. The design allows for the processor main memory to provide fast response times that applications require by utilizing main memory to cache or hold "hot" data. The solution, however provide application access to up to 56 Terabytes of flash memory, using the IBM Flash Systems storage solution. The flash array is attached using a CAPI adapter card to provide a high bandwidth, low latency path between the processor and the flash. The adapter accomplishes this by using a Field-Programmable Gate Array (FPGA) chip and Fiber Channel IO ports. The FPGA device contains purpose-built logic for management and access control of the attached Flash Systems storage array. This logic may be updated as needed, as it resides on the FPGA. Providing the POWER8 processors direct access to both DRAM and FLASH allows application software to adjust memory and flash usage ratios to optimize performance and cost based on the specific service level agreements.

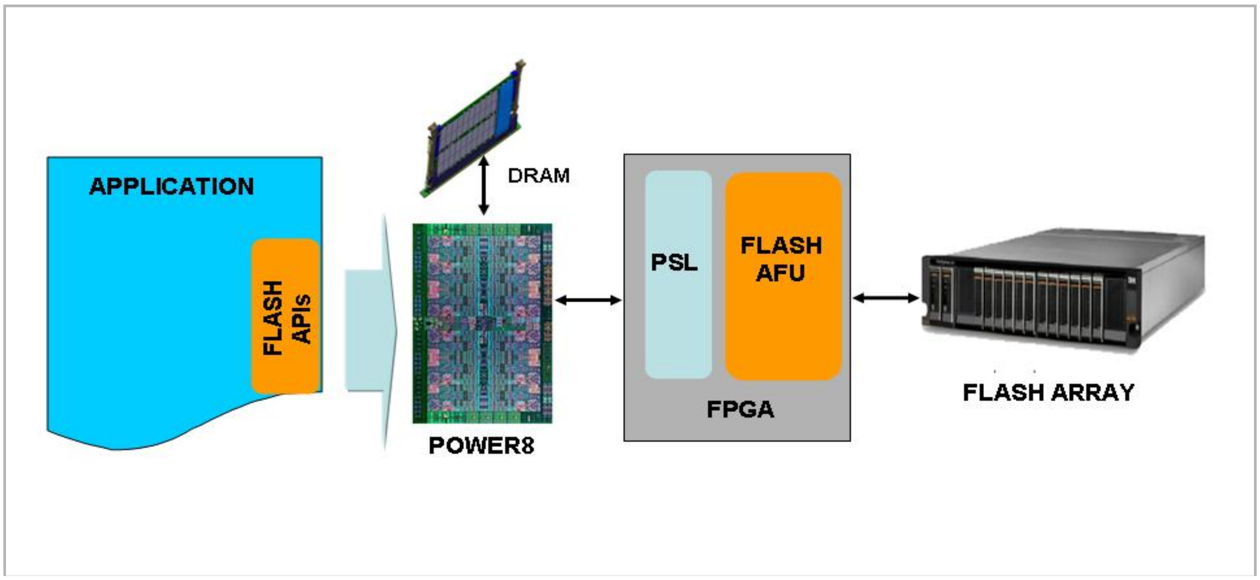

Figure 1: IBM Data Engine for NoSQL Conceptual Diagram

3.3.1 IBM Data Engine for NoSQL Software

IBM Data Engine for NoSQL solution provides the application direct access to the flash through a set of developer APIs that provide a key-value and raw block i/o interfaces to manage and access the data in flash. The total SW package is made up of four components:

Master Context (MC) - Daemon which initializes the adapter, completes LUN discovery and mapping, does error recovery and health checking, addresses uncorrectable errors, and manages link events on behalf of client application software.

Block I/O APIs – Handles read/write requests for specific blocks and issue commands directly to the AFU to read/write data on a logical address in flash. In addition, it handles responses for those requests.

Key Value Storage (KV) APIs - Provides a generic key-value database that forms the bridge between Redis and the Block I/O APIs above.

The diagram below shows the SW components of the solution.

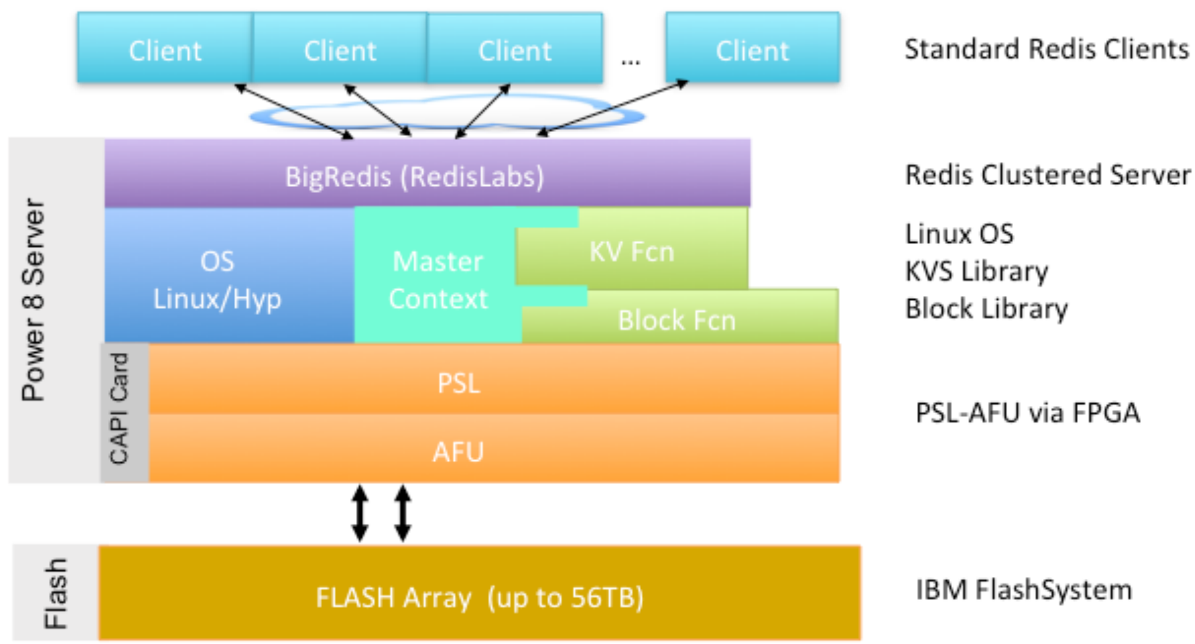

Figure 2: Software Component Overview

3.4 BigRedis Overview

BigRedis is the name Redis Labs gives to this solution. BigRedis is a version of Redis Labs Enterprise Cluster that has been extended to take advantage of the large array of Flash provided by the CAPI interface. It not only leverages the large flash, but the ability of the Power S822L system to deliver up to 192 threads of execution, effectively making this solution a "cluster in a box" implementation. This allows the Redis server to scale using both the large flash array and the large execution thread count available within a single node. BigRedis allows the user to select not only the size of the store required, but also select the price/performance point, or service level agreement, that the customer determines fits his or her individual needs. Based on projected solution costs, the following chart shows the typical relative performance and cost as the user changes the ratio of memory to flash.

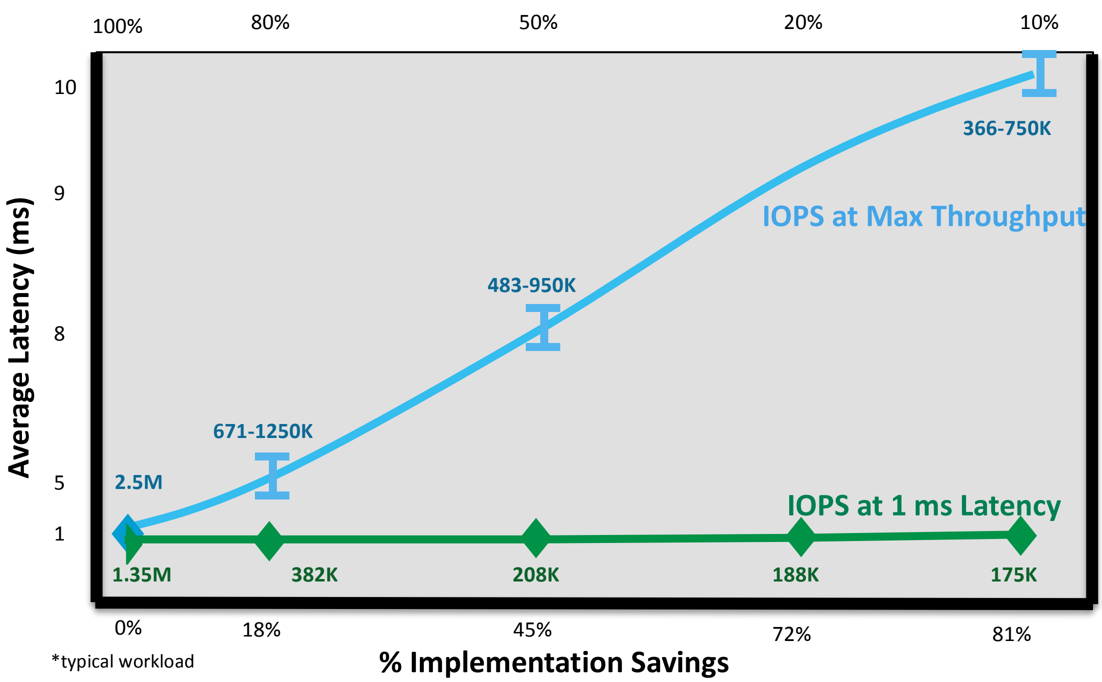

Figure 3: Users may pick the performance / cost point that meets their solution needs, be it IOPs Rate or Latency requirements. As the relative amount of Flash increases in a redis instance, the relative cost of flash vs RAM leads to implementation savings. The use of CAPI-accelerated flash allows the user to "dial in" the desired latency and cost for a particular application. This flexibility is not possible in all-RAM redis.

For an overview video about Big Redis with CAPI Flash, please refer to https://www.youtube.com/watch?v=Wh8cqzFpxCE .

4 Target Markets and Segmentation

Potential consumers of solutions built on the IBM Data Engine for NoSQL come from a diverse set of industries and research areas currently using or investing in NoSQL solutions. Currently BigRedis from Redis Labs is the only NoSQL MSP solution available, but other MSP solutions will be announced in the future. The market for optimized NoSQL databases is vast. Figure 4 below illustrates some of the industries and target markets that use NoSQL based solutions. These potential markets are just a subset of the overall market space where CAPI Flash Optimized NoSQL might be applicable.

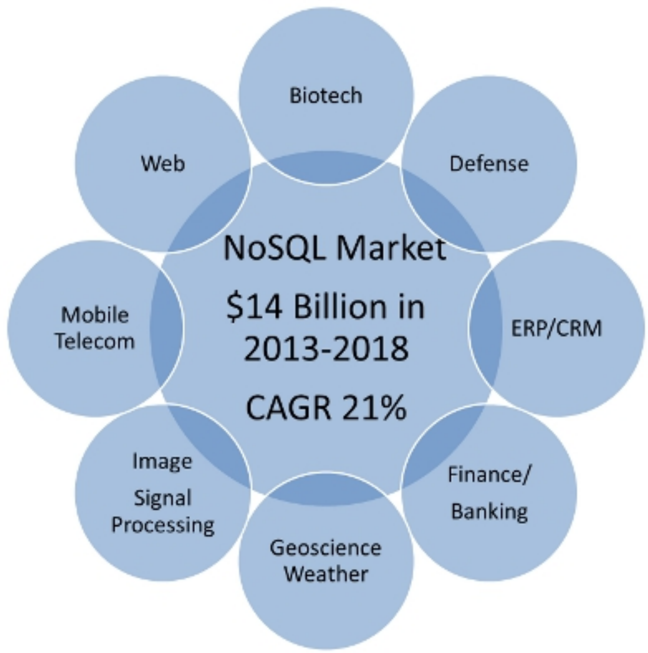

Figure 4: Potential Markets for Solutions leveraging POWER8 and CAPI Acceleration

5 Strategy for Growth and Adoption

The IBM Data Engine for NoSQL is initially targeted at solving the scaling problems for NoSQL deployments, but IBM is evaluating other application types that may leverage this new memory tier and the attributes it brings. With the wide array of NoSQL solutions available in the market, we see the initial Redis offering as a starting point in improving the cost / benefit attributes of a wide range of NoSQL offerings. It also provides a unique solution to other applications that need access to large storage at latencies significantly better than standard I/O attachment methods. In-memory databases and other big data analytics solutions are other possible target applications for this solution

5.1 Leveraging the OpenPOWER Foundation

POWER8's CAPI Interface is a key part of the OpenPOWER architecture, enabling partners to innovate across the entire server solution architecture. Multiple OpenPOWER Foundation members are developing on new CAPI-Flash solutions today. These early adopters will provide solutions in many of the target market spaces discussed previously. Look for additional NoSQL solutions from OpenPOWER Foundation members in the future.

5.2 Early Product Offering and Future Directions

IBM Data Engine for NoSQL is in the developmental and initial production phases of development. IBM's initial product offering is being used for Big Redis a product of Redis Labs. We continue to work with other NoSQL providers and will make the new APIs available in the future for wider development usage. For additional information on Redis offerings, go to: https://**redislabs**.com .

6 Conclusion

IBM Data Engine for NoSQL - Power Systems Edition is an innovation that enables middleware and application developers access to a new tier of storage based on flash technology attached via the POWER8 processor's open CAPI bus, effectively bringing flash "closer" to the processor. This hardware and software solution provides a cost and performance-sensitive alternative to scale-out DRAM-based solutions.

Appendix A Advanced Configuration

Initial setup of the IBM DataEngine for NoSQL software package is automatic. Upon installation, the software package will prompt to begin setup. Setup may be re-run at any time by invoking /opt/ibm/capikv/bin/setup.sh as a superuser.

For advanced configuration, tools such as puppet may be used to create or provision configuration files and keys. Please refer to section A.1.1 below for details of these resources.

A.1 FlashSystem Storage Configuration

Each FlashSystem and POWER8 system must be connected via point-to-point Fiber Channel for high speed IO, and Ethernet for management. Replacement of an EJ16 adapter, a FlashSystem, or an entire POWER8 system will require reconfiguration of the system.

1. To reconfigure the system, first ensure that the “ibmcapikv” package has been installed with “dpkg -l ibmcapikv” from the Linux console.
2. Run “sudo /opt/ibm/capikv/bin/setup.sh” to restart the full setup process. This will allow reconfiguration of ssh keys needed for platform management, creation of a “host” within the FlashSystem for CAPI Acceleration, and (if needed) creation of a small test VDisk which may later be replaced.
3. At the prompt, answer “Y” to start the setup process.
4. Enter the IP address of the FlashSystem when prompted.
5. Enter the username of the FlashSystem administrator that will be used to set up the system. Press “enter” to accept “superuser” as the default value.
6. Enter the host name of the POWER8 system. This will be used to create a “host” in the FlashSystem. Press “enter” to accept the system’s hostname as the default, if desired.
7. Wait while the setup script attempts to establish a management connection to the FlashSystem. If a previous connection has been established to this FlashSystem, the setup script will use existing ssh keys and identities to confirm that the POWER8 and FlashSystem are correctly-configured.
    a. If a connection is established, the script will validate that a Host exists for the POWER8’s Flash Accelerator cards, and that a Vdisk is mapped.
8. If no connection is established, the setup script will create an SSH key pair and save it in /opt/ibm/capikv/data, and upload the key to the FlashSystem.
    a. Each POWER8 must have an SSH public / private key pair and an associated username for FlashSystem management. These are by default the “hostname” of the POWER8. To use the same key pair for multiple POWER8s or use the same management user ID, manually-edit the /opt/ibm/capikv.ini file to change the “username” to the desired value, and change the “keypath” to the location of the private key that will be used. To upload this key to the FlashSystem manually, refer to the FlashSystem help documentation for creating a new user.

   b. If a user already exists, the setup script will fail. To continue, upload the SSH keys generated manually, or delete the user from the flashsystem using the "rmuser" command (or using the GUI).

9. If no Host is defined for the POWER8's Flash Accelerators, the setup script will create a host for each WWPN (2 per adapter) and create a single "host." Note that all accelerators must be mapped to the same FlashSystem.

10. If no Vdisk is mapped to the Host, the setup script will create a small vdisk, and map it to the Host.

   a. To manually-configure this, run "sudo stop mserv" and use the FlashSystem GUI (or CLI) to create a larger Vdisk (at least 16 GB in size) and map it to the Host. Run "sudo start mserv" to restart the mserv daemon which arbitrates access to CAPI-accelerated flash.

A.1.1 Configuration Data Locations

Artifact	Location	Description
Version Marker	/opt/ibm/capikv/version.txt	Build information about the IBM Data Engine for NoSQL User Libraries
Default FlashSystem SSH keys	/opt/ibm/capikv/data/capikv_rsa(.pub)	Public / private keypair used for FlashSystem communication by default. This may be overridden by modifying the capikv.ini "keypath" configuration entry.
capikv.ini	/opt/ibm/capikv/data/capikv.ini	Base configuration information for FlashSystem and POWER8 connectivity.

A.2 Logging and Debug Data

A.2.1 Important Log Locations

Artifact	Location	Description
mserv Daemon	upstart Systems Startup: /var/log/upstart/mserv.log Runtime events: /var/log/syslog (grep for "mserv") systemd Systems Startup: journalctl mserv Runtime events: /var/log/syslog (grep for "mserv")	The "mserv" daemon controls and allocates sections of the CAPI-accelerated Flash device. It is a critical service, and must run in order for KV or Block services to complete successfully.
Block API Errors & Traces	/var/log/syslog (grep for "CXLBLK") To increase verbosity of Block API traces, see section A.2.2.	Block APIs provide "raw" access to underlying CAPI-accelerated Flash.

KV API Errors	Refer to return codes and errno.h for each API call	Key-Value APIs (arkdb.so) provide high-level object storage / retrieval
cxl.ko Kernel Module	dmesg /var/log/kern.log	cxl.ko provides underlying services for core CAPI Accelerator services to the operating system.

A.2.2 Controlling Block API Trace

Users of the cflsh block API shared libraries may find debug and error data in syslog (see the table above). By default, the block APIs log only error events. To control this and gain additional debug (useful during development of a new application), set the "`CFLSH_BLK_TRC_VERBOSITY`" environment variable. Valid values are 0 (all tracing disabled) to 9 (extremely-verbose tracing of all events), inclusive. To turn up tracing to the maximum value:

1. `sudo CFLSH_BLK_TRC_VERBOSITY=9 /opt/ibm/capikv/bin/_tst_ark -n 10 -dev /dev/cxl/afu0.0s #example with maximum verbosity`
2. `cat /var/log/syslog | grep CXL #read block API traces`

```
Mar 29 19:12:09 hostname CXLBLK[28580]: cflash_block.c,cblk_open,1173,opening
/dev/cxl/afu0.0s with max_num_requests = 256 and flags = 0x2
Mar 29 19:12:09 hostname CXLBLK[28580]:
cflash_block_linux.c,cblk_chunk_attach_process_map,234,contxt_handle = 0x23
Mar 29 19:12:09 hostname CXLBLK[28580]:
cflash_block_linux.c,cblk_chunk_attach_process_map,277,mmio = 0x3fff891a0000
```

A.2.3 Collecting Hardware Trace Data

Collection of Accelerator trace and error data is bundled into a single script that produces a compressed tarball of relevant data from all present accelerators in the system. Hardware trace buffers are finite and may be overwritten over time, so please collect trace data promptly if an error event occurs.

1. As a user with sudo authority, cd to the directory where you would like trace and error data to be captured.
2. Run "sudo /opt/ibm/capikv/afu/cxlffdc" and wait for each accelerator to be queried.
3. Look for a file "cxlffdc.tgz" in the current working directory.
4. Provide this cxlffdc.tgz file to IBM for further assistance.

Appendix B Example Workloads

B.1 Block API Exerciser - asyncstress

asyncstress generates a mix of read/write asynchronous I/Os via the cblk IO API. It is intended as a light-weight asynchronous IO stress test.

B.1.1 Usage

```
/opt/ibm/capikv/tests/asyncstress -h
usage: asyncstress <args>
-d <name>  'name of device, usually /dev/cxl/af0.0s or /dev/cxl/afu1.0s'
-b <int> '# of blocks to run the test over, i.e. lba=0..b-1
-n <int> '# of ops to run (combination of reads and writes total'
-a <int> '# of outstanding ops to maintain'
-r <int> '# for read  ratio, read  ratio is r/(r+w)'
-w <int> '# for write ratio, write ratio is w/(r+w)'
-t <int> '# threads (not implemented yet)'
-v <bool> '0/1 to use virtual or physical lba (not implemented'
-o <bool> '0/1 to retire ops inorder/any order'
-c <bool> '0/1 record history (not implemented'
-p <name> 'arbitrary name echoed in the stats line to help identify run stats
```

B.1.2 Example Invocation - 4GB Random Access, 10M 4KB R/W, 75% R, 25% W

```
asyncstress -b 1000000 -n 10000000 -a 128 -r 75 -w 25 -d /dev/cxl/afu0.0s
```

The program starts by issuing 128 ops, then whenever an op completes another op is immediately issued thus trying to maintain 128 ops in flight. Once the 10 millionth op is issued, the program waits for the final 128 ops to complete. Note that no -p parameter defaults to Unnamed, the line above beginning with Unnamed is the final stats line. The application begins by echoing the setting of the program parameters, followed by error count (should be 0 for err, retry and none).

Max IOPS can be attained by running multiple asyncstress programs concurrently, for example, a read only workload achieves >700K IOPS with two processes(threads) via two accelerator cards.

B.1.3 Sample Output

```
$ asyncstress -d /dev/cxl/afu0.0s -n 5000000 -a 192 -t 1 -b 1000000 -v 1
asyncstress -d /dev/cxl/afu0.0s -n 5000000 -a 192 -t 1 -b 1000000 -v 1
Device open success
lsize = 0, rc, = 0
Device open success
lsize = 0, rc, = 0
bsize = 1000000, rc, = 0
bsize = 1000000, rc, = 0
Unnamed,d,/dev/cxl/afu0.0s,n,5000000,a,192,t,1,b,1000000,v,1,h,0,r,100,w,0,o,0,retry,0
,err,0,none,0,thru,381043,rmin,175,rmax,1311,ravg,501,wmin,0,wmax,0,wavg,0,
Unnamed,d,/dev/cxl/afu1.0s,n,5000000,a,192,t,1,b,1000000,v,1,h,0,r,100,w,0,o,0,retry,0
,err,0,none,0,thru,333241,rmin,150,rmax,25246,ravg,573,wmin,0,wmax,0,wavg,0,
```

Appendix C Performance Tuning

C.1 Processor Performance

C.1.1 SMT Modes

POWER8 supports up to SMT8. To query the current SMT mode, run:

```
ibm@capihost:/var/log$ ppc64_cpu -smt
SMT=4
ibm@capihost:/var/log$
```

To set the SMT mode, use the “ppc64_cpu” command. Refer to its help for additional details:

```
ibm@capihost:/var/log$ ppc64_cpu --smt=<1,2,4,8,on,off>
```

C.1.2 Processor Performance Modes

POWER8 includes hardware-assisted frequency / voltage scaling. This is controlled directly by the operating system in OPAL systems. Systems ship with an “on demand” governor enabled. Best performance may be found if the energy management governor is changed to “performance” mode:

1. Edit the **`/etc/init.d/cpufrequtils`** and change the “`GOVENOR`” line from “`ondemand`” to “`performance`.”
2. Disable the ondemand profile completely:
   `sudo update-rc.d ondemand disable`
3. move to performance without rebooting:
   `sudo /etc/init.d/cpufrequtils start #`
4. Check the governor for each CPU:
   `sudo cpufreq-info | grep decide`

C.1.3 Processor Frequency

POWER8 can dynamically change the frequency for each core independently, based on workload or user configuration. To read the processor frequency, use the `ppc64_cpu` command:

```
ibm@hostname:/tmp$ sudo ppc64_cpu --frequency -t 5
min:   3.325 GHz (cpu 152)
max:   3.325 GHz (cpu 8)
avg:   3.325 GHz
```

C.1.4 Resource Affinity

High bandwidth applications may observe better performance if processes are pinned to specific hardware threads. This may be accomplished with the “numactl --physcpubind” command. The specific allocation of resources will vary depending on the application workload characteristics. For more information about binding CPUs, please refer to linux manpages for numactl.

C.2 Scalability

C.2.1 Increase /proc/sys/vm/max_map_count

This tunable allows a very large number of threads, and may be necessary if a large number of processes simultaneously attempt to access the accelerator cards in the system. This will be needed if an application needs a large number of processes in a single system. Values will vary depending on workload, however a value of 256k may be appropriate.

Appendix D Developer APIs

The IBM DataEngine for NoSQL User Libraries provide two general sets of APIs. Each has a different use case, and targeted applications. Please refer to section 3.3.1 above for an overview of the software components of the system.

Please note that these APIs are evolving. **Text shown in red** is forward-looking, and reflects a change from previously published APIs. These items are preliminary, and may change, however they represent the future direction of the API set.

D.1 Key-Value Library APIs (arkdb.h / arkdb.so)

Key-Value Layer APIs (KV) form the highest level of abstraction to accessing the Data Engine for NoSQL. Usage of these APIs is typically by applications the inherently understand semantics of a key/value store, hash table, or collection API. Example use cases include simple object storage / retrieval (e.g. ark_get or ark_set).

D.1.1 ark_create API

Purpose

Create a key/value store instance

Syntax

int ark_create(file, ark, flags)

char * file;

ARK ** handle;

uint64_t flags;

Description

The **ark_create** API will create a key/value store instance on the host system.

The **path** parameter can be used to specify the special file (e.g. /dev/sdx) representative of the physical LUN created on the flash store. If the **path** parameter is not a special file, the API will assume it is a file to be used for the key/value store. If the file does not exist, it will be created. If **path** is NULL, memory will be used for the key/value store.

The parameter, **flags**, will indicate the properties of the KV store. In the case of specifying a special file for the physical LUN, the user can specify whether the KV store is use the physical LUN as is or to create the KV store in a virtual LUN. By default, the entire physical LUN will be used for the KV store. If a virtual LUN is desired, the **ARK_KV_VIRTUAL_LUN** bit must be set in the flags parameter.

In this revision, a KV store configured to use the entire physical LUN can be persisted. Persistence of a KV store allows the user to shut down an ARK instance and at a later time open the same physical LUN and load the previous ARK instance to the same state as it was when it closed. To configure an ARK instance to be persisted at shut down (ark_delete), set the **ARK_KV_PERSIST_STORE** bit in the flags parameter. By default, an ARK instance is not configured to be persisted. To load the persisted ARK instance resident on the physical LUN, set the **ARK_KV_PERSIST_LOAD** bit in the flags parameter. By default, the persisted store, if present, will not be loaded and will potentially be overwritten by any new persisted data.

In this revision, only physical LUN KV stores can be persisted.

Upon successful completion, the handle parameter will represent the newly created key/value store instance to be used for future API calls.

Parameters

Parameter	Description
path	Allows the user to specify a specific CAPI adapter, a file, or memory for the key/value store.
ark	Handle representing the key/value store
flags	Collection of bit flags determining the properties of the KV store. - **ARK_KV_VIRTUAL_LUN**: KV store will use a virtual LUN created from the physical LUN represented by the special file, **file.** **- ARK_KV_PERSIST_STORE:** Configure the ARK instance to be persisted upon closing (ark_delete) **- ARK_KV_PERSIST_LOAD:** If persistence data is present on the physical LUN, then load the configuration stored.

Return Values

Upon successful completion, the **ark_create** API will return 0, and the handle parameter will point to the newly created key/value store instance. If unsuccessful, the **ark_create** API will return a non-zero error code:

Error	Reason
EINVAL	Invalid value for one of the parameters
ENOSPC	Not enough memory or flash storage
ENOTREADY	System not ready for key/value store supported configuration

D.1.2 ark_delete API

Purpose

Delete a key/value store instance

Syntax

int ark_delete(ark)

ARK * ark;

Description

The **ark_delete** API will delete a key/value store instance, specified by the **ark** parameter, on the host system. Upon successfully completion all associated in memory and storage resources will be released at this time.

If the ARK instance is configured to be persisted, it is at this time the configuration will be persisted so that it may be loaded at a future time.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.

Return Values

Upon successful completion, the **ark_delete** API will clean and remove all resources associated with the key/value store instance and return 0. If unsuccessful, the **ark_delete** API will return a non-zero error code:

Error	Reason
EINVAL	key/value store handle is not valid

D.1.3 ark_set, ark_set_async_cb API

Purpose

Write a key/value pair

Syntax

int ark_set(ark, klen, key, vlen, val, res)

int ark_set_async_cb(ark, klen, key, vlen, val, callback, dt)

ARK * ark;

uint64_t klen;

void * key;

uint64_t vlen;

void * val;

void *(*callback)(int errcode, uint64_t dt, uint64_t res);

uint64_t dt;

Description

The **ark_set** API will store the key, **key**, and value, **val**, into the store for the key/value store instance represented by the **ark** parameter The API, **ark_set_async_cb**, will behave in the same manner, but in an asynchronous fashion, where the API immediately returns to the caller and the actual operation is scheduled to run. After the operation is executed, the **callback** function will be called to notify the caller of completion.

If the **key** is present, the stored value will be replaced with the **val** value.

Upon successful completion, the key/value pair will be present in the store and the number of bytes written will be returned to the caller through the **res** parameter.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance connection
klen	Length of the key in bytes.
key	Key
vlen	Length of the value in bytes.
val	Value
res	Upon success, number of bytes written to the store.
callback	Function to call upon completion of the I/O operation.
dt	64bit value to tag an asynchronous API call.

Return Values

Upon successful completion, the **ark_set** and **ark_set_async_cb**API will write the key/value in the store associated with the key/value store instance and return the number of bytes written. The return of **ark_set** will indicate the status of the operation. The **ark_set_async_cb** API return will indicate whether the asynchronous operation was accepted or rejected. The true status will be stored in the **errcode** parameter when the **callback** function is executed. If unsuccessful, the **ark_set** and **ark_set_async_cb** API will return a non-zero error code:

Error	Reason
EINVAL	Invalid parameter
ENOSPC	Not enough space left in key/value store

D.1.4 ark_get, ark_get_async_cb API

Purpose

Retrieve a value for a given key

Syntax

int ark_get(ark, klen, key, vbuflen, vbuf, voff, res)

int ark_get_async_cb(ark, klen, key, vbuflen, vbuf, voff, callback, dt)

ARK * ark;

uint64_t klen;

void * key;

uint64_t vbuflen;

void * vbuf;

uint64_t voff;

void *(*callback)(int errcode, uint64_t dt, uint64_t res);

uint64_t dt;

Description

The **ark_get** and **ark_get_async_cb** API will query the key/value store associated with the **ark** parameter for the given key, **key**. If found, the key's value will be returned in the **vbuf** parameter with at most **vbuflen** bytes written starting at the offset, **voff,** in the key's value. The API, **ark_get_async_cb**, will behave in the same manner, but in an asynchronous fashion, where the API immediately returns to the caller and the actual operation is scheduled to run. After the operation is executed, the **callback** function will be called to notify the caller of completion.

If successful, the length of the key's value is stored in the **res** parameter of the callback function.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance connection.
klen	Length of the key in bytes.
key	Key
vbuflen	Length of the buffer, vbuf
vbuf	Buffer to store the key's value
voff	Offset into the key to start reading.
res	If successful, will store the size of the key in bytes
callback	Function to call upon completion of the I/O operation.
dt	64bit value to tag an asynchronous API call.

Return Values

Upon successful completion, the **ark_get** and **ark_get_async_cb** API will return 0. The return of **ark_get** will indicate the status of the operation. The **ark_get_async_cb** API return will indicate whether the asynchronous operation was accepted or rejected. The true status of the asynchronous API will be stored in the **errcode** parameter of the **callback** function. If unsuccessful, the **ark_get** and **ark_set_async_cb** API will return a non-zero error code:

Error	Reason
EINVAL	Invalid parameter
ENOENT	Key not found
ENOSPC	Buffer not big enough to hold value

D.1.5 ark_del, ark_del_async_cb API

Purpose

Delete the value associated with a given key

Syntax

int ark_del(ark, klen, key, res)

int ark_del_async_cb(ark, klen, key, callback, dt)

ARK * ark

uint64_t klen;

void * key;

void *(*callback)(int errcode, uint64_t dt, uint64_t res);

uint64_t dt;

Description

The **ark_del** and **ark_del_async_cb** API will query the key/value store associated with the **handle** parameter for the given key, **key,** and if found, will delete the value. The API, **ark_del_async_cb**, will behave in the same manner, but in an asynchronous fashion, where the API immediately returns to the caller and the actual operation is scheduled to run. After the operation is executed, the **callback** function will be called to notify the caller of completion.

If successful, the length of the key's value is returned to the caller in the **res** parameter of the callback function.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance connection.
klen	Length of the key in bytes.
key	Key
res	If successful, will store the size of the key in bytes
callback	Function to call upon completion of the I/O operation.
dt	64bit value to tag an asynchronous API call.

Return Values

Upon successful completion, the **ark_del** and **ark_del_async_cb** API will return 0. The return of **ark_del** will indicate the status of the operation. The **ark_del_async_cb** API return will indicate whether the asynchronous operation was accepted or rejected. The true status will be returned in the **errcode** parameter when the **callback** function is executed. If unsuccessful, the **ark_del** and **ark_del_async_cb** API will return a non-zero error code:

Error	Reason

EINVAL	Invalid parameter
ENOENT	Key not found

D.1.6 ark_exists, ark_exists_async_cb API

Purpose

Query the key/value store to see if a given key is present

Syntax

int ark_exists(ark, klen, key, res)

int ark_exists_async_cb(ark, klen, key, callback, dt)

ARK * ark;

uint64_t klen;

void * key;

void *(*callback)(int errcode, uint64_t dt, uint64_t res);

uint64_t dt;

Description

The **ark_exists** and **ark_exists_async_cb** API will query the key/value store associated with the **ark or arc** parameter for the given key, **key,** and if found, return the size of the value in bytes in the **res** parameter. The key and its value will not be altered. The API, **ark_exists_async_cb**, will behave in the same manner, but in an asynchronous fashion, where the API immediately returns to the caller and the actual operation is scheduled to run. After the operation is executed, the **callback** function will be called to notify the caller of completion.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance connection.
klen	Length of the key in bytes.
key	Key
res	If successful, will store the size of the key in bytes
callback	Function to call upon completion of the I/O operation.

dt	64bit value to tag an asynchronous API call.

Return Values

Upon successful completion, the **ark_exists** and **ark_exists_async_cb** API will return 0. The return of **ark_exists** will indicate the status of the operation. The **ark_exists_async_cb** API return will indicate whether the asynchronous operation was accepted or rejected. The true status will be returned in the **errcode** parameter when the **callback** function is executed. If unsuccessful, the **ark_exists and ark_exists_async_cb** API will return a non-zero error code:

Error	Reason
EINVAL	Invalid parameter
ENOENT	Key not found

D.1.7 ark_first API

Purpose

Return the first key and handle to iterate through store.

Syntax

ARI * ark_first(ark, kbuflen, klen, kbuf)

ARK * ark;

uint64_t kbuflen

int64_t *klen;

void * kbuf;

Description

The **ark_first** API will return the first key found in the store in the buffer, **kbuf**, and the size of the key in **klen**, as long as the size is less than the size of the kbuf, **kbuflen**.

If successful, an iterator handle will be returned to the caller to be used to retrieve the next key in the store by calling the **ark_next** API.

Parameters

Parameter	Description

ark	Handle representing the key/value store instance connection.
kbuflen	Length of the kbuf parameter.
klen	Size of the key returned in kbuf
kbuf	Buffer to hold the key

Return Values

Upon successful completion, the **ark_first** API will return a handle to be used to iterate through the store on subsequent calls using the **ark_next** API. If unsuccessful, the **ark_first** API will return NULL with **errno** set to one of the following:

Error	Reason
EINVAL	Invalid parameter
ENOSPC	kbuf is too small to hold key

D.1.8 ark_next API

Purpose

Return the next key in the store.

Syntax

ARI * ark_next(iter, kbuflen, klen, kbuf)

ARI * iter;

uint64_t kbuflen

int64_t *klen;

void * kbuf;

Description

The **ark_next** API will return the next key found in the store based on the iterator handle, **iter**, in the buffer, **kbuf**, and the size of the key in **klen**, as long as the size is less than the size of the kbuf, **kbuflen.**

If successful, a handle will be returned to the caller to be used to retrieve the next key in the store by calling the **ark_next** API. If the end of the store is reached, a NULL value is returned and errno set to **ENOENT**.

Because of the dynamic nature of the store, some recently written keys may not be returned.

Parameters

Parameter	Description
iter	Iterator handle where to begin search in store
kbuflen	Length of the kbuf parameter.
klen	Size of the key returned in kbuf
kbuf	Buffer to hold the key

Return Values

Upon successful completion, the **ark_next** API will return a handle to be used to iterate through the store on subsequent calls using the **ark_next** API. If unsuccessful, the **ark_next** API will return NULL with **errno** set to one of the following:

Error	Reason
EINVAL	Invalid parameter
ENOSPC	kbuf is too small to hold key
ENOENT	End of the store has been reached.

D.1.9 ark_allocated API

Purpose

Return the number of bytes allocated in the store.

Syntax

int ark_allocated(ark, size)

ARK * ark;

uint64_t *size;

Description

The **ark_allocated** API will return the number of bytes allocated in the store in the **size** parameter.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.
size	Will hold the size of the store in bytes

Return Values

Upon successful completion, the **ark_allocated** API will return 0. If unsuccessful, the **ark_allocated** API will return one of the following error codes.

Error	Reason
EINVAL	Invalid parameter

D.1.10ark_inuse API

Purpose

Return the number of bytes in use in the store.

Syntax

int ark_inuse(ark, size)

ARK * ark;

uint64_t *size;

Description

The **ark_inuse** API will return the number of bytes in use in the store in the **size** parameter.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.
size	Will hold the size of number of blocks in use. Size will be in bytes.

Return Values

Upon successful completion, the **ark_inuse** API will return 0. If unsuccessful, the **ark_inuse** API will return one of the following error codes:

Error	Reason
EINVAL	Invalid parameter

D.1.11 ark_actual API

Purpose

Return the actual number of bytes in use in the store.

Syntax

int ark_actual(ark, size)

ARK * ark;

uint64 * size;

Description

The **ark_actual** API will return the actual number of bytes in use in the store in the size parameter. This differs from the **ark_inuse** API as this takes into account the actual sizes of the individual keys and their values instead of generic allocations based on blocks to store these values.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.
size	Will hold the actual number of bytes in use in the store.

Return Values

Upon successful completion, the **ark_actual** API will return the 0. If unsuccessful, the **ark_actual** API will return one of the following error codes:

Error	Reason
EINVAL	Invalid parameter

D.1.12ark_fork, ark_fork_done API

Purpose

Fork a key/value store for archiving purposes.

Syntax

int ark_fork(ark)

int ark_fork_done(ark)

ARK * handle;

Description

The **ark_fork** and **ark_fork_done** API's are to be called by the parent key/value store process to prepare the key/value store to be forked, fork the child process, and to perform any cleanup once it has been detected the child process has exited.

The **ark_fork** API will fork a child process and upon return, will return the process ID of the child in the parent process, and 0 in the child process. Once the parent detects the child has exited, a call to **ark_fork_done** will be needed to clean up any state from the **ark_fork** call.

Note, the **ark_fork** API will fail if there are any outstanding asynchronous commands.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.

Return Values

Upon successful completion, **ark_fork** and **ark_fork_done** will return 0, otherwise one of the following errors:

Error	Reason
EINVAL	Invalid parameter
EBUSY	Outstanding asynchronous operations
ENOMEM	Not enough space to clone store

D.1.13ark_random API

Purpose

Return a random key from the key/value store.

Syntax

int ark_random(ark, kbuflen, klen, kbuf)

ARK * ark;

uint64_t kbuflen

int64_t *klen;

void * kbuf;

Description

The **ark_random** API will return a random key found in the store based on the handle, **ark**, in the buffer, **kbuf**, and the size of the key in **klen**, as long as the size is less than the size of the kbuf, **kbuflen**.

Parameters

Parameter	Description
ark	Handle representing the key/value store
kbuflen	Length of the kbuf parameter.
klen	Size of the key returned in kbuf
kbuf	Buffer to hold the key

Return Values

Upon successful completion, **ark_random** will 0. Otherwise, **ark_random** will return the following error codes:

Error	Reason
EINVAL	Invalid parameter

D.1.14ark_count API

Purpose

Return the count of the number of keys in the key/value store

Syntax

int ark_count(ark, int *count)

ARK * ark;

int * count;

Description

The **ark_count** API will return the total number of keys in the store based on the handle, ark**,** and store the result in the **count** parameter.

Parameters

Parameter	Description
ark	Handle representing the key/value store instance.
count	Number of keys found in the key/value store.

Return Values

Upon successful completion, **ark_count** will return 0. Otherwise, a non-zero error code will be returned:

Error	Reason
EINVAL	Invalid parameter

D.2 Block APIs (capiblock.h / libcflsh_block.so)

Block APIs provide a fast-path between blocks on Flash and a user-space application. Semantics are similar to a low-level block device (e.g. open / close / read / write). Data may be transient or persistent, depending on which APIs are used. Persistent APIs below are preliminary, and may evolve over time.

D.2.1 cblk_init

Purpose
Initializes CAPI block library

Syntax
```
#include <capiblock.h>

int rc = cblk_int(void *arg, int flags)
```

Parameters

Parameter	Description
arg	Currently unused (set to NULL)
flags	Specifies flags for initialization. Currently set to 0.

Description
The cblk_init API initializes the CAPI block library prior to use. cblk_init must be called before any other API in the library is called.

Return Values
Returns 0 on success; otherwise it is an error.

D.2.2 cblk_term

Purpose
Cleans up CAPI block library resources after the library is no longer used.

Syntax
```
#include <capiblock.h>

int rc = cblk_term(void *arg, int flags)
```

Parameters

Parameter	Description
arg	Currently unused (set to NULL)
flags	Specifies flags for initialization. Currently set to 0.

Description
The cblk_term API terminates the CAPI block library after use.

Return Values
Returns 0 on success; otherwise it is an error.

D.2.3 cblk_open

Purpose
Open a collection of contiguous blocks (currently called a "chunk") on a CAPI flash storage device. for which I/O (read and writes) can be done. A chunk can be thought of as a virtual lun, which the provides access to sectors 0 thru n-1 (where n is the size of the chunk in sectors).

Syntax
```
#include <capiblock.h>

chunk_id_t chunk_id = cblk_open(const char *path, int max_num_requests, int mode,
uint64_t ext_arg, int flags)
```

Parameters

Parameters

Parameter	Description
path	This is the CAPI disk special filename (e.g. /dev/sdX)
max_num_requests	This indicates the maximum number of commands that can be queued to the adapter for this chunk at a given time. If this value is 0, then block layer will choose a default size. If the value specified it too large then the cblk_open request will fail with an ENOMEM errno.
mode	Specifies the access mode for the child process (O_RDONLY, O_WRONLY, O_RDWR).
ext_arg	Reserved for future use
flags	This is a collection of bit flags. The **CBLK_OPN_VIRT_LUN** indicates a virtual lun on the one physical lun will be provisioned. If the **CBLK_OPN_VIRT_LUN** is not specified, then direct access to the full physical lun will be provided. The **CBLK_OPN_VIRT_LUN** flag

	must be set for Redis shards. The **CBLK_OPN_NO_INTRP_THREADS** flag indicates that the cflash block library will not start any back ground threads for processing/harvesting of asynchronous completions from the CAPI adapter. Instead the process using this library must call cblk_aresult to poll for I/O completions.

Description
The cblk_open API creates a "chunk" of blocks on a CAPI flash lun. This chunk will be used for I/O (cblk_read/cblk_write) requests. The returned chunk_id is assigned to a specific path via a specific adapter transparently to the caller. The underlying physical sectors used by a chunk will not be directly visible to users of the block layer.

Upon successful completion, a chunk id representing the newly created chunk instance is returned to the caller to be used for future API calls.

Return Values
Returns NULL_CHUNK_ID on error; otherwise it is a chunk_id handle.

D.2.4 cblk_close

Purpose
Closes a collection of contiguous blocks (called a "chunk") on a CAPI flash storage device. for which I/O (read and writes) can be done.

Syntax
```
#include <capiblock.h>

int rc = cblk_close(chunk_id_t chunk_id, int flags))
```

Description
This service releases the blocks associated with a chunk to be reused by others. Prior to them being reused by others the data blocks will need to be "scrubbed" to remove user data if the **CBLK_SCRUB_DATA_FLG** is set.

Parameters

Parameter	**Description**
chunk_id	The handle for the chunk which is being closed (released for reuse)
flags	This is a collection of bit flags. The **CBLK_SCRUB_DATA_FLG** indicates data blocks should be scrubbed before they can be reused by others.

Return Values
Returns 0 on success; otherwise it is an error.

D.2.5 cblk_get_lun_size

Purpose
Returns the "size" (number of blocks) of the physical lun to which this chunk is associated. This call is not valid for a virtual lun.

Syntax
```
#include <capiblock.h>

int rc = cblk_get_lun_size(chunk_id_t chunk_id, size_t *size, int flags))
```

Description
This service returns the number of blocks of the physical lun associated with this chunk. The cblk_get_lun_size service requires one has done a cblk_open to receive a valid chunk_id.

Parameters

Parameter	Description
chunk_id	The handle for the chunk whose size is about to be changed.
size	Total number of 4K block for this physical lun.
flags	This is a collection of bit flags.

Return Values
Returns 0 on success; otherwise it is error.

D.2.6 cblk_get_size

Purpose
Returns the "size" (number of blocks) assigned to a specific chunk id.

Syntax
```
#include <capiblock.h>

int rc = cblk_get_size(chunk_id_t chunk_id, size_t *size, int flags))
```

Description
This service returns the number of blocks allocated to this chunk. The cblk_get_size service requires one has done a cblk_open to receive a valid chunk_id.

Parameters

Parameter	Description
chunk_id	The handle for the chunk whose size is about to be changed.
size	Number of 4K block to be used for this chunk.
flags	This is a collection of bit flags.

Return Values
Returns 0 on success; otherwise it is error.

D.2.7 cblk_set_size

Purpose
Assign "size" blocks to a specific chunk id. If blocks are already assigned to this chunk id, then one can increase/decrease the size by specifying a larger/smaller size, respectively.

Syntax
```
#include <capiblock.h>

int rc = cblk_set_size(chunk_id_t chunk_id, size_t size, int flags))
```

Description
This service allocates "size" blocks to this chunk. The cblk_set_size call must be done prior to any cblk_read or cblk_write calls to this chunk. The cblk_set_size service requires one has done a cblk_open to receive a valid chunk_id.

If there were blocks originally assigned to this chunk and they are not being reused after cblk_set_size allocates the new blocks and the CBLK_SCRUB_DATA_FLG is set in the flags parameter, then those originally blocks will be "scrubbed" prior to allowing them to be reused by other cblk_set_size operations.

Upon successful completion, the chunk will has LBAs 0 thru size – 1 that can be read/written.

Parameters

Parameter	Description
chunk_id	The handle for the chunk whose size is about to be changed.
size	Number of 4K block to be used for this chunk.
flags	This is a collection of bit flags. The CBLK_SCRUB_DATA_FLG indicates data blocks should be scrubbed before they can be reused by others.

Return Values
Returns 0 on success; otherwise it is error.

D.2.8 cblk_get_stats

Purpose
Returns statistics for a specific chunk id.

Syntax
```
#include <capiblock.h>

typedef struct chunk_stats_s {
    uint64_t max_transfer_size;     /* Maximum transfer size in        */
                                       /* blocks of this chunk.           */
    uint64_t num_reads;             /* Total number of reads issued    */
                                       /* via cblk_read interface         */
    uint64_t num_writes;            /* Total number of writes issued   */
                                       /* via cblk_write interface        */
    uint64_t num_areads;            /* Total number of async reads     */
                                       /* issued via cblk_aread interface */
    uint64_t num_awrites;           /* Total number of async writes    */
                                       /* issued via cblk_awrite interface*/
    uint64_t num_act_reads;         /* Current number of reads active  */
                                       /* via cblk_read interface         */
    uint32_t num_act_writes;        /* Current number of writes active */
                                       /* via cblk_write interface        */
    uint32_t num_act_areads;        /* Current number of async reads   */
                                       /* active via cblk_aread interface */
    uint32_t num_act_awrites;       /* Current number of async writes  */
                                       /* active via cblk_awrite interface*/
    uint32_t max_num_act_writes;    /* High water mark on the maximum  */
                                       /* number of writes active at once */
    uint32_t max_num_act_reads;     /* High water mark on the maximum  */
                                       /* number of reads active at once  */
    uint32_t max_num_act_awrites;   /* High water mark on the maximum  */
                                        /* number of asyync writes active  */
                                        /* at once.                        */
    uint32_t max_num_act_areads;    /* High water mark on the maximum  */
                                       /* number of asyync reads active   */
                                       /* at once.                        */
    uint64_t num_blocks_read;       /* Total number of blocks read     */
    uint64_t num_blocks_written;    /* Total number of blocks written  */
    uint64_t num_errors;            /* Total number of all error       */
                                       /* responses seen                  */
    uint64_t num_aresult_no_cmplt;  /* Number of times cblk_aresult    */
                                       /* returned with no command        */
                                       /* completion                      */
    uint64_t num_retries;           /* Total number of all commmand    */
                                       /* retries.                        */
    uint64_t num_timeouts;          /* Total number of all commmand    */
                                       /* time-outs.                      */
    uint64_t num_fail_timeouts;     /* Total number of all commmand    */
                                       /* time-outs that led to a command */
                                       /* failure.                        */
    uint64_t num_no_cmds_free;      /* Total number of times we didm't */
                                       /* have free command available     */
    uint64_t num_no_cmd_room ;      /* Total number of times we didm't */
                                       /* have room to issue a command to */
```

```
                                      /* the AFU.                         */
    uint64_t num_no_cmds_free_fail; /* Total number of times we didn't */
                                      /* have free command available and */
                                      /* failed a request because of this*/
    uint64_t num_fc_errors;         /* Total number of all FC          */
                                      /* error responses seen            */
    uint64_t num_port0_linkdowns;   /* Total number of all link downs  */
                                      /* seen on port 0.                 */
    uint64_t num_port1_linkdowns;   /* Total number of all link downs  */
                                      /* seen on port 1.                 */
    uint64_t num_port0_no_logins;   /* Total number of all no logins   */
                                      /* seen on port 0.                 */
    uint64_t num_port1_no_logins;   /* Total number of all no logins   */
                                      /* seen on port 1.                 */
    uint64_t num_port0_fc_errors;   /* Total number of all general FC  */
                                      /* errors seen on port 0.          */
    uint64_t num_port1_fc_errors;   /* Total number of all general FC  */
                                       /* errors seen on port 1.          */
    uint64_t num_cc_errors;         /* Total number of all check       */
                                      /* condition responses seen        */
    uint64_t num_afu_errors;        /* Total number of all AFU error   */
                                      /* responses seen                  */
    uint64_t num_capi_false_reads;  /* Total number of all times       */
                                      /* poll indicated a read was ready */
                                      /* but there was nothing to read.  */
    uint64_t num_capi_adap_resets;  /* Total number of all adapter     */
                                      /* reset errors.                   */
    uint64_t num_capi_afu_errors;   /* Total number of all             */
                                      /* CAPI error responses seen       */
    uint64_t num_capi_afu_intrpts;  /* Total number of all             */
                                      /* CAPI AFU interrupts for command */
                                      /* responses seen.                 */
    uint64_t num_capi_unexp_afu_intrpts; /* Total number of all of     */
                                      /* unexpected AFU interrupts       */
    uint64_t num_active_threads;    /* Current number of threads       */
                                       /* running.                        */
    uint64_t max_num_act_threads;   /* Maximum number of threads       */
                                      /* running simultaneously.         */
    uint64_t num_cache_hits;        /* Total number of cache hits      */
                                      /* seen on all reads               */
} chunk_stats_t;


int rc = cblk_get_stats(chunk_id_t chunk_id, chunk_stats_t *stats, int flags))
```

Description
This service returns statistics for a specific chunk_id.

Parameters

Parameter	**Description**
chunk_id	The handle for the chunk whose size is about to be changed.
stats	Address of a chunk_stats_t structure.
flags	This is a collection of bit flags.

Return Values
Returns 0 on success; otherwise it is error.

D.2.9 cblk_read

Purpose
Read 4K blocks from the chunk at the specified logical block address (LBA) into the buffer specified. It should be noted that his LBA is not the same as the LUNs LBA, since the chunk does not necessarily start at the lun's LBA 0

Syntax
```
#include <capiblock.h>

int rc = cblk_read(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
flags));
```

Description
This service reads data from the chunk and places that data into the supplied buffer. This call will block until the read completes with success or error. The cblk_set_size call must be done prior to any cblk_read, cblk_write, cblk_aread, or cblk_awrite calls to this chunk.

Parameters

Parameter	Description
chunk_id	The handle for the chunk which is being read.
buf	Buffer to which data is read into from the chunk must be aligned on 16 byte boundaries.
lba	Logical Block Address (4K offset) inside chunk.
nblocks	Specifies the size of the transfer in 4K sectors. Upper bound is 16 MB for physical lun. Upper bound for virtual lun is 4K
flags	This is a collection of bit flags.

Return Values

Return Value	Description
-1	Error and errno is set for more details
0	No data was read.

n > 0	Number, n, of blocks read.

D.2.10cblk_write

Purpose
Write 4K blocks to the chunk at the specified logical block address (LBA) using the data from the buffer. It should be noted that his LBA is not the same as the LUNs LBA, since the chunk does not start at LBA 0

Syntax
```
#include <capiblock.h>

int rc = cblk_write(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
flags));
```

Description
This service writes data from the chunk and places that data into the supplied buffer. This call will block until the write completes with success or error. The cblk_set_size call must be done prior to any cblk_write calls to this chunk.

Parameters

Parameter	Description
chunk_id	The handle for the chunk which is being written.
buf	Buffer to which data is written from onto the chunk must be aligned on 16 byte boundaries.
lba	Logical Block Address (4K offset) inside chunk.
nblocks	Specifies the size of the transfer in 4K sectors. Upper bound is 16 MB for physical lun. Upper bound for virtual lun is 4K
flags	This is a collection of bit flags.

Return Values

Return Value	Description
-1	Error and errno is set for more details
0	No data was written

n > 0	Number, n, of blocks written.

D.2.11 cblk_aread

Purpose
Read 4K blocks from the chunk at the specified logical block address (LBA) into the buffer specified. It should be noted that his LBA is not the same as the LUNs LBA, since the chunk does not start at LBA 0

Syntax
```
#include <capiblock.h>


typedef enum {
    CBLK_ARW_STATUS_PENDING = 0,  /* Command has not completed      */
    CBLK_ARW_STATUS_SUCCESS = 1   /* Command completed successfully */
    CBLK_ARW_STATUS_INVALID = 2   /* Caller's request is invalid    */
    CBLK_ARW_STATUS_FAIL    = 3   /* Command completed with error   */
} cblk_status_type_t;

typedef struct cblk_arw_status_s {
    cblk_status_type_t status;    /* Status of command              */
                                  /* See errno field for additional */
                                    /* details on failure             */
    size_t    blocks_transferred;/* Number of block transferred by */
                                    /* this request.                */
    int       errno;              /* Errno when status indicates    */
                                    /* CBLK_ARW_STAT_FAIL             */
} cblk_arw_status_t;


int rc = cblk_aread(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
*tag, cblk_arw_status_t *status, int flags));
```

Description
This service reads data from the chunk and places that data into the supplied buffer. This call will not block to wait for the read to complete. A subsequent cblk_aresult call must be invoked to poll on completion. The cblk_set_size call must be done prior to any cblk_aread calls to this chunk.

Parameters

Parameter	Description
chunk_id	The handle for the chunk which is being written.
buf	Buffer to which data is written from onto the chunk must be aligned on 16 byte boundaries.
lba	Logical Block Address (4K offset) inside chunk.

nblocks	Specifies the size of the transfer in 4K sectors. Upper bound is 16 MB for physical lun. Upper bound for virtual lun is 4K
tag	Returned Identifier that allows the caller to uniquely identify each command issued.
status	Address or 64-bit field provided by the caller which the capiblock library will update when a this command completes. This can be used by an application in place of using the cblk_aresult service. It should be noted that the CAPI adapter can not do this directly it would require software threads to update the status region. This field is not used if the **CBLK_OPN_NO_INTRP_THREADS** flags was specified for cblk_open the returned this chunk_id.
flags	This is a collection of bit flags. The **CBLK_ARW_WAIT_CMD_FLAGS** will cause this service to block to wait for a free command to issue the request. Otherwise this service could return a value of -1 with an errno of EWOULDBLOCK (if there is no free command currently available). The **CBLK_ARW_USER_TAG_FLAGS** indicates the caller is specifying a user defined tag for this request. The caller would then need to use this tag with cblk_aresult and set its **CBLK_ARESULT_USER_TAG** flag. The **CBLK_ARW_USER_STATUS_FLAG** indicates the caller has set the status parameter which it expects will be updated when the command completes.

Return Values

Return Value	Description
-1	Error and errno is set for more details

0	Successfully issued
n > 0	Indicates read completed (possibly from cache) and Number, n, of blocks written.

D.2.12cblk_awrite

Purpose
Write one 4K block to the chunk at the specified logical block address (LBA) using the data from the buffer. It should be noted that his LBA is not the same as the LUNs LBA, since the chunk does not start at LBA 0

Syntax

```
#include <capiblock.h>

typedef enum {
    CBLK_ARW_STAT_NOT_ISSUED = 0,  /* Command is has not been issued */
    CBLK_ARW_STAT_PENDING    = 1,  /* Command has not completed      */
    CBLK_ARW_STAT_SUCCESS    = 2   /* Command completed successfully */
    CBLK_ARW_STAT_FAIL       = 3   /* Command completed with error   */
} cblk_status_type_t;

typedef struct cblk_arw_status_s {
    cblk_status_type_t status;    /* Status of command              */
                                  /* See errno field for additional */
                                    /* details on failure             */
    size_t    blocks_transferred;/* Number of block transferred by */
                                    /* this request.                 */
    int       errno;              /* Errno when status indicates    */
                                    /* CBLK_ARW_STAT_FAIL             */
} cblk_arw_status_t;


int rc = cblk_awrite(chunk_id_t chunk_id, void *buf, off_t lba, size_t nblocks, int
*tag, cblk_arw_status_t *status, int flags));
```

Description
This service writes data from the chunk and places that data into the supplied buffer. This call will not block waiting for the write to complete. A subsequent cblk_aresult call must be invoked to poll on completion. The cblk_set_size call must be done prior to any cblk_awrite calls to this chunk.

Parameters

Parameter	Description
chunk_id	The handle for the chunk which is being written.
buf	Buffer to which data is written from onto the chunk must be aligned on 16 byte boundaries.
lba	Logical Block Address (4K offset) inside chunk.

nblocks	Specifies the size of the transfer in 4K sectors. Upper bound is 16 MB for physical lun. Upper bound for virtual lun is 4K
tag	Returned identifier that allows the caller to uniquely identify each command issued.
status	Address or 64-bit field provided by the caller which the capiblock library will update when a this command completes. This can be used by an application in place of using the cblk_aresult service. It should be noted that the CAPI adapter can not do this directly it would require software threads to update the status region. This field is not used if the **CBLK_OPN_NO_INTRP_THREADS** flags was specified for cblk_open the returned this chunk_id.
flags	This is a collection of bit flags. The **CBLK_ARW_WAIT_CMD_FLAGS** will cause this service to block to wait for a free command to issue the request. Otherwise this service could return a value of -1 with an errno of EWOULDBLOCK (if there is no free command currently available). The **CBLK_ARW_USER_TAG_FLAGS** indicates the caller is specifying a user defined tag for this request. The caller would then need to use this tag with cblk_aresult and set its **CBLK_ARESULT_USER_TAG** flag. The **CBLK_ARW_USER_STATUS_FLAG** indicates the caller has set the status parameter which it expects will be updated when the command completes.

Return Values
Returns 0 on success; otherwise it returns -1 and errno is set.

D.2.13cblk_aresult

Purpose
Return status and completion information for asynchronous requests.

Syntax
```
#include <capiblock.h>

rc = cblk_aresult(chunk_id_t chunk_id, int *tag, uint64_t *status, int flags);
```

Description
This service returns an indication if the pending request issued via cblk_aread or cblk_awrite, which may have completed and if so its status.

Parameters

Parameter	Description
chunk_id	The handle for the chunk which is being written.
tag	Pointer to tag caller is waiting for completion. If the CBLK_ARESULT_NEXT_TAG is set, then this field returns the tag for the next asynchronous completion
status	Pointer to status. The status will be returned when a request completes.
flags	Flags passed from the caller to cblk_aresult. The flag **CBLK_ARESULT_BLOCKING** is set by the caller if they want cblk_aresult to block until a command completes (provided there are active commands). If the **CBLK_ARESULT_NEXT_TAG** flag is set, then this call returns whenever any asynchronous I/O request completes. The **CBLK_ARESULT_USER_TAG** flag indicates the caller checking for status of an asynchronous request that was issued with a user specified tag.

Return Values

Return Value	Description
-1	Error and errno is set for more details
0	Returned without error but no tag is set. This may occur if an I/O request has not yet completed and the CBLK_ARESULT_BLOCKING flag is not set.
n > 0	Number, n, of blocks read/written.

D.2.14cblk_clone_after_fork

Purpose
Allows a child process to access the same virtual lun as the parent process. The child process must do this operation immediately after the fork, using the parent's chunk id in order to access that storage. If the child does not do this operation then it will not have any access to the parent's chunk ids.

Syntax
```
#include <capiblock.h>

rc = cblk_clone_after_fork(chunk_id_t chunk_id, int mode, int flags);
```

Description
This service allows a child process to access data from the parents process.

Parameters

Parameter	**Description**
chunk_id	The handle for the chunk which is in use by the parent process. If this call returns successfully, then this chunk id can also be used by the child process.
mode	Specifies the access mode for the child process (O_RDONLY, O_WRONLY, O_RDWR). NOTE: child process can not have greater access than parent. cblk_open is O_RDWR for the initial parent. Descendant processes can have less access.
flags	Flags passed from the caller.

Return Values

Return Value	**Description**
0	The request completed successfully.
-1	Error and errno is set for more details

Printed in the United States of America June 2015

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at **ibm.com®**.

3 June 2015