Oracle

NoSQL Database
C API Reference Guide

12c Release 1

Library Version 12.1.3.0

ORACLE
NOSQL DATABASE

Legal Notice

Published 8/19/2014

Table of Contents

Preface

This document describes the Oracle NoSQL Database C API.

This book is aimed at software engineers responsible for building Oracle NoSQL Database applications.

Conventions Used in this Book

The following typographical conventions are used within this manual:

Information that you are to type literally is presented in `monospaced` font.

Variable or non-literal text is presented in *italics*. For example: "Go to your *KVHOME* directory."

Note

Finally, notes of special interest are represented using a note block such as this.

Chapter 1. Introduction to Oracle NoSQL Database C API

Welcome to the Oracle NoSQL Database C API. This API is intended for use with C and C++ applications that want to access and manage data which is placed in a NoSQL Database Key-Value Store (KV Store).

This manual describes version 12.1.3.0 of the C API library. This version of the library is intended for use with Oracle NoSQL Database version 11.2.2.0.

Library Installation

This API is a wrapper around the native Java NoSQL Database interfaces. This means that in order for you to use the API, you must have a Java virtual machine configured for the machine where your client will run.

For information on how to build the library, see the BUILDING file.

Library Usage

Examples of how to use these APIs are included with your distribution. See the `<package>/examples` directory. `hello.c` shows basic API usage to a store that does not require authentication. `hello_secured.c` shows API usage when working with a secure store.

At a high level, to use this library:

1. Initialize the Java Native Interface framework (JNI) using kv_create_jni_impl() (page 27).
2. Create a store configuration using kv_create_config() (page 24).
3. If you are using a store that requires authentication, define the security properties and the authentication credentials. You define the security properties using kv_create_properties() (page 26), kv_set_property() (page 37), and kv_config_set_security_properties() (page 20). You create authentication credentials using kv_create_password_credentials() (page 25).
4. Open a handle to the store using kv_open_store() (page 31) or kv_open_store_login() (page 32).
5. Perform data read and write operations using a variety of functions which are described in Data Operation Functions (page 43). Note that these functions will sometimes require durability and consistency structures and key/value structures. Functions used to create and manage these types of structures are described in Durability and Consistency Functions (page 143) and Key/Value Pair Management Functions (page 121).

   If you are operating against a store that requires authentication, be prepared to handle `KV_AUTH_FAILURE` errors. When you see these, you should reauthenticate using kv_store_login() (page 39).

   You are also strongly recommended to use the Avro data type for storage of your store's values. Strictly speaking, Avro usage is optional at this time, but in order to take advantage of future features, it will be required. The Avro functions provided by this library are described in Avro Functions (page 112).
6. Once you are done accessing the store, close your store handle using kv_close_store() (page 8). If you logged the handle into the store, this will log your handle out.
7. Release your store configuration structure using kv_release_config() (page 33).
8. Release your JNI implementation using kv_release_impl() (page 35).

Notice that in the above process, you allocate and initialize structures using some kind of a creation function, and you are then responsible for releasing those structures using some kind of a release function. This pattern repeats with few exceptions throughout the API. For example, to create a key, you use kv_create_key() (page 124) and to release it you use kv_release_key() (page 140). In all such cases, you are responsible for releasing resources that you acquire through the use of these APIs.

Thread Safety

In all but a few cases, the data structures created and used by this library are not thread-safe. Therefore, they should not be shared across threads.

The exception to this are `kv_impl_t` and `kv_store_t`, which are thread-safe once they are created. However, you should take care to create and release these in a single-threaded manner.

`kv_impl_t` is created using kv_create_jni_impl() (page 27) and released using kv_release_impl() (page 35). `kv_store_t` is created using kv_open_store() (page 31) and released using kv_close_store() (page 8).

Chapter 2. Store and Library Functions

This chapter describes high-level KV Store functions. That is, they are functions used to operate on the store handle itself (as opposed to functions that operate on the data in the store), or they are functions used to examine the KV Store C library that is in use.

Store and Library Operations

Store Operations	Description
kv_close_store()	Close an Oracle NoSQL Database store
kv_get_open_error()	Returns the error encountered opening the store, if any
kv_open_store()	Open an Oracle NoSQL Database store
kv_open_store_login()	Open an Oracle NoSQL Database store and authenticate
kv_store_login()	Update the login credentials
kv_store_logout()	Log out of the store
Store Configuration	
kv_config_add_host_port()	Identifies an additional helper host
kv_config_add_read_zone(), kv_config_get_read_zones()	Adds a zone used for read operations
kv_config_set_consistency()	Sets the default consistency
kv_config_set_durability()	Sets the default durability policy
kv_config_set_security_properties()	Sets security properties
kv_config_set_request_limits()	Sets store request limits
kv_config_set_timeouts()	Sets store request timeouts
kv_create_config()	Create a store configuration
kv_create_password_credentials()	Creates username/password credentials for store authentication
kv_create_properties()	Create a properties structure
kv_release_config()	Release the store configuration
kv_release_credentials()	Release store authentication credentials
kv_release_properties()	Release a properties structure
kv_set_property()	Sets a property
Large Object Configuration	
kv_config_set_lob_suffix(), kv_config_get_lob_suffix()	Sets/gets the suffix used by LOB keys
kv_config_set_lob_timeout(), kv_config_get_lob_timeout()	Sets/gets the LOB chunk timeout value
kv_config_set_verification_bytes(), kv_config_get_lob_verification_bytes()	Sets/gets the number of bytes used to verify a resumed LOB put operation
Library Operations	
kv_create_jni_impl()	Initialize the JNI layer

Store Operations	Description
kv_create_jni_impl_from_jvm()	Initialize the JNI layer using a pointer to a JVM
kv_get_impl_type()	Return the C API implementation type
kv_release_impl()	Release the JNI structures
kv_version()	Return the library version number
kv_version_c()	Return the version number for the C library

kv_close_store()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_close_store(kv_store_t *store)
```

Closes the store handle, releasing all resources used by the handle. If authentication was performed for this store handle (that is, if kv_open_store_login() (page 32) was used), then the client is logged out before the handle's resources are released. After calling this function, you should never use the handle again, even if an error is returned by this function.

Store handles are opened using kv_open_store() (page 31) or kv_open_store_login() (page 32).

Parameters

store

The **store** parameter is the store handle that you want to close.

See Also

Store and Library Operations (page 6)

kv_config_add_host_port()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_add_host_port(kv_config_t *config,
                        const char *host,
                        int port);
```

Adds an additional host and port pair to the store's configuration. The host/port pair identified here must be for an active node in the store because it is used by the application as a helper host when the application first starts up. Usage of this function is optional. At least one helper host is required, but that helper host is identified when you create the store's configuration using kv_create_config() (page 24).

Parameters

config

The **config** parameter points to the configuration structure to which you want to add a helper host. This structure was intially created using kv_create_config() (page 24).

host

The **host** parameter is the network name of a node belonging to the store.

port

The **port** parameter is the helper host's port number.

See Also

Store and Library Operations (page 6)

kv_config_add_read_zone()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_add_read_zone(kv_config_t *config,
                        const char *read_zone);
```

Adds a zone in which nodes must be located if they are to be used for read operations. If this function has not been called, then read operations can be performed on nodes in any zone.

The specified zone must exist at the time that this configuration is used to open a store handle, or `KV_INVALID_ARGUMENT` is returned when you attempt to open the handle.

Zones specified for read operations can include primary and secondary zones. If the master is not located in any of the specified zones, either because the zones are all secondary zones or because the master node is not currently in one of the specified primary zones, then read operations configured for absolute consistency will fail.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the read zone.

read_zone

The **read_zone** parameter is the name of the zone used to service read requests.

See Also

Store and Library Operations (page 6)

kv_config_get_lob_suffix()

```
#include <kvstore.h>

const char *
kv_config_get_lob_suffix(kv_config_t *config);
```

Retrieves the suffix associated with Large Object (LOB) keys. Keys associated with LOBs must have a trailing suffix string at the end of their final Key component. This requirement permits non-LOB methods to check for inadvertent modifications to LOB objects.

You can set the LOB suffix using kv_config_set_lob_suffix() (page 17).

Parameters

config

The **config** parameter points to the configuration structure from which you want to retrieve the LOB suffix.

See Also

Store and Library Operations (page 6)

kv_config_get_lob_timeout()

```
#include <kvstore.h>

kv_timeout_t
kv_config_get_lob_timeout(const kv_config_t *config);
```

Returns the default timeout value (in ms) associated with chunk access during Large Objects operations. LOBs are read from the store in chunks, and each such chunk must be retrieved within the timeout period identified by this function or an error is returned on the read attempt.

Parameters

config

The **config** parameter points to the configuration structure from which you want to retrieve the LOB timeout value.

See Also

Store and Library Operations (page 6)

kv_config_get_lob_verification_bytes()

```
#include <kvstore.h>

kv_long_t
kv_config_get_lob_verification_bytes(const kv_config_t *config);
```

Returns the number of trailing bytes of a partial LOB that must be verified against the user supplied LOB stream when resuming a LOB put operation. This value is set using the kv_config_set_verification_bytes() (page 19).

Parameters

config

The **config** parameter points to the configuration structure from which you want to retrieve the verification bytes value.

See Also

Store and Library Operations (page 6)

kv_config_get_read_zones()

```
#include <kvstore.h>

kv_read_zones_t *
kv_config_get_read_zones(kv_config_t *config)
```

Retrieves the structure containing the read zones used by this configuration object. A node must belong to one of the zones identified in the `kv_read_zones_t` structure if it is to be used to service read requests. If `NULL` is returned, any node in any zone can be used to service read requests.

The structure returned by this function is as follows:

```
typedef struct {
    kv_int_t kz_num_zones;
    char **kz_zones;
} kv_read_zones_t;
```

You add read zones using kv_config_add_read_zone() (page 10).

Parameters

config

The **config** parameter points to the configuration structure from which you want to retrieve the read zones.

See Also

Store and Library Operations (page 6)

kv_config_set_consistency()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_consistency(kv_config_t *config,
                          kv_consistency_t *consistency);
```

Identifies the default consistency policy to be used by this process. Note that this default can be overridden on a per-operation basis.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the default consistency. This structure was intially created using kv_create_config() (page 24).

consistency

The **consistency** parameter identifies the consistency policy to be used as the default. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), and kv_create_version_consistency() (page 149).

See Also

Store and Library Operations (page 6)

kv_config_set_durability()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_durability(kv_config_t *config,
                         kv_durability_t durability);
```

Identifies the default durability policy to be used by this process. Note that this default can be overridden on a per-operation basis.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the default durability. This structure was intially created using kv_create_config() (page 24).

durability

The **durability** parameter identifies the durability policy to be used as the default. Durability policies are created using kv_create_durability() (page 145).

See Also

Store and Library Operations (page 6)

kv_config_set_lob_suffix()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_lob_suffix(kv_config_t *config,
                         const char *suffix);
```

Sets the suffix associated with Large Object (LOB) keys. Keys associated with LOBs must have a trailing suffix string at the end of their final key component. This requirement permits non-LOB methods to check for inadvertent modifications to LOB objects.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the LOB suffix.

suffix

The **suffix** parameter identifies the suffix you want to use. By default ".lob" is used.

See Also

Store and Library Operations (page 6)

kv_config_set_lob_timeout()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_lob_timeout(kv_config_t *config,
                          kv_timeout_t timeout_ms);
```

Sets the default timeout value associated with chunk access during Large Object operations. LOBs are read from the store in chunks, and each such chunk must be retrieved within the timeout period defined by this function or an error is returned on the read attempt.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the chunk timeout value.

timeout_ms

The **timeout_ms** parameter is the timeout in milliseconds used for LOB chunk reads.

See Also

Store and Library Operations (page 6)

kv_config_set_verification_bytes()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_lob_verification_bytes(kv_config_t *config,
                                     kv_long_t num_bytes);
```

Sets the number of trailing bytes of a partial LOB that must be verified against the user supplied LOB stream when resuming a LOB put operation.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set the number of LOB verification bytes.

num_bytes

The **num_bytes** parameter identifies the number of trailing bytes used to verify a resumed LOB put operation. This number of bytes is taken from the partial LOB that exists from suspending the put operation, and is compared to your LOB input stream.

See Also

Store and Library Operations (page 6)

kv_config_set_security_properties()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_security_properties(kv_config_t *config,
                                  kv_properties_t *props)
```

Sets the security properties for use by the client for authentication to the store. The properties are set to the properties structure using kv_set_property() (page 37).

Use this function only if you opening a handle to a secure store using kv_open_store_login() (page 32).

Parameters

config

The **config** parameter is the configuration structure that you want to configure.

props

The **props** parameter is the properties structure which contains the security properties that you want to set for the store handle.

See Also

Store and Library Operations (page 6)

kv_config_set_request_limits()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_request_limits(kv_config_t *config,
                             kv_int_t max_active_requests,
                             kv_int_t request_threshold_percent,
                             kv_int_t node_limit_percent);
```

Configures the maximum number of requests that this client can have active for a node in the KVStore. Limiting requests in this way helps minimize the possibility of thread starvation in situations where one or more nodes in the store exhibits long service times and as a result retains threads, making them unavailable to service requests to other reachable and healthy nodes.

The long service times can be due to problems at the node itself, or in the network path to that node. The KVS request dispatcher will, whenever possible, minimize use of nodes with long service times automatically, by re-routing requests to other nodes that can handle them. So the mechanism provided by this function offers an additional margin of safety when such re-routing of requests is not possible.

The request limiting mechanism created by this function is only activated when the number of active requests exceeds the threshold specified by the parameter **request_threshold_percent**. Both the threshold and limit parameters provided to this function are expressed as a percentage of **max_active_requests**.

The limits set by this function only matter if the client is mult-threaded.

When the mechanism is active, the number of active requests to a node is not allowed to exceed **node_limit_percent**. Any new requests that would exceed this limit are rejected and the function making the request returns with an error.

For example, consider a configuration with max_active_requests=10, request_threshold_percent=80 and node_limit_percent=50. If 8 requests are already active at the client, and a 9th request is received that would be directed at a node which already has 5 active requests, it would result in an error being raised. If only 7 requests were active at the client, the 8th request would be directed at the node with 5 active requests and the request would be processed normally.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set request limits. This structure was intially created using kv_create_config() (page 24).

max_active_requests

The **max_active_requests** parameter is the maximum number of active requests permitted by the KV client. This number is typically derived from the maximum number of threads that the

client has set aside for processing requests. The default is 100. Note that the KVStore does not actually enforce this maximum directly. It only uses this parameter as the basis for calculating the requests limits to be enforced at a node.

request_threshold_percent

The **request_threshold_percent** parameter is the threshold computed as a percentage of **max_active_requests** at which requests are limited. The default is 90.

node_limit_percent

The **node_limit_percent** parameter determines the maximum number of active requests that can be associated with a node when the request limiting mechanism is active. The default is 80.

See Also

Store and Library Operations (page 6)

kv_config_set_timeouts()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_config_set_timeouts(kv_config_t *config,
                       kv_long_t socket_read_timeout,
                       kv_long_t request_timeout,
                       kv_long_t socket_open_timeout);
```

Configures default timeout values that are used for when this client performs store data access. All timeout values are specified in milliseconds.

Parameters

config

The **config** parameter points to the configuration structure for which you want to set request timeouts. This structure was intially created using kv_create_config() (page 24).

socket_read_timeout

The **socket_read_timeout** parameter configures the read timeout associated with sockets used to make client requests. It applies to both read and write requests, and represents the amount of time the client will wait for a response from the store. Shorter timeouts result in more rapid failure detection and recovery. However, this timeout should be sufficiently long so as to allow for the longest timeout associated with a request.

request_timeout

The **request_timeout** parameter configures the default request timeout. That is, client read and write requests must fully complete within the period of time identified on this parameter, or the request fails with an error.

socket_open_timeout

The **socket_open_timeout** parameter configures the amount of time the client will wait when opening a socket to the store. Shorter timeouts result in more rapid failure detection and recovery.

See Also

Store and Library Operations (page 6)

kv_create_config()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_config(kv_config_t **config,
                 const char *store_name,
                 const char *host,
                 int port)
```

Creates a configuration structure to be used with kv_open_store() (page 31). You release the resources used by this structure using kv_release_config() (page 33), but only if your application encounters an error when kv_open_store() (page 31) is called or when this function returns an error.

Note that you must identify at least one helper host when you call this function, using the **host** and **port** parameters. The helper host is used by the application to locate other nodes in the store. Additional helper hosts can be identified using kv_config_add_host_port() (page 9).

This function defines default client behavior. All of the defaults that you can configure using this function can be overridden on a per-operation basis using the appropriate parameters to this API's put/get/delete functions.

You can only define default store behavior before the store handle is opened; changing these configuration options after open time has no effect on store behavior.

Parameters

config

The **config** parameter references memory into which a pointer to the allocated configuration structure is copied.

store_name

The **store_name** parameter is the name of the KV Store. The store name is used to guard against accidental use of the wrong host or port. The store name must consist entirely of upper or lowercase, letters and digits.

host

The **host** parameter is the network name of a node belonging to the store. The node must be currently active because it is used by the application as a helper host to locate other nodes in the store.

port

The **port** parameter is the helper host's port number.

See Also

Store and Library Operations (page 6)

kv_create_password_credentials()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_password_credentials(kv_impl_t *impl,
                  const char *username,
                  const char *password,
                  kv_credentials_t **creds)
```

Creates a credentials structure with the username/password pair that you want to use to authenticate to a store. These credentials are used with kv_open_store_login() (page 32) or kv_store_login() (page 39).

Release these credentials using kv_release_credentials() (page 34).

Parameters

impl

The **impl** parameter is the implementation structure you are using for the library. It is created using kv_create_jni_impl() (page 27).

username

The **username** parameter is the username you want to use for store authentication.

password

The **password** parameter is the password parameter is the user's password.

creds

The **creds** parameter references memory into which is passed the allocated `kv_credentials_t` structure.

See Also

Store and Library Operations (page 6)

kv_create_properties()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_properties(kv_impl_t *impl,
                     kv_properties_t **props)
```

Creates and allocates resources for a properties structure. Use kv_set_property() (page 37) to set a property to the structure allocated here. Use kv_release_properties() (page 36) to release the structure.

The properties you can set here are used to control the behavior of the underlying Java code. At present, only the Oracle NoSQL Database security properties can be used. See kv_set_property() (page 37) for details. These are set for the store configuration using kv_config_set_security_properties() (page 20).

Parameters

impl

The **impl** parameter is the implementation structure you are using for the library. It is created using kv_create_jni_impl() (page 27).

props

The **props** parameter references memory into which is placed the initialized `kv_properties_t` structure.

See Also

Store and Library Operations (page 6)

kv_create_jni_impl()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_jni_impl(kv_impl_t **impl,
                   const char *classpath)
```

Creates a JNI implementation structure. This object initializes the Java Native Interface layer. Programs written using this implementation will use JNI, and so require a Java runtime to be available in order for the program to run.

The implementation structure is used with kv_open_store() (page 31).

You release the implementation structure using kv_release_impl() (page 35).

Parameters

impl

The **impl** parameter references memory into which a pointer to the allocated implementation structure is placed.

classpath

The **classpath** parameter provides a path to the `kvclient.jar` file, which is required in order to use this library.

See Also

Store and Library Operations (page 6)

kv_create_jni_impl_from_jvm()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_jni_impl_from_jvm(kv_impl_t **impl, void *jvm)
```

Creates a JNI implementation structure based on a JVM instantiation created by the application.. This object initializes the Java Native Interface layer. Programs written using this implementation will use JNI, and so require a Java runtime to be available in order for the program to run.

The implementation structure is used with kv_open_store() (page 31).

You release the implementation structure using kv_release_impl() (page 35).

Parameters

impl

The **impl** parameter references memory into which a pointer to the allocated implementation structure is placed.

jvm

The **jvm** parameter is a pointer to a Java Virtual Machine that was created by the application.

See Also

Store and Library Operations (page 6)

kv_get_impl_type()

```
#include <kvstore.h>

kv_api_type_enum (page 185)
kv_get_impl_type(kv_impl_t *impl)
```

Returns the type of implementation in use by the KVStore C Library. Currently, there is only one possible implementation type: `KV_JNI`.

Parameters

impl

The **impl** parameter is the implementation structure whose type you want to examine.

See Also

Store and Library Operations (page 6)

kv_get_open_error()

```
#include <kvstore.h>

const char *
kv_get_open_error(kv_impl_t *impl);
```

Returns a descriptive string of any error that was encountered opening the store. This method is not thread-safe.

Parameters

impl

The **impl** parameter is the implementation structure containing the open error that you want to examine.

See Also

Store and Library Operations (page 6)

kv_open_store()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_open_store(const kv_impl_t *impl,
              kv_store_t **store,
              kv_config_t *config)
```

Opens a KV Store handle (structure). Call kv_close_store() (page 8) to release the resources allocated for this structure.

The **config** parameter is donated upon success, and so upon a successful open your application should ignore the `kv_config_t` structure. If this function fails, you must call kv_release_config() (page 33) (it is only when an error occurs on store open that you should explicitly release the configuration structure). Upon failure, you might be able to obtain error information using kv_get_open_error() (page 30).

This function is not thread-safe; it must not be called concurrently on the same `kv_impl_t` instance.

Parameters

impl

The **impl** parameter is the implementation structure you are using for the library. It is created using kv_create_jni_impl() (page 27).

store

The **store** parameter references memory into which a pointer to the allocated store handle (structure) is copied.

config

The **config** parameter is the configuration structure that you want to use to configure this handle. It is created using kv_create_config() (page 24).

See Also

Store and Library Operations (page 6)

kv_open_store_login()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_open_store_login(const kv_impl_t *impl,
              kv_store_t **store,
              kv_config_t *config,
              kv_credentials_t *creds)
```

Opens a KV Store handle (structure) and authenticates to the store using the supplied authentication credentials. Call kv_close_store() (page 8) to log out of the store and release the resources allocated for this structure. Note that you can also log out of the store using kv_store_logout() (page 40).

The **config** parameter is donated upon success, and so upon a successful open your application should ignore the `kv_config_t` structure. If this function fails, you must call kv_release_config() (page 33) (it is only when an error occurs on store open that you should explicitly release the configuration structure). Upon failure, you might be able to obtain error information using kv_get_open_error() (page 30).

This function is not thread-safe; it must not be called concurrently on the same `kv_impl_t` instance.

Parameters

impl

The **impl** parameter is the implementation structure you are using for the library. It is created using kv_create_jni_impl() (page 27).

store

The **store** parameter references memory into which a pointer to the allocated store handle (structure) is copied.

config

The **config** parameter is the configuration structure that you want to use to configure this handle. It is created using kv_create_config() (page 24).

creds

The **creds** parameter is the credentials structure that you want to use to login to the store. You create this structure using kv_create_password_credentials() (page 25).

See Also

Store and Library Operations (page 6)

kv_release_config()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_release_config(kv_config_t **config)
```

Releases the resources used by a KV Store configuration object. This function should only be called if an error occurs when kv_open_store() (page 31) or kv_open_store_login() (page 32) is called. The `kv_config_t` structure was initially allocated using kv_create_config() (page 24).

Parameters

config

The **config** parameter is the configuration structure that you want to release.

See Also

Store and Library Operations (page 6)

kv_release_credentials()

```
#include <kvstore.h>

void
kv_release_credentials(kv_credentials_t **creds)
```

Releases password credentials created using kv_create_password_credentials() (page 25)

Parameters

creds

The **creds** parameter references the password credentials you want to release.

See Also

Store and Library Operations (page 6)

kv_release_impl()

```
#include <kvstore.h>

void
kv_release_impl(kv_impl_t **impl)
```

Releases the implementation structure created by kv_create_jni_impl() (page 27).

Parameters

impl

The **impl** parameter is the implementation structure whose resources you want to release.

See Also

Store and Library Operations (page 6)

kv_release_properties()

```
#include <kvstore.h>

void
kv_release_properties(kv_properties_t **props)
```

Releases a properties structure created using kv_create_properties() (page 26).

Parameters

props

The **props** parameter references the properties you want to release.

See Also

Store and Library Operations (page 6)

kv_set_property()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_set_property(kv_properties_t* props,
                const char* prop_name,
                const char* prop_value)
```

Sets a Java property to the `kv_properties_t` structure. At present, only the Oracle NoSQL Database security properties can be set. The resulting properties structure is assigned to the store configuration structure using kv_config_set_security_properties() (page 20).

Parameters

props

The **props** parameter references the properties structure on which you want to set the properties.

props_name

The **props_name** parameter must be a property name. Supported properties are defined in `kvstore.h`:

- `KV_SEC_SECURITY_FILE_PROPERTY`

  Identifies a security property configuration file to be read when a `KVStoreConfig` is created, as a set of overriding property definitions.

- `KV_SEC_TRANSPORT_PROPERTY`

  If set to `ssl`, enables the use of SSL/TLS communications.

- `KV_SEC_SSL_CIPHER_SUITES_PROPERTY`

  Controls what SSL/TLS cipher suites are acceptable for use. The property value is a comma-separated list of SSL/TLS cipher suite names. Refer to your Java documentation for the list of valid values.

- `KV_SEC_SSL_PROTOCOLS_PROPERTY`

  Controls what SSL/TLS protocols are acceptable for use. The property value is a comma-separated list of SSL/TLS protocol names. Refer to your Java documentation for the list of valid values.

- `KV_SEC_SSL_HOSTNAME_VERIFIER_PROPERTY`

  Specifies a verification step to be performed when connecting to a NoSQL DB server when using SSL/TLS. The only verification step currently supported is the `dnmatch` verifier.

The dnmatch verifier must be specified in the form `dnmatch(`*`distinguished-name`*`)`, where *distinguished-name* must be the NoSQL DB server certificate's distinguished name. For a typical secure deployment this should be `dnmatch(CN=NoSQL)`.

- `KV_SEC_SSL_TRUSTSTORE_FILE_PROPERTY`

  Identifies the location of a Java truststore file used to validate the SSL/TLS certificates used by the Oracle NoSQL Database server. This property must be set to an absolute path for the file. If this property is not set, a system property setting of `javax.net.ssl.trustStore` is used.

- `KV_SEC_SSL_TRUSTSTORE_TYPE_PROPERTY`

  Identifies the type of Java truststore that is referenced by the `KV_SEC_SSL_TRUSTSTORE_FILE_PROPERTY` property. This is only needed if using a non-default truststore type. The specified type must be supported by your Java implementation.

- `KV_SEC_SSL_AUTH_USERNAME_PROPERTY`

  Specifies the username used for authentication.

- `KV_SEC_SSL_AUTH_WALLET_PROPERTY`

  Identifies an Oracle Wallet directory containing the password of the user to authenticate. This is only used in the Enterprise Edition of the product.

- `KV_SEC_SSL_AUTH_PWDFILE_PROPERTY`

  Identifies a password store file containing the password of the user to authenticate

prop_value

The **prop_value** parameter must be the property's value.

See Also

Store and Library Operations (page 6)

kv_store_login()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_store_login(kv_store_t *store,
               kv_credentials_t *creds)
```

Updates the login credentials used by the store handle. Use this function under one of the two following circumstances:

1. The application returns KV_AUTH_FAILURE. Calling this function causes the handle to attempt to re-establish the authentication to the store. The credentials used in this case must be for the handle's currently logged in user.
2. If the handle is currently logged out due to a call to kv_store_logout() (page 40), then this function can be used to log in to the store. In that case, login credentials for any valid user can be used.

You create a credentials structure using kv_create_password_credentials() (page 25).

Parameters

store

The **store** parameter references the store handle that you want to use for authentication or reauthentication.

creds

The **creds** parameter is the authentication credentials structure you want to use for this log in attempt.

See Also

Store and Library Operations (page 6)

kv_store_logout()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_store_logout(kv_store_t *store)
```

Logs the store handle out of the store. The handle remains valid, but you should not use it with any functions except kv_close_store() (page 8) or kv_store_login() (page 39). Once logged out, all other functions will return KV_AUTH_FAILURE.

Parameters

store

The **store** parameter references the store handle that you want to log out.

See Also

Store and Library Operations (page 6)

kv_version()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_version(kv_impl_t *impl, kv_int_t *major,
           kv_int_t *minor, kv_int_t *patch)
```

Identifies the Oracle NoSQL Database version information. Versions consist of major, minor and patch numbers.

Parameters

impl

The **impl** is the implementation structure used to open the KV Store. It is created using kv_create_jni_impl() (page 27).

major

The **major** parameter identifies the store's major release number.

minor

The **minor** parameter identifies the store's minor release number.

patch

The **patch** parameter identifies the store's patch release number.

See Also

Store and Library Operations (page 6)

kv_version_c()

```
#include <kvstore.h>

void
kv_version_c(kv_int_t *major, kv_int_t *minor, kv_int_t *patch)
```

Identifies the version information for the C API library. Versions consist of major, minor and patch numbers.

Parameters

major

The **major** parameter identifies the store's major release number.

minor

The **minor** parameter identifies the store's minor release number.

patch

The **patch** parameter identifies the store's patch release number.

See Also

Store and Library Operations (page 6)

Chapter 3. Data Operation Functions

This chapter describes the functions used to perform data read and writes on the store. Functions are described that allow single records to be read or written at a time, and for many records to be read and written in a single operation. Functions that return iterators are also described. These allow you to walk over some or all of the records contained in the store.

This chapter also describes functions used to perform multiple write operations that are organized in a single sequence of operations. That is, you can in one atomic unit perform several put operations that create new records, then put operations that update those records, and/or delete those records. These sequences are performed under a single transaction that effectively offers serializable isolation.

Data Operations and Related Functions

Data Operations	Description
kv_delete()	Delete the key/value pair associated with the key
kv_delete_with_options()	Delete the key/value pair using options
kv_get()	Get the value associated with the key
kv_get_with_options()	Get the value that matches the options
kv_put()	Put a key/value pair, inserting or overwriting as appropriate
kv_put_with_options()	Put the key/value pair using options
Multiple-Key Operations	
kv_init_key_range()	Create and initialize a key range for use in multiple-key operations
kv_init_key_range_prefix()	Create and initialize a key range using prefix information
kv_iterator_next()	Return the next key/value pair from a store iterator
kv_iterator_next_key()	Return the next key from a store iterator
kv_iterator_size()	Return the number of elements in the store iterator
kv_multi_delete()	Delete all descendant key/value pairs associated with the parent key
kv_multi_get()	Return all descendant key/value pairs associated with the parent key
kv_multi_get_iterator()	Returns an iterator that provides traversal of descendant key/value pairs
kv_multi_get_iterator_keys()	Returns an iterator that provides traversal of descendant keys
kv_multi_get_keys()	Return the descendant keys associated with the parent key
kv_parallel_scan_iterator_next()	Return the next key/value pair from a parallel scan iterator
kv_parallel_scan_iterator_next_key()	Return the next key from a parallel scan iterator
kv_parallel_store_iterator()	Return a parallel scan store iterator
kv_parallel_store_iterator_keys()	Return an parallel scan iterator providing traversal of store keys
kv_release_iterator()	Release the store iterator
kv_release_parallel_scan_iterator()	Release the parallel scan iterator

Data Operations	Description
kv_store_iterator()	Return an iterator that provides traversal of all store key/value pairs
kv_store_iterator_keys()	Return an iterator that provides traversal of all store keys
Multi-Step Operations	
kv_create_delete_op()	Create a delete operation to be used with an operation sequence
kv_create_delete_with_options_op()	Create a delete operation, with parameters, to be used with an operation sequence
kv_create_operations()	Creates and initializes a structure used to contain a sequence of store operations
kv_create_put_op()	Create a put operation to be used with an operation sequence
kv_create_put_with_options_op()	Create a put operation, with parameters, to be used with an operation sequence
kv_execute()	Execute the operation
kv_operation_get_abort_on_failure()	Returns whether the entire operation aborts in the event of an execution failure
kv_operation_get_key()	Returns the Key associated with the operation
kv_operation_get_type()	Returns the operation type
kv_operation_results_size()	Returns the size of the operation's result set
kv_operations_set_copy()	Configures a list of operations to copy user-supplied structures and buffers
kv_operations_size()	Returns the number of operations in the operation structure
kv_release_operation_results()	Release the results obtained by executing an operation
kv_release_operations()	Release the operation structure
kv_result_get_previous_value()	For a put or delete operation, returns the previous value associated with the key
kv_result_get_previous_version()	For a put or delete operation, returns the version or the previous value associated with the Key
kv_result_get_success()	Returns whether the operation was successful
kv_result_get_version()	For a put operation, returns the Version of the new key/value pair
Large Object Operations	
kv_lob_delete()	Delete the LOB

Data Operations	Description
kv_lob_get_for_read()	Open a LOB handle for read operations
kv_lob_get_for_write()	Open a LOB handle for write operations
kv_lob_get_version()	Returns the Version of the LOB key/value pair
kv_lob_put_from_file()	Put a LOB key/value pair, inserting or overwriting as appropriate
kv_lob_read()	Read a LOB from the store
kv_lob_release_handle()	Release the LOB handle

kv_create_delete_op()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_delete_op(kv_operations_t *list,
                    const kv_key_t *key,
                    kv_int_t abort_on_failure)
```

Creates a simple delete operation suitable for use as part of a multi-step operation to be run using kv_execute() (page 56). The semantics of the returned operation when executed are identical to that of kv_delete() (page 53).

Parameters

list

The **list** parameter is the operation sequence to which this delete operation is appended. The list is allocated using kv_create_operations() (page 49).

key

The **key** parameter identifies the Key portion of the record to be deleted.

Note that all of the operations performed under a single call to kv_execute() (page 56) must share the same major key path, and that major key path must be complete.

abort_on_failure

The **abort_on_failure** parameter indicates whether the entire operation should abort if this delete operation fails. Specify 1 if you want the operation to abort upon deletion failure.

See Also

Data Operations and Related Functions (page 44)

kv_create_delete_with_options_op()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_delete_with_options_op(kv_operations_t *list,
                      const kv_key_t *key,
                      const kv_version_t *version,
                      kv_return_value_version_enum (page 180) return_info,
                      kv_int_t abort_on_failure)
```

Create a complex delete operation suitable for use as part of a multi-step operation to be run using kv_execute() (page 56). The semantics of the returned operation when executed are identical to that of kv_delete_with_options() (page 54).

Parameters

list

The **list** parameter is the operation sequence to which this delete operation is appended. The sequence is allocated using kv_create_operations() (page 49).

key

The **key** parameter identifies the Key portion of the record to be deleted.

Note that all of the operations performed under a single call to kv_execute() (page 56) must share the same major key path, and that major key path must be complete.

version

The **version** parameter indicates the Version that the record must match before it can be deleted. The Version is obtained using the kv_get_version() (page 139) function.

return_info

The **return_info** parameter indicates what information should be returned to the kv_execute() (page 56) **results** list by this operation. See kv_return_value_version_enum (page 180) for a description of the options accepted by this parameter.

abort_on_failure

The **abort_on_failure** parameter indicates whether the entire operation should abort if this delete operation fails. Specify 1 if you want the operation to abort upon deletion failure.

See Also

Data Operations and Related Functions (page 44)

kv_create_operations()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_operations(kv_store_t *store,
                     kv_operations_t **operations)
```

Allocates a structure containing a multi-step sequence of operations to be performed by kv_execute() (page 56). Release the resources used by this sequence using kv_release_operations() (page 103).

Immediately upon calling this function, you might want to call kv_operations_set_copy() (page 88). See that function's description for details.

Parameters

store

The **store** parameter is the handle to the store in which you want to run the sequence of operations.

operations

The **operations** parameter references memory into which a pointer to the allocated operations structure is copied.

See Also

Data Operations and Related Functions (page 44)

kv_create_put_op()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_put_op(kv_operations_t *list,
                 const kv_key_t *key,
                 const kv_value_t *value,
                 kv_int_t abort_on_failure)
```

Creates a simple put operation suitable for use as part of a multi-step operation to be run using kv_execute() (page 56). The semantics of the returned operation when executed are identical to that of kv_put() (page 97).

Parameters

list

The **list** parameter operation sequence to which this put operation is appended. The sequence is allocated using kv_create_operations() (page 49).

key

The **key** parameter is the Key portion of the Key/Value pair that you want to write to the store.

Note that all of the operations performed under a single call to kv_execute() (page 56) must share the same major key path, and that major key path must be complete.

value

The **value** parameter is the Value portion of the Key/Value pair that you want to write to the store.

abort_on_failure

The **abort_on_failure** parameter indicates whether the entire operation should abort if this put operation fails. Specify 1 if you want the operation to abort upon put failure.

See Also

Data Operations and Related Functions (page 44)

kv_create_put_with_options_op()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_put_with_options_op(kv_operations_t *list,
                    const kv_key_t *key,
                    const kv_value_t *value,
                    const kv_version_t *version,
                    kv_presence_enum (page 180) if_presence,
                    kv_return_value_version_enum (page 180) return_info,
                    kv_int_t abort_on_failure)
```

Creates a complex put operation suitable for use as part of a multi-step operation to be run using kv_execute() (page 56). The semantics of the returned operation when executed are identical to that of kv_put() (page 97).

Parameters

list

The **list** parameter operation sequence to which this put operation is appended. The sequence is allocated using kv_create_operations() (page 49).

key

The **key** parameter is the Key portion of the Key/Value pair that you want to write to the store.

Note that all of the operations performed under a single call to kv_execute() (page 56) must share the same major key path, and that major key path must be complete.

value

The **value** parameter is the Value portion of the Key/Value pair that you want to write to the store.

version

The **version** parameter indicates that the record should be put only if the existing value matches the version supplied to this parameter. Use this parameter when updating a value to ensure that it has not changed since it was last read. The version is obtained using the kv_get_version() (page 139) function.

if_presence

The **if_presence** parameter describes the conditions under which the record can be put, based on the presence or absence of the record in the store. For example, `KV_IF_PRESENT` means that the record can only be written to the store if a version of the record already exists there.

For a list of all the available presence options, see kv_presence_enum (page 180).

return_info

The **return_info** parameter indicates what information should be returned to the kv_execute() (page 56) **results** list by this operation. See kv_return_value_version_enum (page 180) for a description of the options accepted by this parameter.

abort_on_failure

The **abort_on_failure** parameter indicates whether the entire operation should abort if this put operation fails. Specify 1 if you want the operation to abort upon put failure.

See Also

Data Operations and Related Functions (page 44)

kv_delete()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_delete(kv_store_t *store,
          const kv_key_t *key)
```

Delete the key/value pair associated with the key.

Deleting a key/value pair with this method does not automatically delete its children or descendant key/value pairs. To delete children or descendants, use kv_multi_delete() (page 74) instead.

This function uses the default durability and default request timeout. Default durabilities are set using kv_config_set_durability() (page 16). The default request timeout is set using kv_config_set_timeouts() (page 23).

Possible outcomes when calling this method are:

- The KV pair was deleted and the number of records deleted is returned. For this function, a successful return value will always be `0` or `1`.
- The KV pair was not guaranteed to be deleted successfully. A non-success kv_error_t (page 185) error is returned; that is, a negative integer is returned.

Parameters

store

The **store** parameter is the handle to the store in which you want to perform the delete operation.

key

The **key** parameter is the key used to look up the key/value pair to be deleted.

See Also

Data Operations and Related Functions (page 44)

kv_delete_with_options()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_delete_with_options(kv_store_t *store,
                       const kv_key_t *key,
                       const kv_version_t *if_version,
                       kv_value_t **previous_value,
                       kv_return_value_version_enum (page 180) return_info,
                       kv_durability_t durability,
                       kv_timeout_t timeout_ms)
```

Delete the key/value pair associated with the key.

Deleting a key/value pair with this method does not automatically delete its children or descendant key/value pairs. To delete children or descendants, use kv_multi_delete() (page 74) instead.

Possible outcomes when calling this method are:

- The KV pair was deleted and the number of records deleted is returned. For this function, a successful return value will always be `0` or `1`.
- The KV pair was not guaranteed to be deleted successfully. A non-success kv_error_t (page 185) error is returned; that is, a negative integer is returned.

Parameters

store

The **store** parameter is the handle to the store in which you want to perform the delete operation.

key

The **key** parameter is the key used to look up the key/value pair to be deleted.

if_version

The **if_version** parameter indicates the Version that the record must match before it can be deleted. The Version is obtained using the kv_get_version() (page 139) function.

previous_value

The **previous_value** parameter references memory into which is copied the Value portion of the Key/Value pair that this function deleted.

return_info

The **return_info** parameter indicates what information should be returned to the kv_execute() (page 56) **results** list by this operation. See

kv_return_value_version_enum (page 180) for a description of the options accepted by this parameter.

durability

The **durability** parameter provides the durability guarantee to be used with this delete operation. Durability guarantees are created using kv_create_durability() (page 145).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_execute()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_execute(kv_store_t *store,
           const kv_operations_t *list,
           kv_operation_results_t **result,
           kv_durability_t durability,
           kv_timeout_t timeout_ms)
```

Executes a sequence of operations. The operations list is created using kv_create_operations() (page 49), and individual steps in the operation sequence are created using functions such as kv_create_delete_op() (page 47) and kv_create_put_op() (page 50).

Each operation created for this sequence operates on a single key and matches the corresponding Oracle NoSQL Database operation. For example, the operation generated by the kv_create_put_with_options_op() (page 51) function corresponds to the kv_put_with_options() (page 98) function. The argument pattern for creating each operation is similar, but they differ in the following respects:

- The durability argument is not passed to the operations created for this sequence, because that argument applies to the execution of the entire batch of operations and is passed to this function.
- Each individual operation indicates whether the entire sequence of operations should abort if the individual operation is unsuccessful.

Note that all of the operations performed under a single call to kv_execute() (page 56) must share the same major key path, and that major key path must be complete.

Parameters

store

The **store** parameter is the store handle in which you want to run this sequence of operations.

list

The **list** parameter is the list of operations. This list structure is allocated using kv_create_operations() (page 49).

result

The **result** parameter is a list of operations result. Each element in the results list describes the results of executing one of the operations in the sequence of operations.

Use kv_operation_results_size() (page 87) to discover how many elements are in the operation results set.

To determine if a given operation was successful, using kv_result_get_success() (page 106). To determine the version of the key/value pair operated upon by the operation,

use kv_result_get_version() (page 107). To determine the previous version of the key/value pair before the operation was executed, use kv_result_get_previous_version() (page 105). To determine the value of the key/value pair prior to executing the operation, use kv_result_get_previous_value() (page 104).

To release the resources used by the results list, use kv_release_operation_results() (page 102).

durability

The **durability** parameter identifies the durability policy in use when this sequence of operations is executed. All the operations contained within the specified sequence are executed within the scope of a single transaction that effectively provides serializable isolation. The transaction is started and either committed or aborted by this function. This means the operations are all atomic in nature: either they all succeed or the store is left in a state as if none of them had ever been run at all.

Durability policies are created using kv_create_durability() (page 145).

If this parameter is NULL, the store's default durability policy is used.

timeout_ms

The **timeout_ms** parameter identifies the upper bound on the time interval, in milliseconds, for processing the sequence of operations. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used.

See Also

Data Operations and Related Functions (page 44)

kv_get()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_get(kv_store_t *store,
       const kv_key_t *key,
       kv_value_t **valuep)
```

Get the value associated with the key. This function uses the store's default consistency policy and timeout value. To use values other than the defaults, use kv_get_with_options() (page 59) instead.

Parameters

store

The **store** parameter is the handle to the store from which you want to retrieve the value.

key

The **key** parameter is the key portion of the record that you want to read.

valuep

The **valuep** parameter references memory into which is copied the value portion of the retrieved record. Release the resources used by this structure using kv_release_value() (page 141).

This parameter must either be `0`, or it must point to a previously used, not-yet-released `kv_value_t` structure.

See Also

Data Operations and Related Functions (page 44)

kv_get_with_options()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_get_with_options(kv_store_t *store,
                    const kv_key_t *key,
                    kv_value_t **valuep,
                    kv_consistency_t *consistency,
                    kv_timeout_t timeout_ms)
```

Get the value associated with the key. This function allows you to use a non-default consistency policy and timeout value.

Parameters

store

The **store** parameter is the handle to the store from which you want to retrieve the value.

key

The **key** parameter is the key you want to use to look up the key/value pair.

valuep

The **valuep** parameter references memory into which is copied the value portion of the retrieved record. Release the resources used by this structure using kv_release_value() (page 141).

consistency

The **consistency** parameter is the consistency policy you want to use with this operation. You create the consistency policy using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147) or kv_create_version_consistency() (page 149).

If NULL, the store's default consistency policy is used.

timeout_ms

The **timeout_ms** parameter identifies the upper bound on the time interval, in milliseconds, for processing the get operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used.

See Also

Data Operations and Related Functions (page 44)

kv_init_key_range()

```
#include <kvstore.h>

void
kv_init_key_range(kv_key_range_t *key_range,
                  const char *start, kv_int_t start_inclusive,
                  const char *end, kv_int_t end_inclusive)
```

Creates a key range to be used in multiple-key operations and iterations.

The key range defines a range of string values for the key components immediately following the last component of a parent key that is used in a multiple-key operation. In terms of a tree structure, the range defines the parent key's immediate children that are selected by the multiple-key operation.

Parameters

key_range

The **key_range** parameter is the handle to the key range structure.

The `kv_key_range_t` structure contains a series of `const char *` and `int` data members owned by you. For this reason, a release function is not needed for this structure. However, you should not free the strings until you are done using the key range.

start

The **start** parameter defines the lower bound of the key range. If NULL, no lower bound is enforced.

You must not free this string until you are done with the **key_range** created by this function.

start_inclusive

The **start_inclusive** parameter indicates whether the value specified to **start** is included in the range. Specify `1` if it is included; `0` otherwise.

end

The **end** parameter defines the upper bound of the key range. If NULL, no upper bound is enforced.

You must not free this string until you are done with the **key_range** created by this function.

end_inclusive

The **end_inclusive** parameter indicates whether the value specified to **end** is included in the range. Specify `1` if it is included; `0` otherwise.

See Also

Data Operations and Related Functions (page 44)

kv_init_key_range_prefix()

```
#include <kvstore.h>

void
kv_init_key_range_prefix(kv_key_range_t *key_range,
                         const char *prefix)
```

Creates a key range based on a single string that defines the range's prefix. Using this function is the equivalent to using the kv_init_key_range() (page 60) function like this:

```
kv_init_key_range(key_range_p, prefix_str, 1, prefix_str, 1);
```

The key range defined here is for use with multiple-key operations and iterations.

Parameters

key_range

The **key_range** parameter is the handle to the key range structure.

The `kv_key_range_t` structure contains a series of `const char *` and `int` data members owned by you. For this reason, a release function is not needed for this structure. However, you should not free the strings until you are done using the key range.

prefix

The **prefix** parameter is the string that defines both the lower and upper bounds, inclusive, of the key range.

You must not free this string until you are done with the **key_range** created by this function.

See Also

Data Operations and Related Functions (page 44)

kv_iterator_next()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_iterator_next(kv_iterator_t *iterator,
                 const kv_key_t **key,
                 const kv_value_t **value)
```

Returns the iterator's next record. If another record exists, this function returns KV_SUCCESS, and the **key** and **value** parameters are populated. If there are no more records, the return value is KV_NO_SUCH_OBJECT. If the return value is something other than KV_SUCCESS or KV_NO_SUCH_OBJECT, there was an operational failure.

Parameters

iterator

The **iterator** parameter is the handle to the iterator. It is allocated using one of functions that performs multiple reads of the store (such as kv_multi_get() (page 76)). It is released using kv_release_iterator() (page 100).

key

The **key** parameter references memory in which a pointer to the next key is copied.

Note, you should *not* release this key structure. The resources used here will be released when the iterator is released.

value

The **value** parameter references memory in which a pointer to the next value is copied.

Note, you should *not* release this value structure. The resources used here will be released when the iterator is released.

See Also

Data Operations and Related Functions (page 44)

kv_iterator_next_key()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_iterator_next_key(kv_iterator_t *iterator,
                     const kv_key_t **key)
```

Returns the iterator's next key. If another key exists, this function returns KV_SUCCESS, and the **key** parameter is populated. If there are no more keys, the return value is KV_NO_SUCH_OBJECT. If the return value is something other than KV_SUCCESS or KV_NO_SUCH_OBJECT, there was an operational failure.

Parameters

store

The **store** parameter is the handle to the store to which the iterator belongs.

iterator

The **iterator** parameter is the handle to the iterator. It is allocated using one of functions that performs multiple reads of the store (such as kv_multi_get() (page 76)). It is released using kv_release_iterator() (page 100).

key

The **key** parameter references memory in which a pointer to the next key is copied.

Note, you should *not* release this key structure. The resources used here will be released when the iterator is released.

See Also

Data Operations and Related Functions (page 44)

kv_iterator_size()

```
#include <kvstore.h>

kv_int_t
kv_iterator_size(const kv_iterator_t *iterator)
```

Returns the number of items contained in the iterator. This function can only be used with iterators returned by the `kv_multi_*` line of functions. The iterators returned by other functions, such as kv_store_iterator() (page 108), are not usable by this function.

Parameters

store

The **store** parameter is the handle to the store to which the iterator belongs.

iterator

The **iterator** parameter is the handle to the iterator for which you want sizing information.

See Also

Data Operations and Related Functions (page 44)

kv_lob_delete()

```
#include <kvstore.h>

kv_int_t
kv_lob_delete(kv_store_t *store,
              const kv_key_t *key,
              kv_durability_t durability,
              kv_timeout_t timeout_ms);
```

Deletes the Large Object record from the store.

Parameters

store

The **store** parameter is the handle to the store from which you want to delete the LOB.

key

The **key** parameter is the key used to look up the key/value pair to be deleted.

durability

The **durability** parameter provides the durability guarantee to be used with this delete operation. Durability guarantees are created using kv_create_durability() (page 145).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_lob_get_for_read()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_lob_get_for_read(kv_store_t *store,
                    const kv_key_t *key,
                    kv_lob_handle_t **handle,
                    kv_consistency_t *consistency,
                    kv_timeout_t timeout_ms);
```

Allocates and configures a LOB handle for reading a Large Object from the store. If the handle is successfully created, KV_SUCCESS is returned; otherwise, KV_NO_MEMORY.

Upon opening this handle, you perform the actual read operation using kv_lob_read() (page 72).

The LOB handle allocated by this function must be released using kv_lob_release_handle() (page 73).

Parameters

store

The **store** parameter is the handle to the store from which you want to read the LOB record.

key

The **key** parameter is the LOB record's key. Note that the final path component used here must specify the LOB suffix configured for the store, or the read operation will fail. The LOB suffix is configured for your store using kv_config_set_lob_suffix() (page 17).

handle

The **handle** parameter references memory into which a pointer to the allocated LOB handle (structure) is copied.

consistency

The **consistency** parameter is the consistency policy you want to use with this read operation. You create the consistency policy using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147) or kv_create_version_consistency() (page 149).

If NULL, the store's default consistency policy is used.

timeout_ms

The **timeout_ms** parameter identifies the upper bound on the time interval, in milliseconds, for processing the get operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used.

See Also

Data Operations and Related Functions (page 44)

kv_lob_get_for_write()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_lob_get_for_write(kv_store_t *store,
                     const kv_key_t *key,
                     kv_lob_handle_t **handle,
                     kv_presence_enum if_presence,
                     kv_durability_t durability,
                     kv_timeout_t timeout_ms);
```

Allocates and configures a LOB handle for writing a Large Object to the store. If the handle is successfully created, KV_SUCCESS is returned; otherwise, KV_NO_MEMORY.

Upon opening this handle, you perform the actual write operation using kv_lob_put_from_file() (page 71). Note that no method currently exists for writing a large object directly from memory.

The LOB handle allocated by this function must be released using kv_lob_release_handle() (page 73).

Parameters

store

The **store** parameter is the handle to the store where you want to write the Large Object.

key

The **key** parameter is the LOB record's key. Note that the final path component used here must specify the LOB suffix configured for the store, or the write operation will fail. The LOB suffix is configured for your store using kv_config_set_lob_suffix() (page 17).

handle

The **handle** parameter references memory into which a pointer to the allocated LOB handle (structure) is copied.

if_presence

The **if_presence** parameter describes the conditions under which the record can be written to the store, based on the presence or absence of the record in the store. For example, KV_IF_PRESENT means that the record can only be written to the store if a version of the record already exists there.

For a list of all the available presence options, see kv_presence_enum (page 180).

durability

The **durability** parameter provides the durability guarantee to be used with this write operation. Durability guarantees are created using kv_create_durability() (page 145).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for writing each chunk of the LOB. (Large Objects are written to the store in multiple chunks.) A best effort is made not to exceed the specified limit. If zero, the default LOB timeout value defined for the store is used. This value is set using kv_config_set_lob_timeout() (page 18).

See Also

Data Operations and Related Functions (page 44)

kv_lob_get_version()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_lob_get_version(kv_lob_handle_t *handle,
                   kv_version_t **version);
```

Returns the record's version. The version is owned by the handle and must not be released independently of the handle. If no version is available, `KV_INVALID_ARGUMENT` is returned.

The record's version is available immediately upon handle creation using kv_lob_get_for_read() (page 66). If kv_lob_get_for_write() (page 68) is used, the version is available only after the LOB has been written to the store (using kv_lob_put_from_file() (page 71)).

Parameters

handle

The **handle** parameter is the LOB handle from which you want to retrieve the record's version.

version

The **version** parameter references memory into which the value is copied. Release the resources used by this value using kv_release_version() (page 142).

See Also

Data Operations and Related Functions (page 44)

kv_lob_put_from_file()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_lob_put_from_file(kv_lob_handle_t *handle,
                     const char *path_to_file);
```

Writes the Large Object stored in **path_to_file** to the store. The key used for the resulting record is the key that was used to create the `kv_lob_handle_t` (using kv_lob_get_for_write() (page 68)). The handle must have been open for writing, or `KV_INVALID_ARGUMENT` is returned.

If the put is successful, `KV_SUCCESS` is returned.

The object is written to the store in chunks. Each chunk must be written to the store within the timeout period defined when the `kv_lob_handle_t` was created, or the put will fail.

When the handle is created, it is possible to specify restrictions on the write depending on whether the LOB currently exists in the store (using the **if_presence** parameter for kv_lob_get_for_write() (page 68)). If `KV_IF_PRESENT` was specified and the key does not exist, `KV_KEY_NOT_FOUND` is returned. If `KV_IF_ABSENT` was specified and the key exists, `KV_KEY_EXISTS` is returned.

Parameters

handle

The **handle** parameter is the handle to the store where you want to write the LOB.

path_to_file

The **path_to_file** parameter is the filesystem path to the file that contains the LOB value that you want to write to the store.

See Also

Data Operations and Related Functions (page 44)

kv_lob_read()

```
#include <kvstore.h>

kv_int_t
kv_lob_read(kv_lob_handle_t *handle,
            kv_long_t offset,
            kv_int_t num_bytes_to_read,
            unsigned char *buffer);
```

Performs a read of a single chunk, or portion, of a LOB from the store into a buffer. The `kv_lob_handle_t` must have been opened for read (using kv_lob_get_for_read() (page 66)) or `KV_INVALID_ARGUMENT` is returned. Otherwise, the number of bytes read is returned. The end of the LOB is indicated by a return value of `0`. A negative return value indicates an error on the read.

Parameters

handle

The **handle** parameter is the LOB handle that you want to use to perform the read. It must have been created using kv_lob_get_for_read() (page 66).

offset

The **offset** parameter is the offset into the LOB where this read is to begin.

num_bytes_to_read

The **num_bytes_to_read** parameter is the number of bytes you want to read from the LOB.

buffer

The **buffer** parameter is the user-supplied buffer into which the LOB chunk is placed. This buffer must be at least **num_bytes_to_read** bytes in size.

See Also

Data Operations and Related Functions (page 44)

kv_lob_release_handle()

```
#include <kvstore.h>

void
kv_lob_release_handle(kv_lob_handle_t **handle);
```

Release a LOB handle that was allocated by kv_lob_get_for_read() (page 66) or kv_lob_get_for_write() (page 68).

Parameters

handle

The **handle** parameter is the LOB handle that you want to release.

See Also

Data Operations and Related Functions (page 44)

kv_multi_delete()

```
#include <kvstore.h>

kv_int_t
kv_multi_delete(kv_store_t *store, const kv_key_t *parent_key,
                const kv_key_range_t *key_range,
                kv_depth_enum (page 179) depth, kv_durability_t durability,
                kv_timeout_t timeout_ms)
```

Deletes the descendent key/value pairs associated with the **parent_key**. Returns the total number of keys deleted. If an error, returns an integer value less than zero.

All of the deletions performed as a result of this operation are performed within the context of a single transaction. This means that either all matching key/value pairs are deleted, or none of them are.

Parameters

store

The **store** parameter is the handle to the store in which you want to perform the delete operation.

parent_key

The **parent_key** parameter is the parent key whose "child" records are to be deleted. It must not be NULL. The major key path must be complete. The minor key path may be omitted or may be a partial path.

You construct a key using kv_create_key() (page 124).

key_range

The **key_range** parameter further restricts the range under the **parent_key** to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the deletion can go. You can allow only children to be deleted, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

durability

The **durability** parameter identifies the durability to be used for this write operation. Durability guarantees are created using kv_create_durability() (page 145).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_multi_get()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_multi_get(kv_store_t *store,
             const kv_key_t *parent_key,
             kv_iterator_t **return_iterator,
             const kv_key_range_t *sub_range,
             kv_depth_enum (page 179) depth,
             kv_consistency_t *consistency,
             kv_timeout_t timeout_ms)
```

Returns the descendant key/value pairs associated with the **parent_key**. The **sub_range** and the **depth** arguments can be used to further limit the key/value pairs that are retrieved. The key/value pairs are fetched within the scope of a single transaction that effectively provides serializable isolation.

This API should be used with caution because it could result in errors due to running out of memory, or excessive garbage collection activities in the underlying Java virtual machine, if the results cannot all be held in memory at one time. Consider using kv_multi_get_iterator() (page 78) instead.

This function only allows fetching key/value pairs that are descendants of a **parent_key** that has a complete major path. To fetch the descendants of a **parent_key** with a partial major path, use kv_store_iterator() (page 108) instead.

Parameters

store

The **store** parameter is the handle to the store from which you want to retrieve key/value pairs.

parent_key

The **parent_key** parameter is the parent key whose "child" records are to be retrieved. It must not be NULL. The major key path must be complete. The minor key path may be omitted or may be a partial path.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

The iterator returned here is transactional, which means the contents of the iterator will be static (isolated) until such a time as the iterator is released.

sub_range

The **sub_range** parameter further restricts the range under the **parent_key** to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_multi_get_iterator()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_multi_get_iterator(kv_store_t *store,
                      const kv_key_t *parent_key,
                      kv_iterator_t **return_iterator,
                      const kv_key_range_t *sub_range,
                      kv_depth_enum (page 179) depth,
                      kv_direction_enum (page 179) direction,
                      int batch_size,
                      kv_consistency_t *consistency,
                      kv_timeout_t timeout_ms)
```

Returns an iterator that permits an ordered traversal of the descendant key/value pairs associated with the **parent_key**. It is useful when the expected result set is too large to fit in memory. Note that the result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of key/value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

This function requires the **parent_key** to not be NULL, and to have a complete major key path. If you want to obtain an iterator based on a NULL key, or on a key with a partial major key path, use kv_store_iterator() (page 108) instead.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" records are to be retrieved. It must not be NULL. The major key path must be complete. The minor key path may be omitted or may be a partial path.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

sub_range

The **sub_range** parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which records are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_FORWARD` and `kv_direction_enum.KV_DIRECTION_REVERSE` are supported by this function.

batch_size

The **batch_size** parameter specifies the suggested number of keys to fetch during each network round trip. If only the first or last key-value pair is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_multi_get_iterator_keys()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_multi_get_iterator_keys(kv_store_t *store,
                           const kv_key_t *parent_key,
                           kv_iterator_t **return_iterator,
                           const kv_key_range_t *sub_range,
                           kv_depth_enum (page 179) depth,
                           kv_direction_enum (page 179) direction,
                           int batch_size,
                           kv_consistency_t *consistency,
                           kv_timeout_t timeout_ms)
```

Returns an iterator that permits an ordered traversal of the descendant keys associated with the **parent_key**. It is useful when the expected result set is too large to fit in memory. Note that the result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of key/value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

This function requires the **parent_key** to not be NULL, and to have a complete major key path. If you want to obtain an iterator based on a NULL key, or on a key with a partial major key path, use kv_store_iterator_keys() (page 110) instead.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" keys are to be retrieved. It must not be NULL. The major key path must be complete. The minor key path may be omitted or may be a partial path.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

sub_range

The **sub_range** parameter parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which keys are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_FORWARD` and `kv_direction_enum.KV_DIRECTION_REVERSE` are supported by this function.

batch_size

The **batch_size** parameter specifies the suggested number of keys to fetch during each network round trip. If only the first or last key is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_multi_get_keys()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_multi_get_keys(kv_store_t *store,
                  const kv_key_t *parent_key,
                  kv_iterator_t **return_iterator,
                  const kv_key_range_t *sub_range,
                  kv_depth_enum (page 179) depth,
                  kv_consistency_t *consistency,
                  kv_timeout_t timeout_ms)
```

Returns the descendant keys associated with the **parent_key**. The **sub_range** and the **depth** arguments can be used to further limit the keys that are retrieved. The keys are fetched within the scope of a single transaction that effectively provides serializable isolation.

This API should be used with caution because it could result in errors due to running out of memory, or excessive garbage collection activities in the underlying Java virtual machine, if the results cannot all be held in memory at one time. Consider using kv_multi_get_iterator_keys() (page 80) instead.

This function only allows fetching keys that are descendants of a **parent_key** that has a complete major path. To fetch the descendants of a **parent_key** with a partial major path, use kv_store_iterator_keys() (page 110) instead.

Parameters

store

The **store** parameter is the handle to the store from which you want to retrieve keys.

parent_key

The **parent_key** parameter is the parent key whose "child" keys are to be retrieved. It must not be NULL. The major key path must be complete. The minor key path may be omitted or may be a partial path.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

sub_range

The **sub_range** parameter further restricts the range under the **parent_key** to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_operation_get_abort_on_failure()

```
#include <kvstore.h>

kv_int_t
kv_operation_get_abort_on_failure(const kv_operations_t *operations,
                                  kv_int_t index)
```

Returns whether a failure for the identified operation causes the entire sequence of operations to fail. If the entire sequence will fail, then this function returns `1`. If the **index** parameter is out of range, this function returns `KV_NO_SUCH_OBJECT`.

Parameters

operations

The **operations** parameter is the operation sequence to which the operation in question belongs.

index

The **index** parameter identifies the exact operation in the sequence that you want to examine.

See Also

Data Operations and Related Functions (page 44)

kv_operation_get_key()

```
#include <kvstore.h>

const kv_key_t *
kv_operation_get_key(const kv_operations_t *operations,
                     kv_int_t index)
```

Returns the key associated with the operation. If the **index** parameter is out of range, this function returns `NULL`.

Parameters

operations

The **operations** parameter is the operation sequence to which the operation in question belongs.

index

The **index** parameter identifies the exact operation in the sequence that you want to examine.

See Also

Data Operations and Related Functions (page 44)

kv_operation_get_type()

```
#include <kvstore.h>

kv_operation_enum
kv_operation_get_type(const kv_operations_t *operations,
                      kv_int_t index)
```

Returns the type of operation being performed in a particular step in an operation sequence. This function can return one of the following values:

- `KV_NO_SUCH_OBJECT`

  The **index** parameter is out of range.

- `KV_OP_DELETE`

  The operation was created using kv_create_delete_op() (page 47).

- `KV_OP_DELETE_IF_VERSION`

  The operation was created using kv_create_delete_with_options_op() (page 48).

- `KV_OP_PUT`

  The operation was created using kv_create_put_op() (page 50).

- `KV_OP_PUT_WITH_OPTIONS`

  The operation was created using kv_create_put_with_options_op() (page 51).

Parameters

operations

The **operations** parameter is the operation sequence to which the operation in question belongs.

index

The **index** parameter identifies the exact operation in the sequence that you want to examine.

See Also

Data Operations and Related Functions (page 44)

kv_operation_results_size()

```
#include <kvstore.h>

kv_int_t
kv_operation_results_size(const kv_operation_results_t *results)
```

Returns the number of results in the results set created by running kv_execute() (page 56).

Parameters

results

The **results** parameter is the results set whose size you want to obtain.

See Also

Data Operations and Related Functions (page 44)

kv_operations_set_copy()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_operations_set_copy(kv_operations_t **operations)
```

Configures the supplied kv_operations_t so that user-supplied buffers, strings and structures are copied. Normally, buffers, text strings, and structures provided to this API are owned by the application. (The API simply points to the memory in question when making use of it.) This is highly efficient for short-lived user-supplied memory because it avoids memory allocation/copying of the memory's content.

However, if there is a need to retain the data supplied by the user for longer periods of time (beyond which the user-supplied memory might go out of scope, or otherwise be reused or even deallocated), then call this function immediately after creating the operations list. Doing so will cause the API to copy the contents of all user-supplied memory to memory owned by the API. In this way, the application can do whatever is appropriate with the memory it supplies, while the API is able to retain the contents of that memory for however long it needs that content.

As an example, suppose you were creating an operations list by iterating over records in the store, like this:

```
// Create an operation
// Store open skipped for brevity
kv_create_operations(store, &operations);

// Tell the operations list to copy keys/values
kv_operations_set_copy(operations);

// Create a store iterator that walks over every record in the store
err = kv_store_iterator(store, NULL, &iter, NULL, 0,
                        KV_DIRECTION_UNORDERED, 0, NULL, 0);
if (err != KV_SUCCESS) {
    fprintf(stderr, "Error obtaining store iterator: %d\n", err);
    goto done;
}

// Step through the iterator, doing work on each record's value.
// If kv_operations_set_copy() had not been called, iter_key and
// iter_value would go out of scope with each step through the store
// iterator. This would cause unpredictable results when it came time
// to execute the sequence of operations.
while (kv_iterator_next(iter,
                        (const kv_key_t **)&iter_key,
                        (const kv_value_t **)&iter_value)
                         == KV_SUCCESS) {
    // Do some work to iter_value
    kv_create_put_op(operations,  iter_key,  iter_value, 0);
```

```
}

if (iter)
    kv_release_iterator(&iter);

kv_execute(store, operations, &results, 0, 0);
```

Parameters

operations

The **operations** parameter references the operations list which you want to configure for copy.

See Also

Data Operations and Related Functions (page 44)

kv_operations_size()

```
#include <kvstore.h>

kv_int_t
kv_operations_size(const kv_operations_t *operations)
```

Returns the number of operations in the operation sequence.

Parameters

operations

The **operations** parameter is the operation sequence whose size you want to obtain.

See Also

Data Operations and Related Functions (page 44)

kv_parallel_scan_iterator_next()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_scan_iterator_next(kv_parallel_scan_iterator_t *iterator,
                 const kv_key_t **key,
                 const kv_value_t **value)
```

Returns the iterator's next record. If another record exists, this function returns KV_SUCCESS, and the **key** and **value** parameters are populated. If there are no more records, the return value is KV_NO_SUCH_OBJECT. If the return value is something other than KV_SUCCESS or KV_NO_SUCH_OBJECT, there was an operational failure.

Parameters

iterator

The **iterator** parameter is the handle to the iterator. It is allocated using kv_parallel_store_iterator() (page 93). It is released using kv_release_parallel_scan_iterator() (page 101).

key

The **key** parameter references memory in which a pointer to the next key is copied.

Note, you should *not* release this key structure. The resources used here will be released when the iterator is released.

value

The **value** parameter references memory in which a pointer to the next value is copied.

Note, you should *not* release this value structure. The resources used here will be released when the iterator is released.

See Also

Data Operations and Related Functions (page 44)

kv_parallel_scan_iterator_next_key()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_scan_iterator_next_key(
                     kv_parallel_scan_iterator_t *iterator,
                     const kv_key_t **key)
```

Returns the iterator's next key. If another key exists, this function returns KV_SUCCESS, and the **key** parameter is populated. If there are no more keys, the return value is KV_NO_SUCH_OBJECT. If the return value is something other than KV_SUCCESS or KV_NO_SUCH_OBJECT, there was an operational failure.

Parameters

iterator

The **iterator** parameter is the handle to the iterator. It is allocated using one of functions that performs multiple reads of the store (such as kv_multi_get() (page 76)). It is released using kv_release_iterator() (page 100).

key

The **key** parameter references memory in which a pointer to the next key is copied.

Note, you should *not* release this key structure. The resources used here will be released when the iterator is released.

See Also

Data Operations and Related Functions (page 44)

kv_parallel_store_iterator()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_store_iterator(kv_store_t *store,
          const kv_key_t *parent_key,
          kv_parallel_scan_iterator_t **return_parallel_scan_iterator,
          const kv_key_range_t *sub_range,
          kv_depth_enum (page 179) depth,
          kv_direction_enum (page 179) direction,
          int batch_size,
          kv_consistency_t *consistency,
          kv_timeout_t timeout_ms,
          kv_store_iterator_config_t (page 185) *store_iterator_config)
```

Creates a parallel scan iterator which iterates over all key/value pairs in unsorted order. The result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of key/value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" records are to be retrieved. May be NULL, or may have only a partial major key path.

return_parallel_scan_iterator

The **return_parallel_scan_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_parallel_scan_iterator() (page 101).

sub_range

The **sub_range** parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which records are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_UNORDERED` is supported by this function.

batch_size

The **batch_size** parameter provides the suggested number of records to fetch during each network round trip. If only the first or last key/value pair is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

store_iterator_config

The **store_iterator_config** parameter configures the parallel scan operation. Both the maximum number of requests and the maximum number of results batches can be specified using a kv_store_iterator_config_t (page 185) structure.

See Also

Data Operations and Related Functions (page 44)

kv_parallel_store_iterator_keys()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_store_iterator_keys(kv_store_t *store,
          const kv_key_t *parent_key,
          kv_parallel_scan_iterator_t **return_parallel_scan_iterator,
          const kv_key_range_t *sub_range,
          kv_depth_enum (page 179) depth,
          kv_direction_enum (page 179) direction,
          int batch_size,
          kv_consistency_t *consistency,
          kv_timeout_t timeout_ms,
          kv_store_iterator_config_t (page 185) *store_iterator_config)
```

Creates a parallel scan iterator which iterates over all keys in the store in unsorted order. The result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of keys in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" keys are to be retrieved. May be NULL, or may have only a partial major key path.

return_iterator

The **return_parallel_scan_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_parallel_scan_iterator() (page 101).

sub_range

The **sub_range** parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which records are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_UNORDERED` is supported by this function.

batch_size

The **batch_size** parameter provides the suggested number of keys to fetch during each network round trip. If only the first or last key is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

store_iterator_config

The **store_iterator_config** parameter configures the parallel scan operation. Both the maximum number of requests and the maximum number of results batches can be specified using a kv_store_iterator_config_t (page 185) structure.

See Also

Data Operations and Related Functions (page 44)

kv_put()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_put(kv_store_t *store,
       const kv_key_t *key,
       const kv_value_t *value,
       kv_version_t **new_version)
```

Writes the key/value pair to the store, inserting or overwriting as appropriate.

This is the simplified version of the function that uses default values for most of put options. For a more complete version that lets you use non-default values, use kv_put_with_options() (page 98).

Parameters

store

The **store** parameter is the handle to the store where you want to write the key/value pair.

key

The **key** parameter is the key that you want to write to the store. It is created using kv_create_key() (page 124) or kv_create_key_from_uri() (page 128).

value

The **value** parameter is the value that you want to write to the store. It is created using kv_create_value() (page 132).

new_version

The **new_version** parameter references memory into which is copied the key/value pair's new version information. This pointer will be NULL if this function produces a non-zero return code.

You release the resources used by the version data structure using kv_release_version() (page 142).

See Also

Data Operations and Related Functions (page 44)

kv_put_with_options()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_put_with_options(kv_store_t *store,
                    const kv_key_t *key,
                    const kv_value_t *value,
                    const kv_version_t *if_version,
                    kv_presence_enum (page 180) if_presence,
                    kv_version_t **new_version,
                    kv_value_t **previous_value,
                    kv_return_value_version_enum (page 180) return_info,
                    kv_durability_t durability,
                    kv_timeout_t timeout_ms)
```

Writes the key/value pair to the store, inserting or overwriting as appropriate.

Parameters

store

The **store** parameter is the handle to the store where you want to write the key/value pair.

key

The **key** parameter is the key that you want to write to the store. It is created using kv_create_key() (page 124) or kv_create_key_from_uri() (page 128).

value

The **value** parameter is the value that you want to write to the store. It is created using kv_create_value() (page 132).

if_version

The **if_version** parameter indicates that the record should be put only if the existing value matches the version supplied to this parameter. Use this parameter when updating a value to ensure that it has not changed since it was last read. The version is obtained using the kv_get_version() (page 139) function.

if_presence

The **if_presence** parameter describes the conditions under which the record can be put, based on the presence or absence of the record in the store. For example, `KV_IF_PRESENT` means that the record can only be written to the store if a version of the record already exists there.

For a list of all the available presence options, see kv_presence_enum (page 180).

new_version

The **new_version** parameter references memory into which is copied the key/value pair's new version information. This pointer will be NULL if this function produces a non-zero return code, or if **return_info** is not `KV_RETURN_VALUE_ALL` or `KV_RETURN_VALUE_VERSION`.

You release the resources used by the version data structure using kv_release_version() (page 142).

previous_value

The **previous_value** parameter references memory into which is copied the previous value associated with the given key. Returns NULL if there was no previous value (the operation is inserting a new record, rather than updating an existing one); or if **return_info** is not `KV_RETURN_VALUE_ALL` or `KV_RETURN_VALUE_VALUE`.

You release the resources used by this parameter using kv_release_value() (page 141).

return_info

The **return_info** parameter indicates what version and value information should be returned as a part of this operation. See kv_return_value_version_enum (page 180) for a list of possible options.

durability

The **durability** parameter provides the durability guarantee to be used with this write operation. Durability guarantees are created using kv_create_durability() (page 145).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_release_iterator()

```
#include <kvstore.h>

void
kv_release_iterator(kv_iterator_t **iterator)
```

Releases the resources used by the iterator. Iterators are created using multiple-key operations, such as is performed using kv_multi_get_iterator() (page 78).

Parameters

iterator

The **iterator** parameter is the iterator that you want to release.

See Also

Data Operations and Related Functions (page 44)

kv_release_parallel_scan_iterator()

```
#include <kvstore.h>

void
kv_release_parallel_scan_iterator(
    kv_parallel_scan_iterator_t **iterator)
```

Releases the resources used by the iterator, which was created using kv_parallel_store_iterator() (page 93) or kv_parallel_store_iterator_keys() (page 95).

Parameters

iterator

The **iterator** parameter is the iterator that you want to release.

See Also

Data Operations and Related Functions (page 44)

kv_release_operation_results()

```
#include <kvstore.h>

void
kv_release_operation_results(kv_operation_results_t **results)
```

Releases the results list created by running kv_execute() (page 56).

Parameters

store

The **store** parameter is the handle to the store to which the results list belongs.

results

The **results** parameter is the results list that you want to release.

See Also

Data Operations and Related Functions (page 44)

kv_release_operations()

```
#include <kvstore.h>

void
kv_release_operations(kv_operations_t **operations)
```

Releases a multi-step, sequence of operations that was initially created using kv_create_operations() (page 49).

Parameters

operations

The **operations** parameter is the multi-step operation that you want to release.

See Also

Data Operations and Related Functions (page 44)

kv_result_get_previous_value()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_result_get_previous_value(const kv_operation_results_t *res,
                             kv_int_t index,
                             const kv_value_t **value)
```

Returns the previous value that existed before a put operation was run as a part of multi-step, sequence of operations. A previous value will only exist if the provided index in the operation contains a put operation as created by kv_create_put_with_options_op() (page 51), and if that function's **return_info** parameter is `KV_RETURN_VALUE_ALL` or `KV_RETURN_VALUE_VALUE`.

If the **index** parameter is out of range, this function returns `KV_NO_SUCH_OBJECT`.

Parameters

res

The **res** parameter is the operation results list that contains the previous value you want to examine.

index

The **index** parameter is the index in the results list which holds the information you want to retrieve.

value

The **value** parameter references memory to which is copied the previous value. Release the resources used by this value using kv_release_value() (page 141).

This parameter will be NULL if the indexed result was not generated by put operation that was configured to return previous values.

See Also

Data Operations and Related Functions (page 44)

kv_result_get_previous_version()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_result_get_previous_version(const kv_operation_results_t *res,
                               kv_int_t index,
                               const kv_version_t **version)
```

Returns the value's version that existed before a put operation was run as a part of multi-step, sequence of operations. A previous version will only exist if the provided index in the operation contains a put operation as created by kv_create_put_with_options_op() (page 51), and if that function's **return_info** parameter is `KV_RETURN_VALUE_ALL` or `KV_RETURN_VALUE_VERSION`.

If the **index** parameter is out of range, this function returns `KV_NO_SUCH_OBJECT`.

Parameters

res

The **res** parameter is the operation results list that contains the previous version you want to examine.

index

The **index** parameter is the index in the results list which holds the information you want to retrieve.

version

The **version** parameter references memory to which is copied the previous version. Release the resources used by this value using kv_release_version() (page 142).

This parameter will be NULL if the indexed result was not generated by put operation that was configured to return previous versions.

See Also

Data Operations and Related Functions (page 44)

kv_result_get_success()

```
#include <kvstore.h>

kv_int_t
kv_result_get_success(const kv_operation_results_t *res,
                      kv_int_t index)
```

Identifies whether the operation at the provided index was successful. Returns KV_TRUE if it was successful; KV_FALSE otherwise. If the **index** parameter is out of range, this function returns KV_NO_SUCH_OBJECT.

Parameters

res

The **res** parameter is the operation results list containing the operation result that you want to examine.

index

The **index** parameter is the index in the results list which hold the information you want to retrieve.

See Also

Data Operations and Related Functions (page 44)

kv_result_get_version()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_result_get_version(const kv_operation_results_t *res,
                      kv_int_t index,
                      const kv_version_t **version)
```

Returns the value's version. The version information is stored at the identified index in the results list as a consequence of successfully executing either kv_create_put_op() (page 50) or kv_create_put_with_options_op() (page 51). If the **index** parameter is out of range, this function returns KV_NO_SUCH_OBJECT.

Parameters

res

The **res** parameter is the operation results list that contains the version information you want to examine.

index

The **index** parameter is the index in the results list which holds the information you want to retrieve.

version

The **version** parameter references memory to which is copied the version information. Release the resources used by this value using kv_release_version() (page 142).

See Also

Data Operations and Related Functions (page 44)

kv_store_iterator()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_store_iterator(kv_store_t *store,
                  const kv_key_t *parent_key,
                  kv_iterator_t **return_iterator,
                  const kv_key_range_t *sub_range,
                  kv_depth_enum (page 179) depth,
                  kv_direction_enum (page 179) direction,
                  int batch_size,
                  kv_consistency_t *consistency,
                  kv_timeout_t timeout_ms)
```

Creates an iterator which iterates over all key/value pairs in unsorted order. The result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of key/value pairs in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

This function differs from kv_multi_get_iterator() (page 78) in that it allows the **parent_key** to be NULL, or to have only a partial major key path.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" records are to be retrieved.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

sub_range

The **sub_range** parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which records are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_UNORDERED` is supported by this function.

batch_size

The **batch_size** parameter provides the suggested number of records to fetch during each network round trip. If only the first or last key/value pair is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

kv_store_iterator_keys()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_store_iterator_keys(kv_store_t *store,
                       const kv_key_t *parent_key,
                       kv_iterator_t **return_iterator,
                       const kv_key_range_t *sub_range,
                       kv_depth_enum (page 179) depth,
                       kv_direction_enum (page 179) direction,
                       int batch_size,
                       kv_consistency_t *consistency,
                       kv_timeout_t timeout_ms)
```

Creates an iterator which iterates over all keys in the store in unsorted order. The result is not transactional and the operation effectively provides read-committed isolation. The implementation batches the fetching of keys in the iterator, to minimize the number of network round trips, while not monopolizing the available bandwidth.

This function differs from kv_multi_get_iterator_keys() (page 80) in that it allows the **parent_key** to be NULL, or to have only a partial major key path.

Parameters

store

The **store** parameter is the handle to the store for which you want an iterator.

parent_key

The **parent_key** parameter is the parent key whose "child" keys are to be retrieved.

return_iterator

The **return_iterator** parameter references memory into which is copied the results set. Release the resources used by this iterator using kv_release_iterator() (page 100).

sub_range

The **sub_range** parameter further restricts the range under the parent_key to the minor path components in this key range. It may be NULL.

You construct a key range using kv_init_key_range() (page 60).

depth

The **depth** parameter specifies how deep the retrieval can go. You can allow only children to be retrieved, the parent and all the children, all descendants, and so forth. See kv_depth_enum (page 179) for a description of all your depth options.

direction

The **direction** parameter specifies the order in which records are returned by the iterator. Only `kv_direction_enum.KV_DIRECTION_UNORDERED` is supported by this function.

batch_size

The **batch_size** parameter provides the suggested number of keys to fetch during each network round trip. If only the first or last key is desired, passing a value of one (1) is recommended. If zero, an internally determined default is used.

consistency

The **consistency** parameter identifies the consistency policy to use for this read operation. Consistency policies are created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

timeout_ms

The **timeout_ms** parameter provides an upper bound on the time interval for processing the operation. A best effort is made not to exceed the specified limit. If zero, the default request timeout is used. The default request timeout is set using kv_config_set_timeouts() (page 23).

See Also

Data Operations and Related Functions (page 44)

Chapter 4. Avro Functions

This chapter describes the functions used to manage the Avro data format. Avro provides the ability to describe the schema used for the value portion of a key-value pair. You describe this schema in a flat-text file using Javascript Object Notation (JSON). A trivial example of an Avro schema would be:

```
{
     "type": "record",
     "namespace": "com.example",
     "name": "FullName",
     "fields": [
       { "name": "first", "type": "string" },
       { "name": "last", "type": "string" }
     ]
}
```

Once you create your schema, you provide it to both the store and to your client code. You can then efficiently serialize and deserialize the data stored in the record's value.

For more information about Avro, see the *Oracle NoSQL Database Getting Started Guide*. For information on using the Avro C API, see http://avro.apache.org/docs/current/api/c/index.html.

Avro Management Functions

Avro Functions	Description
kv_avro_get_current_schemas()	Get all the schemas enabled in the store
kv_avro_release_schemas()	Release the schemas datastructure
kv_avro_get_schema()	Retrieve the schema associated with a value
kv_avro_generic_to_value()	Serialize using a generic Avro binding
kv_avro_generic_to_object()	Deserialize using a generic Avro binding
kv_avro_raw_to_value()	Serialize using a raw Avro binding
kv_avro_raw_to_bytes()	Deserialize using a raw Avro binding

kv_avro_get_current_schemas()

```
#include <kvstore.h>

kv_int_t
kv_avro_get_current_schemas(kv_store_t *store,
                            avro_schema_t **schemas);
```

Returns all schemas currently active in the store. The return value represents the number of schemas retrieved from the store. If the retrieval effort encounters an error, the return value is less than 0.

The schema array retrieved by this function must be released using kv_avro_release_schemas() (page 115). If you want the schema to outlast the release call, copy the object and then call `avro_schema_incref()` on it. In this case, you are responsible for decrementing the reference using `avro_schema_decref()`.

Parameters

store

The **store** parameter is the handle to the store from which you want to retrieve schemas.

schemas

The **schemas** parameter references memory into which a pointer to the allocated schema structure is copied.

See Also

Avro Management Functions (page 113)

kv_avro_release_schemas()

```
#include <kvstore.h>

void
kv_avro_release_schemas(kv_store_t *store,
                        avro_schema_t **schemas);
```

Releases memory allocated by kv_avro_get_current_schemas() (page 114).

Parameters

store

The **store** parameter is the handle to the store from which you retrieved the schemas.

schemas

The **schemas** parameter is the array of schemas that you want to release.

See Also

Avro Management Functions (page 113)

kv_avro_get_schema()

```
#include <kvstore.h>

kv_error_t
kv_avro_get_schema(kv_value_t *value,
                   avro_schema_t *schema);
```

Returns the schema used by the value returned from the store. You must call `avro_schema_decref()` on the returned schema when done.

In Avro terminology, the retrieved schema is the *writer schema* used to serialize the value prior to writing it to the store.

Parameters

value

The **value** parameter is the value from which you want to retrieve the schema.

schema

The **schema** parameter is the writer schema retrieved from the store. When you are done with this schema, call `avro_schema_decref()` on it.

See Also

Avro Management Functions (page 113)

kv_avro_generic_to_value()

```
#include <kvstore.h>

kv_error_t
kv_avro_generic_to_value(kv_store_t *store,
                         avro_value_t *avro_value,
                         kv_value_t **value);
```

Converts (serializes) an generic Avro value to a `kv_value_t` so that the data can be written to the store. The Avro schema associated with the Avro value is used to perform the serialization.

Parameters

store

The **store** parameter is the handle to the store where the data will be written.

avro_value

The **avro_value** parameter is the Avro value that you want to serialize. Once you are done with the serialization, you are still responsible for ensuring that the `avro_value_t` is freed (by decrement the structure's reference count to 0).

value

The **value** parameter references memory into which a pointer to the allocated `kv_value_t` structure is copied. When you are done with this structure, it is your responsibility to free it using kv_release_value() (page 141).

See Also

Avro Management Functions (page 113)

kv_avro_generic_to_object()

```
#include <kvstore.h>

kv_error_t
kv_avro_generic_to_object(kv_value_t *value,
                          avro_value_t *avro_value,
                          avro_schema_t reader_schema);
```

Converts (deserializes) a `kv_value_t` to an Avro value.

For deserialization to succeed, the **reader_schema** (which is optional) need not be loaded into the store. It is only an error to attempt to put an Avro value with a schema that is not loaded into the store.

This function optionally supports schema evolution by resolving the **reader_schema** with the **value**'s writer schema. If the two are identical, there are no issues. If they are not identical, but they are compatible, the result will be created based on Avro's resolution rules. If they are not compatible, the error `KV_AVRO` is returned, and additional information may be available using kv_get_last_error() (page 177).

Note that the The Avro C API does not currently support default values. This means that if the **reader_schema** adds fields relative to the existing value, those fields will be zero. Further, unless Avro itself was built such that it will allow missing fields in the resolved writer schema (see the build instructions), the operation will fail entirely because of mismatched schema.

Parameters

value

The **value** parameter is Oracle NoSQL Database value that you want to deserialize into a generic Avro value.

avro_value

The **avro_value** parameter is a pointer to the destination object for the deserialized data.

reader_schema

The **reader_schema** parameter is optional. It provides the reader schema used for schema evolution.

See Also

Avro Management Functions (page 113)

kv_avro_raw_to_value()

```
#include <kvstore.h>

kv_error_t
kv_avro_raw_to_value(kv_store_t *store,
                     const char *bytes,
                     uint64_t len,
                     const avro_schema_t schema,
                     kv_value_t **value);
```

Converts (serialize) a raw byte array to a `kv_value_t`. The schema argument is associated with the value, but no attempt is made to compare the schema to the contents of the array. If the schema does not exist in the store, an error is returned.

Parameters

store

The **store** parameter is a handle to the store where you want to write the value.

bytes

The **bytes** parameter is the raw byte array that you want to convert to an `kv_value_t`.

len

The **len** parameter is is the size of the byte array.

schema

The **schema** parameter is the Avro schema associated with the serialized value.

value

The **value** parameter references memory into which a pointer to the allocated `kv_value_t` structure is copied. When you are done with this structure, it is your responsibility to free it using kv_release_value() (page 141).

See Also

Avro Management Functions (page 113)

kv_avro_raw_to_bytes()

```
#include <kvstore.h>

kv_error_t
kv_avro_raw_to_bytes(const kv_value_t *value,
                     char *bytes,
                     uint64_t len);
```

Converts a `kv_value_t` to a raw byte array, which are assumed to be serialized Avro content. You must allocate the buffer. The buffer size can be determined using kv_get_value_size() (page 138), which will return a value that is slightly larger than required. The Avro schema associated with the value can be obtained using kv_avro_get_schema() (page 116).

Parameters

value

The **value** parameter is the `kv_value_t` that you want to convert to a raw byte array.

bytes

The **bytes** parameter is buffer where you want the converted bytes to be placed.

len

The **len** parameter is is the size of the buffer. You can discover the required buffer size using kv_get_value_size() (page 138).

See Also

Avro Management Functions (page 113)

Chapter 5. Key/Value Pair Management Functions

This chapter describes the functions used to manage keys and values. Both are used to describe a single entry (or record) in the KV Store. In addition, this chapter describes functions used to manage versions; that is, data structures that identify the key-value pair's specific version.

Key/Value Pair Management Functions

Key Functions	Description
kv_create_key()	Allocate and initialize a key structure using major and minor path components
kv_create_key_copy()	Allocate and initialize a key structure using major and minor path components
kv_create_key_from_uri()	Allocate and initialize a key structure using a URI string
kv_create_key_from_uri_copy()	Allocate and initialize a key structure using a URI string
kv_get_key_major()	Return the major path components for the key
kv_get_key_minor()	Return the minor path components for the key
kv_get_key_uri()	Return the key's major and minor path components as a URI
kv_release_key()	Release the key structure, freeing all associated memory
Value Functions	
kv_create_value()	Allocate and initialize a value structure
kv_create_value_copy()	Allocate and initialize a value structure
kv_get_value()	Returns the value as a string
kv_get_value_size()	Returns the value's size
kv_release_value()	Release a value structure
Version Functions	
kv_copy_version()	Copies a version structure
kv_get_version()	Returns a value's version
kv_release_version()	Release a version structure

kv_copy_version()

```
#include <kvstore.h>

kv_error_t
kv_copy_version(const kv_version_t *from, kv_version_t **to)
```

Copies a version structure.

Parameters

from

The **from** parameter is the version structure you want to copy. Normally, these are created using kv_get_version() (page 139).

from

The **from** parameter references memory into which a pointer to the allocated version structure is copied. Release the resources used by this structure using kv_release_version() (page 142).

See Also

Key/Value Pair Management Functions (page 122)

kv_create_key()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_key(kv_store_t *store,
              kv_key_t **key,
              const char **major,
              const char **minor)
```

Creates a key in the key/value store. To release the resources used by this structure, use kv_release_key() (page 140).

This function differs from kv_create_key_copy() (page 126) in that it does not copy the contents of the strings passed to the function. Therefore, these strings should not be released or modified until the kv_key_t structure created by this function is released.

A key represents a path to a value in a hierarchical namespace. It consists of a sequence of string path component names, and each component name is used to navigate the next level down in the hierarchical namespace. The complete sequence of string components is called the *full key path*.

The sequence of string components in a full key path is divided into two groups or sub-sequences: The major key path is the initial or beginning sequence, and the minor key path is the remaining or ending sequence. The Full Path is the concatenation of the Major and Minor Paths, in that order. The Major Path must have at least one component, while the Minor path may be empty (have zero components).

Each path component must be a non-null String. Empty (zero length) Strings are allowed, except that the first component of the major path must be a non-empty String.

Given a key, finding the location of a key/value pair is a two step process:

1. The major path is used to locate the node on which the key/value pair can be found.

2. The full path is then used to locate the key/value pair within that node.

Therefore all key/value pairs with the same major path are clustered on the same node.

Keys which share a common major path are physically clustered by the KVStore and can be accessed efficiently via special multiple-operation APIs, e.g. kv_multi_get() (page 76). The APIs are efficient in two ways:

1. They permit the application to perform multiple-operations in single network round trip.

2. The individual operations (within a multiple-operation) are efficient since the common-prefix keys and their associated values are themselves physically clustered.

Multiple-operation APIs also support ACID transaction semantics. All the operations within a multiple-operation are executed within the scope of a single transaction.

Parameters

store

The **store** parameter is the handle to the store in which the key is used. The store handle is obtained using kv_open_store() (page 31).

key

The **key** parameter references memory into which a pointer to the allocated key is copied.

major

The **major** parameter is an array of strings, each element of which represents a major path component.

Note that the string used here is *not* copied. You must not release or modify this memory until the structure in which it is used is released.

minor

The **minor** parameter is an array of strings, each element of which represents a minor path component.

Note that the string used here is *not* copied. You must not release or modify this memory until the structure in which it is used is released.

See Also

Key/Value Pair Management Functions (page 122)

kv_create_key_copy()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_key_copy(kv_store_t *store,
                   kv_key_t **key,
                   const char **major,
                   const char **minor)
```

Creates a key in the key/value store. To release the resources used by this structure, use kv_release_key() (page 140).

This function differs from kv_create_key() (page 124) in that it copies the contents of the strings passed to the function, so that those strings can be released, or modified and then reused in whatever way is required by the application.

A key represents a path to a value in a hierarchical namespace. It consists of a sequence of string path component names, and each component name is used to navigate the next level down in the hierarchical namespace. The complete sequence of string components is called the *full key path*.

The sequence of string components in a full key path is divided into two groups or sub-sequences: The major key path is the initial or beginning sequence, and the minor key path is the remaining or ending sequence. The Full Path is the concatenation of the Major and Minor Paths, in that order. The Major Path must have at least one component, while the Minor path may be empty (have zero components).

Each path component must be a non-null String. Empty (zero length) Strings are allowed, except that the first component of the major path must be a non-empty String.

Given a key, finding the location of a key/value pair is a two step process:

1. The major path is used to locate the node on which the key/value pair can be found.

2. The full path is then used to locate the key/value pair within that node.

Therefore all key/value pairs with the same major path are clustered on the same node.

Keys which share a common major path are physically clustered by the KVStore and can be accessed efficiently via special multiple-operation APIs, e.g. kv_multi_get() (page 76). The APIs are efficient in two ways:

1. They permit the application to perform multiple-operations in single network round trip.

2. The individual operations (within a multiple-operation) are efficient since the common-prefix keys and their associated values are themselves physically clustered.

Multiple-operation APIs also support ACID transaction semantics. All the operations within a multiple-operation are executed within the scope of a single transaction.

Parameters

store

The **store** parameter is the handle to the store in which the key is used. The store handle is obtained using kv_open_store() (page 31).

key

The **key** parameter references memory into which a pointer to the allocated key is copied.

major

The **major** parameter is an array of strings, each element of which represents a major path component.

Note that the string used here *is* copied. You may release or modify this memory as needed because the contents of this memory is copied to memory owned by the `kv_key_t` structure.

minor

The **minor** parameter is an array of strings, each element of which represents a minor path component.

Note that the string used here *is* copied. You may release or modify this memory as needed because the contents of this memory is copied to memory owned by the `kv_key_t` structure.

See Also

Key/Value Pair Management Functions (page 122)

kv_create_key_from_uri()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_key_from_uri(kv_store_t *store,
                       kv_key_t **key,
                       const char *uri)
```

Creates a key in the key/value store based on a URI. To release the resources used by this structure, use kv_release_key() (page 140).

This function differs from kv_create_key_from_uri_copy() (page 130) in that it does not copy the contents of the URI string passed to the function. Therefore, the URI strings should not be released or modified until the `kv_key_t` structure created by this function is released.

The key path string format used here is designed to work with URIs and URLs. It is intended to be used as a general purpose string identifier. The key path components are separated by slash (/) delimiters. A special slash-hyphen-slash delimiter (`/-/`) is used to separate the major and minor paths. Characters that are not allowed in a URI path are encoded using URI syntax (%XX where XX are hexadecimal digits). The string always begins with a leading slash to prevent it from begin treated as a URI relative path. Some examples are below.

- `/SingleComponentMajorPath`
- `/MajorPathPart1/MajorPathPart2/-/MinorPathPart1/MinorPathPart2`
- `/HasEncodedSlash:%2F,Zero:%00,AndSpace:%20`

Example 1 demonstrates the simplest possible path. Note that a leading slash is always necessary.

Example 2 demonstrates the use of the `/-/` separator between the major and minor paths. If a key happens to have a path component that is nothing but a hyphen, to distinguish it from that delimiter it is encoded as `%2D`. For example: `/major/%2d/path/-/minor/%2d/path`.

Example 3 demonstrates encoding of characters that are not allowed in a path component. For URI compatibility, characters that are encoded are the ASCII space and other Unicode separators, the ASCII and Unicode control characters, and the following 15 ASCII characters: (" # % / < > ? [\ ] ^ ` { | }). The hyphen (-) is also encoded when it is the only character in the path component, as described above.

Note that although any Unicode character may be used in a key path component, in practice it may be problematic to include control characters because web user agents, proxies, and so forth, may not be tolerant of all characters. Although it will be encoded, embedding a slash in a path component may also be problematic. It is the responsibility of the application to use characters that are compatible with other software that processes the URI.

Parameters

store

The **store** parameter is the handle to the store in which the key is used. The store handle is obtained using kv_open_store() (page 31).

key

The **key** parameter references memory into which a pointer to the allocated key is copied.

uri

The **uri** parameter is the full key path, both major and minor components, described as a string. See the description at the beginning of this page for how that string should be formatted.

Note that the string used here is *not* copied. You must not release or modify this memory until the structure in which it is used is released.

See Also

Key/Value Pair Management Functions (page 122)

kv_create_key_from_uri_copy()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_key_from_uri_copy(kv_store_t *store,
                            kv_key_t **key,
                            const char *uri)
```

Creates a key in the key/value store based on a URI. To release the resources used by this structure, use kv_release_key() (page 140).

This function differs from kv_create_key_from_uri() (page 128) in that it copies the contents of the URI string passed to the function, so that the string can be released, or modified and then reused in whatever way is required by the application.

The key path string format used here is designed to work with URIs and URLs. It is intended to be used as a general purpose string identifier. The key path components are separated by slash (/) delimiters. A special slash-hyphen-slash delimiter (/-/) is used to separate the major and minor paths. Characters that are not allowed in a URI path are encoded using URI syntax (%XX where XX are hexadecimal digits). The string always begins with a leading slash to prevent it from begin treated as a URI relative path. Some examples are below.

- `/SingleComponentMajorPath`
- `/MajorPathPart1/MajorPathPart2/-/MinorPathPart1/MinorPathPart2`
- `/HasEncodedSlash:%2F,Zero:%00,AndSpace:%20`

Example 1 demonstrates the simplest possible path. Note that a leading slash is always necessary.

Example 2 demonstrates the use of the `/-/` separator between the major and minor paths. If a key happens to have a path component that is nothing but a hyphen, to distinguish it from that delimiter it is encoded as `%2D`. For example: `/major/%2d/path/-/minor/%2d/path`.

Example 3 demonstrates encoding of characters that are not allowed in a path component. For URI compatibility, characters that are encoded are the ASCII space and other Unicode separators, the ASCII and Unicode control characters, and the following 15 ASCII characters: (" # % / < > ? [\ ] ^ ` { | }). The hyphen (-) is also encoded when it is the only character in the path component, as described above.

Note that although any Unicode character may be used in a key path component, in practice it may be problematic to include control characters because web user agents, proxies, and so forth, may not be tolerant of all characters. Although it will be encoded, embedding a slash in a path component may also be problematic. It is the responsibility of the application to use characters that are compatible with other software that processes the URI.

Parameters

store

The **store** parameter is the handle to the store in which the key is used. The store handle is obtained using kv_open_store() (page 31).

key

The **key** parameter references memory into which a pointer to the allocated key is copied.

uri

The **uri** parameter is the full key path, both major and minor components, described as a string. See the description at the beginning of this page for how that string should be formatted.

Note that the string used here *is* copied. You may release or modify this memory as needed because the contents of this memory is copied to memory owned by the `kv_key_t` structure.

See Also

Key/Value Pair Management Functions (page 122)

kv_create_value()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_value(kv_store_t *store,
                kv_value_t **value,
                const unsigned char *data,
                int data_len)
```

Creates the value in a key/value store. To release the resources used by this structure, use kv_release_value() (page 141).

This function differs from kv_create_value_copy() (page 133) in that it does not copy the contents of the data buffer passed to the function. Therefore, the data buffer should not be released or modified until the `kv_value_t` structure created by this function is released.

Parameters

store

The **store** parameter is the handle to the store in which the value is stored. The store handle is obtained using kv_open_store() (page 31).

value

The **value** parameter references memory into which a pointer to the allocated value is copied.

data

The **data** parameter is a buffer containing the data to be contained in the value.

Note that the buffer used here is *not* copied. You must not release or modify this memory until the structure in which it is used is released.

data_len

The **data_len** parameter indicates the size of the `data` buffer.

See Also

Key/Value Pair Management Functions (page 122)

kv_create_value_copy()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_value_copy(kv_store_t *store,
                     kv_value_t **value,
                     const unsigned char *data,
                     int data_len)
```

Creates the value in a key/value store. To release the resources used by this structure, use kv_release_value() (page 141).

This function differs from kv_create_value() (page 132) in that it copies the contents of the data buffer passed to the function, so that the buffer can be released, or modified and then reused in whatever way is required by the application.

Parameters

store

The **store** parameter is the handle to the store in which the value is stored. The store handle is obtained using kv_open_store() (page 31).

value

The **value** parameter references memory into which a pointer to the allocated value is copied.

data

The **data** parameter is a buffer containing the data to be contained in the value.

Note that the buffer used here *is* copied. You may release or modify this memory as needed because the contents of this memory is copied to memory owned by the `kv_value_t` structure.

data_len

The **data_len** parameter indicates the size of the `data` buffer.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_key_major()

```
#include <kvstore.h>

const char **
kv_get_key_major(const kv_key_t *key)
```

Returns the major path components used by the provided key.

Note that the string returned by this function is owned by the key structure, and is valid until the key is released.

Parameters

key

The **key** parameter is the key structure from which you want to obtain the major path components.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_key_minor()

```
#include <kvstore.h>

const char **
kv_get_key_minor(const kv_key_t *key)
```

Returns the minor path components used by the provided key.

Note that the string returned by this function is owned by the key structure, and is valid until the key is released.

Parameters

key

The **key** parameter is the key structure from which you want to obtain the minor path components.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_key_uri()

```
#include <kvstore.h>

const char *
kv_get_key_uri(const kv_key_t *key)
```

Returns a string representing the key's major and minor path components. See kv_create_key_from_uri() (page 128) for a description of the string's syntax.

Note that the string returned by this function is owned by the key structure, and is valid until the key is released.

Parameters

key

The **key** parameter is the key structure from which you want to obtain the path URI.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_value()

```
#include <kvstore.h>

const unsigned char *
kv_get_value(const kv_value_t *value)
```

Returns the value as a string.

Note that the string returned by this function is owned by the value structure, and is valid until the value is released.

Parameters

value

The **value** parameter is the value structure from which you want to extract its contents as a string.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_value_size()

```
#include <kvstore.h>

kv_int_t
kv_get_value_size(const kv_value_t *value)
```

Returns the size of the value, in bytes.

Parameters

value

The **value** parameter is the value structure for which you want its size.

See Also

Key/Value Pair Management Functions (page 122)

kv_get_version()

```
#include <kvstore.h>

const kv_version_t *
kv_get_version(const kv_value_t *value)
```

Creates a version structure, which refers to a specific version of a key-value pair. Note that the `kv_version_t` structure returned by this function is owned by the `kv_value_t` structure from which is was obtained. As such, you should not explicitly release the version structure returned by this function; it will be automatically released when the value structure is released.

When a key-value pair is initially inserted in the KV Store, and each time it is updated, it is assigned a unique version token. The version is associated with the version portion of the key-value pair. The version is important for two reasons:

- When an update or delete is to be performed, it may be important to only perform the update or delete if the last known value has not changed. For example, if an integer field in a previously known value is to be incremented, it is important that the previous value has not changed in the KV Store since it was obtained by the client. This can be guaranteed by passing the version of the previously known value to the `if_version` parameter of the kv_put_with_options() (page 98) or kv_delete_with_options() (page 54) functions. If the version specified does not match the current version of the value in the KV Store, these functions will not perform the update or delete operation and will return an indication of failure. Optionally, they will also return the current version and/or value so the client can retry the operation or take a different action.
- When a client reads a value that was previously written, it may be important to ensure that the KV Store node servicing the read operation has been updated with the information previously written. This can be accomplished by using a version-based consistency policy with the read operation. See kv_create_version_consistency() (page 149) for more information.

Be aware that the system may infrequently assign a new version to a key-value pair; for example, when migrating data for better resource usage. Therefore, when using kv_put_with_options() (page 98) or kv_delete_with_options() (page 54), do not assume that the version will remain constant until it is changed by the application.

Parameters

value

The **value** parameter is the value structure from which you want to extract version information.

See Also

Key/Value Pair Management Functions (page 122)

kv_release_key()

```
#include <kvstore.h>

void
kv_release_key(kv_key_t **key)
```

Releases the resources used by a key. The structure was intially allocated using kv_create_key() (page 124) or kv_create_key_from_uri() (page 128).

Parameters

key

The **key** parameter references the kv_key_t structure that you want to release.

See Also

Key/Value Pair Management Functions (page 122)

kv_release_value()

```
#include <kvstore.h>

void
kv_release_value(kv_value_t **value)
```

Releases the resources used by a value. The value was initially created using kv_create_value() (page 132), or it may have been created as a return value from some data operation function.

Parameters

value

The **value** parameter is the `kv_value_t` structure that you want to release.

See Also

Key/Value Pair Management Functions (page 122)

kv_release_version()

```
#include <kvstore.h>

void
kv_release_version(kv_version_t **version)
```

Releases a version structure. The version structure was initially created using kv_get_version() (page 139), or through some store write operation such as is performed by kv_put_with_options() (page 98).

Parameters

version

The **value** parameter is the `kv_version_t` structure that you want to release.

See Also

Key/Value Pair Management Functions (page 122)

Chapter 6. Durability and Consistency Functions

This chapter describes the functions used to manage durability and consistency policies. Durability policies are used with write operations to manage how likely your data writes are to persist in the event of a catastrophic failure, be it in your hardware or software layers. By default, your writes are highly durable. So managing durability policies is mostly about relaxing your durability guarantees in an effort to improve your write throughput.

Consistency policies are used with read operations to describe how likely it is that the data on your replicas will be identical to, or *consistent with*, the data on your master server. The most stringent consistency policy requires that the read operation be performed on the master server. In general, the stricter your consistency policy, the slower your store's read throughput.

Durability and Consistency Management Functions

Consistency Functions	Description
kv_create_simple_consistency()	Create and initialize a Consistency structure
kv_create_time_consistency()	Create and initialize a Consistency structure using time information
kv_create_version_consistency()	Create and initialize a Consistency structure using a Version
kv_get_consistency_type()	Return the Consistency type
kv_release_consistency()	Release the Consistency structure
Durability Functions	
kv_create_durability()	Allocate and initialize a Durability structure
kv_get_default_durability()	Return the store's default Durability
kv_get_durability_master_sync()	Return the transaction synchronization policy used on the Master
kv_get_durability_replica_ack()	Return the replica's acknowledgement policy
kv_get_durability_replica_sync()	Return the transaction synchronization policy used on the replica
kv_is_default_durability()	Return whether the durability is the store's default

kv_create_durability()

```
#include <kvstore.h>

kv_durability_t
kv_create_durability(kv_sync_policy_enum (page 183) master,
                     kv_sync_policy_enum (page 183) replica,
                     kv_ack_policy_enum (page 182) ack)
```

Creates a durability policy, which is then used for store write operations such as kv_put_with_options() (page 98) or kv_delete_with_options() (page 54). The durability policy can also be used with a set of operations performed in a single transaction, using kv_execute() (page 56).

The overall durability is a function of the sync policy in effect for the master, the sync policy in effect for each replica, and the replication acknowledgement policy in effect for the replication group.

Parameters

master

The **master** parameter defines the synchronization policy in effect for the master in this replication group for this durability guarantee. See kv_sync_policy_enum (page 183) for a list of the synchronization policies that you can set.

replica

The **replica** parameter defines the synchronization policy in effect for the replicas in this replication group for this durability guarantee. See kv_sync_policy_enum (page 183) for a list of the synchronization policies that you can set.

ack

The **ack** parameter defines the acknowledgement policy to be used for this durability guarantee. The acknowledgment policy describes how many replicas must respond to, or *acknowledge* a transaction commit before the master considers the transaction completed. See kv_ack_policy_enum (page 182) for a list of the possible acknowledgement policies.

See Also

Durability and Consistency Management Functions (page 144)

kv_create_simple_consistency()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_simple_consistency(kv_consistency_t **consistency,
                             kv_consistency_enum (page 182) type)
```

Creates a simple consistency guarantee used for read operations.

In general, read operations may be serviced either at a master or replica node. When reads are serviced at the master node, consistency is always absolute. For reads that might be performed at a replica, you can specify ABSOLUTE consistency to force the operation to be serviced at the master. For other types of consistency, when the operation is serviced at a replica, the read transaction will not begin until the consistency policy is satisfied.

Consistency policies can be used for read operation performed in the store, such as with kv_get_with_options() (page 59) or kv_store_iterator() (page 108).

You release the memory allocated for the consistency structure using kv_release_consistency() (page 157).

Parameters

consistency

The **consistency** parameter references memory into which a pointer to the allocated consistency policy is copied.

type

The **type** parameter defines the type of consistency you want to use. See kv_consistency_enum (page 182) for a list of the simple consistency policies that you can specify.

See Also

Durability and Consistency Management Functions (page 144)

kv_create_time_consistency()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_time_consistency(kv_consistency_t **consistency,
                           kv_timeout_t time_lag,
                           kv_timeout_t timeout_ms)
```

Creates a consistency policy which describes the amount of time the replica is allowed to lag the master. The application can use this policy to ensure that the replica node sees all transactions that were committed on the master before the lag interval.

You release the memory allocated for the consistency structure using kv_release_consistency() (page 157).

Effective use of this policy requires that the clocks on the master and replica are synchronized by using a protocol like NTP.

Parameters

consistency

The **consistency** parameter references memory into which a pointer to the allocated consistency policy is copied.

time_lag

The **time_lag** parameter specifies the time interval, in milliseconds, by which the replica may be out of date with respect to the master when a transaction is initiated on the replica.

timeout_ms

The **timeout_ms** parameter describes how long a replica may wait for the desired consistency to be achieved before giving up.

To satisfied the consistency policy, the KVStore client driver implements a read operation by choosing a node (usually a replica) from the proper replication group, and sending it a request. If the replica cannot guarantee the desired Consistency within the Consistency timeout, it replies to the request with a failure indication. If there is still time remaining within the operation timeout, the client driver picks another node and tries the request again (transparent to the application).

KVStore operations which accept a consistency policy also accept a separate operation timeout. It makes sense to think of the operation timeout as the maximum amount of time the application is willing to wait for the operation to complete. On the other hand, the consistency timeout is like a performance hint to the implementation, suggesting that it can generally expect a healthy replica to become consistent within the given amount of time, and that if it does not, then it is probably more likely worth the overhead of abandoning the request attempt and retrying with a different replica. Note that for the consistency timeout to be meaningful, it must be smaller than the operation timeout.

Choosing a value for the operation timeout depends on the needs of the application. Finding a good consistency timeout value is more likely to depend on observations made of real system performance.

See Also

Durability and Consistency Management Functions (page 144)

kv_create_version_consistency()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_create_version_consistency(kv_consistency_t **consistency,
                              const kv_version_t *version,
                              kv_timeout_t timeout_ms)
```

Creates a consistency policy which ensures that the environment on a replica node is at least as current as denoted by the specified `version`. The `version` is created by providing a `Value` portion of a Key/Value pair to kv_get_version() (page 139), or is obtained from the result set provided to kv_result_get_version() (page 107) or kv_result_get_previous_version() (page 105). Versions are also returned in the `new_version` parameter of the kv_put_with_options() (page 98) function.

The version of a Key-Value pair represents a point in the serialized transaction schedule created by the master. In other words, the `version` is like a bookmark, representing a particular transaction commit in the replication stream. The replica ensures that the commit identified by the `version` has been executed before allowing the transaction on the replica to proceed.

For example, suppose the application is a web application. Each request to the web server consists of an update operation followed by read operations (say from the same client). The read operations naturally expect to see the data from the updates executed by the same request. However, the read operations might have been routed to a replica node that did not execute the update.

In such a case, the update request would generate a `version`, which would be resubmitted by the browser, and then passed with subsequent read requests to the KV Store. The read request may be directed by the KV Store's load balancer to any one of the available replicas. If the replica servicing the request is already current (with regards to the version token), it will immediately execute the transaction and satisfy the request. If not, the transaction will stall until the replica replay has caught up and the change is available at that node.

You release the memory allocated for the consistency structure using kv_release_consistency() (page 157).

Parameters

consistency

The **consistency** parameter references memory into which a pointer to the allocated consistency policy is copied.

version

The **version** parameter identifies the version that must be seen at the replica in order to consider it current. This value is created by providing a `Value` portion of a Key/Value pair to kv_get_version() (page 139), or is obtained from the result set provided to kv_result_get_version() (page 107) or kv_result_get_previous_version() (page 105).

timeout_ms

The **timeout_ms** parameter describes how long a replica may wait for the desired consistency to be achieved before giving up.

To satisfied the consistency policy, the KVStore client driver implements a read operation by choosing a node (usually a replica) from the proper replication group, and sending it a request. If the replica cannot guarantee the desired Consistency within the Consistency timeout, it replies to the request with a failure indication. If there is still time remaining within the operation timeout, the client driver picks another node and tries the request again (transparent to the application).

KVStore operations which accept a consistency policy also accept a separate operation timeout. It makes sense to think of the operation timeout as the maximum amount of time the application is willing to wait for the operation to complete. On the other hand, the consistency timeout is like a performance hint to the implementation, suggesting that it can generally expect a healthy replica to become consistent within the given amount of time, and that if it does not, then it is probably more likely worth the overhead of abandoning the request attempt and retrying with a different replica. Note that for the consistency timeout to be meaningful, it must be smaller than the operation timeout.

Choosing a value for the operation timeout depends on the needs of the application. Finding a good consistency timeout value is more likely to depend on observations made of real system performance.

See Also

Durability and Consistency Management Functions (page 144)

kv_get_consistency_type()

```
#include <kvstore.h>

kv_consistency_enum (page 182)
kv_get_consistency_type(kv_consistency_t *consistency)
```

Identifies the consistency policy type used by the provided policy. See kv_consistency_enum (page 182) for a list of possible consistency policy types.

Parameters

consistency

The **consistency** parameter points to the consistency policy for which you want to identify the type.

See Also

Durability and Consistency Management Functions (page 144)

kv_get_default_durability()

```
#include <kvstore.h>

kv_durability_t
kv_get_default_durability()
```

Returns the default durability policy in use by the KV Store. You set the default durability policy using kv_config_set_durability() (page 16).

See Also

Durability and Consistency Management Functions (page 144)

kv_get_durability_master_sync()

```
#include <kvstore.h>

kv_sync_policy_enum (page 183)
kv_get_durability_master_sync(kv_durability_t durability)
```

Returns the sync policy in use by the Master for the given durability policy. See kv_sync_policy_enum (page 183) for a list of the possible sync policies.

Parameters

durability

The **durability** parameter identifies the durability policy to be examined.

See Also

Durability and Consistency Management Functions (page 144)

kv_get_durability_replica_ack()

```
#include <kvstore.h>

kv_ack_policy_enum (page 182)
kv_get_durability_replica_ack(kv_durability_t durability)
```

Returns the acknowledgement policy in use for the given durability policy. The acknowledgement policy identifies how many replicas must acknowledge a transaction commit before the master considers the transaction to be completed. See kv_ack_policy_enum (page 182) for a list of the possible sync policies.

Parameters

durability

The **durability** parameter identifies the durability policy to be examined.

See Also

Durability and Consistency Management Functions (page 144)

kv_get_durability_replica_sync()

```
#include <kvstore.h>

kv_sync_policy_enum (page 183)
kv_get_durability_replica_sync(kv_durability_t durability)
```

Returns the sync policy in use by the replicas for the given durability policy. See kv_sync_policy_enum (page 183) for a list of the possible sync policies.

Parameters

durability

The **durability** parameter identifies the durability policy to be examined.

See Also

Durability and Consistency Management Functions (page 144)

kv_is_default_durability()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_is_default_durability(kv_durability_t durability)
```

Returns whether the identified durability policy is identical to the default durability policy. KV_SUCCESS indicates that the provided durability is equal to the default durability.

You set the default durability using kv_config_set_durability() (page 16).

Parameters

durability

The **durability** parameter identifies the durability policy to be examined.

See Also

Durability and Consistency Management Functions (page 144)

kv_release_consistency()

```
#include <kvstore.h>

void
kv_release_consistency(kv_consistency_t **consistency)
```

Releases (or frees) the memory allocated for the `kv_consistency_t` structure. This structure is initially created using kv_create_simple_consistency() (page 146), kv_create_time_consistency() (page 147), or kv_create_version_consistency() (page 149).

Parameters

consistency

The **consistency** structure to release.

See Also

Durability and Consistency Management Functions (page 144)

Chapter 7. Statistics Functions

This chapter describes functions used to retrieve and examine statistical information. There are two types of statistics described here: statistics related to store operations and state, and statistics related to parallel scan operations.

For store-related statistics, metrics can be obtained on a per-node or per-operation basis. All statistical information is contained within a structure that you allocate using kv_get_stats() (page 169). You release the resources allocated for this structure using kv_release_stats() (page 173).

In many cases, statistical information is reported for a time interval. For example, this happens when information is retrieved that reports on maximum, minimum and average numbers. In this case, the reporting interval can be restarted by calling kv_get_stats() (page 169) with a value of 1 for the **clear** parameter.

For parallel scan statistics, the statistics are retrieved from a parallel scan iterator using either kv_parallel_scan_get_partition_metrics() (page 170) or kv_parallel_scan_get_shard_metrics() (page 171). This returns a structure that you can examine using a number of different functions. You release this structure using kv_release_detailed_metrics_list() (page 172).

Statistics and Related Functions

Statistics Functions	Description
kv_get_node_metrics()	Returns metrics associated with each node in the KV Store
kv_get_num_nodes()	Return the number of nodes contained in the KV Store
kv_get_num_operations()	Return the number of operations that were executed
kv_get_operation_metrics()	Aggregates the metrics associated with a KV Store operation
kv_get_stats()	Return statistics associated with the KV Store
kv_release_stats()	Release the statistics structure
kv_stats_string()	Returns a descriptive string containing metrics for each operation
Parallel Scan Statistics	
kv_detailed_metrics_list_size()	Returns the size of the detailed metrics list
kv_detailed_metrics_list_get_record_count()	Returns the number of records in the detailed metrics list
kv_detailed_metrics_list_get_scan_time()	Returns the time used to perform the parallel scan
kv_detailed_metrics_list_get_name()	Returns the name of the shard or partition used by the parallel scan
kv_parallel_scan_get_partition_metrics()	Returns parallel scan metrics for the partition
kv_parallel_scan_get_shard_metrics()	Returns parallel scan metrics for the partition
kv_release_detailed_metrics_list()	Releases the detailed metrics list

kv_detailed_metrics_list_size()

```
#include <kvstore.h>

kv_int_t
kv_detailed_metrics_list_size(
    const kv_detailed_metrics_list_t *result)
```

Returns the size of the detailed metrics list, as created by kv_parallel_scan_get_partition_metrics() (page 170) and kv_parallel_scan_get_shard_metrics() (page 171).

Parameters

result

The **result** parameter is the detailed metrics list for which you want size information.

See Also

Statistics and Related Functions (page 159)

kv_detailed_metrics_list_get_record_count()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_detailed_metrics_list_get_record_count(
    const kv_detailed_metrics_list_t *res,
    kv_int_t index,
    kv_long_t *count)
```

Returns the record count for the shard or partition.

Parameters

res

The **res** parameter is the detailed metrics list for which you want the record count. It is created using either kv_parallel_scan_get_partition_metrics() (page 170) or kv_parallel_scan_get_shard_metrics() (page 171).

index

The **index** parameter is the point in the scan to which you want to examine the record count.

count

The **count** parameter references memory into which is placed the record count up to the point in the scan identified by **index**.

See Also

Statistics and Related Functions (page 159)

kv_detailed_metrics_list_get_scan_time()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_detailed_metrics_list_get_scan_time(
    const kv_detailed_metrics_list_t *res,
    kv_int_t index,
    kv_long_t *time)
```

Returns the time in milliseconds used to scan the partition or shard.

Parameters

res

The **res** parameter is the detailed metrics list for which you want the scan time. It is created using either kv_parallel_scan_get_partition_metrics() (page 170) or kv_parallel_scan_get_shard_metrics() (page 171).

index

The **index** parameter is the point in the scan up to which you want to examine the scan time.

time

The **time** parameter references memory into which is placed the time in milliseconds taken to perform the scan up to the point identified by **index**.

See Also

Statistics and Related Functions (page 159)

kv_detailed_metrics_list_get_name()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_detailed_metrics_list_get_name(
    const kv_detailed_metrics_list_t *res,
    kv_int_t index,
    char **name)
```

Returns the name of the partion or shard which was examined by the parallel scan.

Parameters

res

The **res** parameter is the detailed metrics list for which you want the partition or shard name. It is created using either kv_parallel_scan_get_partition_metrics() (page 170) or kv_parallel_scan_get_shard_metrics() (page 171).

index

The **index** parameter is the point in the scan from which you want to return the partition or shard name.

name

The **name** parameter references memory into which is placed the shard or partition name.

See Also

Statistics and Related Functions (page 159)

kv_get_node_metrics()

```
#include <kvstore.h>

kv_node_metrics_t *
kv_get_node_metrics(kv_stats_t *stats,
                    kv_int_t index)
```

Returns a list of metrics associated with each node in the store. The information is returned using a `kv_node_metrics_t` structure, which includes the following data members:

- `kv_int_t avg_latency_ms`

  Returns the trailing average latency (in ms) over all requests made to this node.

- `kv_int_t max_active_request_count`

  Returns the number of requests that were concurrently active for this node at this Oracle NoSQL Database client.

- `kv_long_t request_count`

  Returns the total number of requests processed by the node.

- `kv_int_t is_active`

  Returns `1` if the node is currently active. That is, it is reachable and can service requests.

- `kv_int_t is_master`

  Returns `1` if the node is currently a master.

- `const char *node_name`

  Returns the internal name associated with the node.

- `const char *zone_name;`

  Returns the name of the zone which hosts the node.

Note that if the **index** parameter is out of range, then this functions returns `NULL`.

Parameters

stats

The **stats** parameter is the statistics structure containing the node metrics information. This structure is allocated using kv_get_stats() (page 169), and is released using kv_release_stats() (page 173).

index

The **index** parameter is the integer designation of the node for which you want to retrieve statistical information. You can discover the total number of nodes for which statistical information is available using kv_get_num_nodes() (page 166).

See Also

Statistics and Related Functions (page 159)

kv_get_num_nodes()

```
#include <kvstore.h>

kv_int_t
kv_get_num_nodes(const kv_stats_t *stats)
```

Returns the total number of nodes currently in the store, active and inactive.

Parameters

stats

The **stats** parameter is the structure containing the statistical information that you want to examine. This structure is allocated using kv_get_stats() (page 169), and is released using kv_release_stats() (page 173).

See Also

Statistics and Related Functions (page 159)

kv_get_num_operations()

```
#include <kvstore.h>

kv_int_t
kv_get_num_operations(const kv_stats_t *stats)
```

Returns the total number of store operations described by the provided statistics structure.

Parameters

stats

The **stats** parameter is the statistics structure containing the node metrics information. This structure is allocated using kv_get_stats() (page 169), and is released using kv_release_stats() (page 173).

See Also

Statistics and Related Functions (page 159)

kv_get_operation_metrics()

```
#include <kvstore.h>

kv_operation_metrics_t *
kv_get_operation_metrics(kv_stats_t *stats,
                         kv_int_t index)
```

Aggregates the metrics associated with an Oracle NoSQL Database operation. The information is returned using a `kv_operation_metrics_t` structure, which includes the following data members:

- `kv_float_t avg_latency_ms`

  Returns the average latency associated with the operation in milliseconds.

- `kv_int_t max_latency_ms`

  Returns the maximum latency associated with the operation in milliseconds.

- `kv_int_t min_latency_ms`

  Returns the minimum latency associated with the operation in milliseconds.

- `kv_int_t total_operations`

  Returns the number of operations that were executed.

- `const char *operation_name`

  Returns the name of the Oracle NoSQL Database operation associated with the metrics.

Note that if the **index** parameter is out of range, then this functions returns `NULL`.

Parameters

stats

The **stats** parameter is the statistics structure containing the operation metrics information. This structure is allocated using kv_get_stats() (page 169), and is released using kv_release_stats() (page 173).

index

The **index** parameter is the integer designation of the operation for which you want to retrieve statistical information. You can discover the total number of operations for which statistical information is available using kv_get_num_operations() (page 167).

See Also

Statistics and Related Functions (page 159)

kv_get_stats()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_get_stats(kv_store_t *store,
             kv_stats_t **stats,
             kv_int_t clear)
```

Returns a statistics structure, which you can then examine using kv_get_node_metrics() (page 164), kv_get_operation_metrics() (page 168), or kv_stats_string() (page 174). You release the resources allocated for the statistics structure using kv_release_stats() (page 173).

Parameters

store

The **store** parameter is the handle to the store for which you want to examine statistical information.

stats

The **stats** parameter references memory into which a pointer to the allocated statistics structure is copied.

clear

The **clear** parameter resets all counters within the statistics structure to zero. Setting this value to 1 creates a new reporting interval for which minimum, maximum, average, and total values are computed.

See Also

Statistics and Related Functions (page 159)

kv_parallel_scan_get_partition_metrics()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_scan_get_partition_metrics(
    const kv_parallel_scan_iterator_t *iterator,
    kv_detailed_metrics_list_t **result)
```

Gets the per-partition metrics for this parallel scan. This may be called at any time during the iteration in order to obtain metrics to that point or it may be called at the end to obtain metrics for the entire scan.

Parameters

iterator

The **iterator** parameter is the iterator for which you want to return statistics. This iterator is allocated using kv_parallel_store_iterator() (page 93) or kv_parallel_store_iterator_keys() (page 95)

result

The **result** parameter is the list of parallel scan partition metrics. Use kv_detailed_metrics_list_get_record_count() (page 161), kv_detailed_metrics_list_get_scan_time() (page 162), and kv_detailed_metrics_list_get_name() (page 163) to examine this list. Return the size of this list using kv_detailed_metrics_list_size() (page 160). Release this list using kv_release_detailed_metrics_list() (page 172).

See Also

Statistics and Related Functions (page 159)

kv_parallel_scan_get_shard_metrics()

```
#include <kvstore.h>

kv_error_t (page 185)
kv_parallel_scan_get_shard_metrics(
    const kv_parallel_scan_iterator_t *iterator,
    kv_detailed_metrics_list_t **result)
```

Gets the per-shard metrics for this parallel scan. This may be called at any time during the iteration in order to obtain metrics to that point or it may be called at the end to obtain metrics for the entire scan.

Parameters

iterator

The **iterator** parameter is the iterator for which you want to return statistics. This iterator is allocated using kv_parallel_store_iterator() (page 93) or kv_parallel_store_iterator_keys() (page 95)

result

The **result** parameter is the list of parallel scan shard metrics. Use kv_detailed_metrics_list_get_record_count() (page 161), kv_detailed_metrics_list_get_scan_time() (page 162), and kv_detailed_metrics_list_get_name() (page 163) to examine this list. Return the size of this list using kv_detailed_metrics_list_size() (page 160). Release this list using kv_release_detailed_metrics_list() (page 172).

See Also

Statistics and Related Functions (page 159)

kv_release_detailed_metrics_list()

```
#include <kvstore.h>

void
kv_release_detailed_metrics_list(kv_detailed_metrics_list_t **results)
```

Releases the resources used by the detailed metrics list, as created by kv_parallel_scan_get_partition_metrics() (page 170) and kv_parallel_scan_get_shard_metrics() (page 171).

Parameters

results

The **results** parameter is the detailed metrics list that you want to release.

See Also

Statistics and Related Functions (page 159)

kv_release_stats()

```
#include <kvstore.h>

void
kv_release_stats(kv_stats_t **stats)
```

Releases all the resources allocated for the provided statistics structure. The statistics structure is initially allocated using kv_get_stats() (page 169).

Parameters

stats

The **stats** parameter is the statistics structure that you want to release.

See Also

Statistics and Related Functions (page 159)

kv_stats_string()

```
#include <kvstore.h>

const char *
kv_stats_string(kv_store_t *store,
                const kv_stats_t *stats)
```

Returns a descriptive string containing metrics for each operation that was actually performed during the statistics gathering interval, one per line.

Parameters

store

The **store** parameter the handle to the store for which you want to examine statistical information.

stats

The **stats** parameter is the structure containing the statistical information. This structure is allocated using kv_get_stats() (page 169), and is released using kv_release_stats() (page 173).

See Also

Statistics and Related Functions (page 159)

Chapter 8. Error Functions

This chapter contains functions used to investigate and examine error returns obtained from the C API functions described in this manual.

Most methods in this library return an enumeration, kv_error_t (page 185), indicating success or failure. Because of the need to also return integer and "boolean-like" values in some cases, all actual error values are negative. 0 (KV_SUCCESS) indicates no error. Valid integer return values are non-negative. When an error is returned the kv_get_last_error() (page 177) method may have additional information about the error.

Error Functions

Error Functions	Description
kv_get_last_error()	Return a string explaining the last error

kv_get_last_error()

```
#include <kvstore.h>

const char *
kv_get_last_error(kv_store_t *store)
```

Returns a string explaining the last error. This string is only useful immediately following an error return; otherwise it may indicate an older, irrelevant error. This value is maintained per-thread and is not valid across threads.

Parameters

store

The **store** parameter is the handle to the store in which an operation returned an error.

See Also

Error Functions (page 176)

Appendix A. Data Types

This appendix describes the enum datatypes used by the various Oracle NoSQL Database functions:

Data Operations Data Types

This section defines the data types used by the functions described in this chapter.

kv_depth_enum

```
typedef enum {
    KV_DEPTH_DEFAULT = 0,
    KV_DEPTH_CHILDREN_ONLY,
    KV_DEPTH_DESCENDANTS_ONLY,
    KV_DEPTH_PARENT_AND_CHILDREN,
    KV_DEPTH_PARENT_AND_DESCENDANTS
} kv_depth_enum;
```

Used with multiple-key and iterator operations to specify whether to select (return or operate on) the key-value pair for the parent key, and the key-value pairs for only immediate children or all descendants.

Options are:

- `KV_DEPTH_DEFAULT`

  No depth constraints are placed on the operation.

- `KV_DEPTH_CHILDREN_ONLY`

  Select only immediate children, do not select the parent.

- `KV_DEPTH_DESCENDANTS_ONLY`

  Select all descendants, do not select the parent.

- `KV_DEPTH_PARENT_AND_CHILDREN`

  Select immediate children and the parent.

- `KV_DEPTH_PARENT_AND_DESCENDANTS`

  Select all descendants and the parent.

kv_direction_enum

```
typedef enum {
    KV_DIRECTION_FORWARD,
    KV_DIRECTION_REVERSE,
    KV_DIRECTION_UNORDERED
} kv_direction_enum;
```

Used with iterator operations to specify the order that keys are returned.

- `KV_DIRECTION_FORWARD`

  Iterate in ascending key order.

- KV_DIRECTION_REVERSE

  Iterate in descending key order.

- KV_DIRECTION_UNORDERED

  Iterate in no particular key order.

kv_presence_enum

```
typedef enum {
    KV_IF_DONTCARE = 0,
    KV_IF_ABSENT,
    KV_IF_PRESENT
} kv_presence_enum;
```

Defines under what circumstances a Key/Value record will be put into the store if kv_put_with_options() (page 98) is in use.

- KV_IF_DONTCARE

  The record is put into the store without constraint.

- KV_IF_ABSENT

  Put the record into the store only if a value for the the supplied key does not currently exist in the store.

- KV_IF_PRESENT

  Put the record into the store only if a value for the supplied key does currently exist in the store.

kv_return_value_version_enum

```
typedef enum {
    KV_RETURN_VALUE_NONE = 0,
    KV_RETURN_VALUE_ALL,
    KV_RETURN_VALUE_VALUE,
    KV_RETURN_VALUE_VERSION
} kv_return_value_version_enum;
```

Used with put and delete operations to define what to return as part of the operations.

- KV_RETURN_VALUE_NONE

  Do not return the value or the version.

- KV_RETURN_VALUE_ALL

  Return both the value and the version.

- KV_RETURN_VALUE_VALUE

Return the value only.

- KV_RETURN_VALUE_VERSION

  Return the version only.

Durability and Consistency Data Types

This section defines the data types used to support durability and consistency policies.

kv_ack_policy_enum

```
typedef enum {
    KV_ACK_ALL = 1,
    KV_ACK_NONE = 2,
    KV_ACK_MAJORITY = 3
} kv_ack_policy_enum;
```

A replicated environment makes it possible to increase an application's transaction commit guarantees by committing changes to its replicas on the network. This enumeration defines the policy for how such network commits are handled.

Ack policies are set as a part of defining a durability guarantee. You create a durability guarantee using kv_create_durability() (page 145).

Possible ack policies are:

- `KV_ACK_ALL`

  All replicas must acknowledge that they have committed the transaction.

- `KV_ACK_NONE`

  No transaction commit acknowledgments are required and the master will never wait for replica acknowledgments.

- `KV_ACK_MAJORITY`

  A simple majority of replicas must acknowledge that they have committed the transaction.

kv_consistency_enum

```
typedef enum {
    KV_CONSISTENCY_ABSOLUTE = 0,
    KV_CONSISTENCY_NONE,
    KV_CONSISTENCY_TIME,
    KV_CONSISTENCY_VERSION,
    KV_CONSISTENCY_NONE_NO_MASTER
} kv_consistency_enum;
```

Enumeration that is used to define the consistency guarantee used for read operations. Values are:

- `KV_CONSISTENCY_ABSOLUTE`

  A consistency policy that requires a read transaction be serviced on the Master so that consistency is absolute.

- `KV_CONSISTENCY_NONE`

A consistency policy that allows a read transaction performed at a Replica to proceed regardless of the state of the Replica relative to the Master.

- `KV_CONSISTENCY_TIME`

  A consistency policy which describes the amount of time the Replica is allowed to lag the Master. This policy cannot be specified using kv_create_simple_consistency() (page 146). Instead, use kv_create_time_consistency() (page 147).

- `KV_CONSISTENCY_VERSION`

  A consistency policy which ensures that the environment on a Replica node is at least as current as that used by the Value provided to kv_get_version() (page 139), or by the result set provided to kv_result_get_version() (page 107) or kv_result_get_previous_version() (page 105).

  This policy cannot be specified using kv_create_simple_consistency() (page 146). Instead, use kv_create_version_consistency() (page 149).

- `KV_CONSISTENCY_NONE_NO_MASTER`

  A consistency policy that requires a read operation be serviced on a replica; never the Master. When this consistency policy is used, the read operation will not be performed if the only node available is the Master.

  For read-heavy applications (ex. analytics), it may be desirable to reduce the load on the master by restricting the read requests to only the replicas in the store. Use of the secondary zones feature is preferred over this consistency policy as the mechanism for achieving this sort of read isolation. But for cases where the use of secondary zones is either impractical or not desired, this consistency policy can be used to achieve a similar effect; without employing the additional resources that secondary zones may require.

kv_sync_policy_enum

```
typedef enum {
    KV_SYNC_NONE = 1,
    KV_SYNC_FLUSH = 2,
    KV_SYNC_WRITE_NO_SYNC = 3
} kv_sync_policy_enum;
```

Defines the synchronization policy to be used when committing a transaction. High levels of synchronization offer a greater guarantee that the transaction is persistent to disk, but trade that off for lower performance.

Sync policies are set as a part of defining a durability guarantee. You create a durability guarantee using kv_create_durability() (page 145).

Possible sync policies are:

- `KV_SYNC_NONE`

Do not write or synchronously flush the log on transaction commit.

- `KV_SYNC_FLUSH`

  Write and synchronously flush the log on transaction commit.

- `KV_SYNC_WRITE_NO_SYNC`

  Write but do not synchronously flush the log on transaction commit.

Store Operations Data Types

This section defines data types that are by the store or API at a high level, or data types that are commonly used by all areas of the API.

kv_api_type_enum

```
typedef enum {
    KV_JNI
} kv_api_type_enum;
```

Structure used to describe the API implementation type. Currently only one option is available: `KV_JNI`.

kv_error_t

```
typedef enum {
    KV_SUCCESS = 0,
    KV_NO_MEMORY = -1,
    KV_NOT_IMPLEMENTED = -2,
    KV_ERROR_JVM = -3,
    KV_KEY_NOT_FOUND = -4,
    KV_KEY_EXISTS = -5,
    KV_NO_SUCH_VERSION = -6,
    KV_NO_SUCH_OBJECT = -7,
    KV_INVALID_OPERATION = -8,
    KV_INVALID_ARGUMENT = -9,
    KV_TIMEOUT = -10,
    KV_CONSISTENCY = -11,
    KV_DURABILITY = -12,
    KV_FAULT = -13,
    KV_AVRO = -14,
    KV_AUTH_FAILURE = -15,
    KV_AUTH_REQUIRED = -16,
    KV_ACCESS_DENIED = -17,
    KV_ERROR_JAVA_UNKNOWN = -99,
    KV_ERROR_UNKNOWN = -100
} kv_error_t

#define KV_FALSE 0
#define KV_TRUE 1
```

All non-void API methods return `kv_error_t`. With few exceptions a return value of KV_SUCCESS (or 0) means no error and a negative value means an error.

The exceptions are the methods that return integer values. In these cases a negative return means an error and a non-negative return is the correct value.

kv_store_iterator_config_t

```
typedef struct {
```

```
    kv_int_t max_conc_req;
    kv_int_t max_res_batches;
} kv_store_iterator_config_t;
```

Used to configure a parallel scan of the store.

max_conc_req identifies the maximum number of concurrent requests the parallel scan will make. That is, this is the maximum number of client-side threads that are used to perform this scan. Setting this value to 1 causes the store iteration to be performed using only the current thread. Setting it to 0 lets the KV Client determine the number of threads based on topology information (up to a maximum of the number of available processors). Values less than 0 are reserved for some future use and cause an error to be returned.

max_res_batches specifies the maximum number of results batches that can be held in the NoSQL Database client process before processing on the Replication Node pauses. This ensures that client side memory is not exceeded if the client cannot consume results as fast as they are generated by the Replication Nodes. The default value is the value specified for **max_conc_req**.