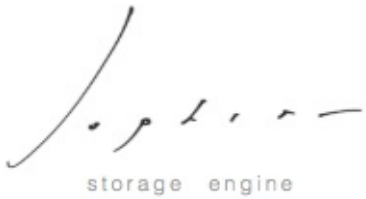
# Table of Contents




Hybrid Embeddable Key-Value Storage.

# Sophia 2.1 Manual

Welcome to the Sophia 2.1 Manual.

Sophia is advanced Embeddable Transactional Key-Value Storage. Open-Source, available free of charge under terms of BSD License.

This manual is also available in PDF .

## RAM-Disk Hybrid

Sophia has unique hybrid architecture that was specifically designed to efficiently store data using the combination of HDD, Flash and RAM.

Sophia allows to distinct Hot (read-intensive) and Cold data.

The data storage engine was created as a result of research and reconsideration primary algorithmic constraints of Log-file based data structures.

Write and Range Scan optimized. It can efficiently work with terrabyte-sized datasets.

## Language bindings

Bindings for the most common languages are available here.

## v2.1 features

- Full ACID compliancy
- Multi-Version Concurrency Control (MVCC) engine
- Pure Append-Only
- Multi-threaded (Both client access and near-linear compaction scalability)
- Multi-databases support (Single environment and WAL)
- Multi-Statement and Single-Statement Transactions (Multi-databases)
- Serialized Snapshot Isolation (SSI)
- Persistent RAM Storage mode
- Persistent Caching mode
- Anti-Cache Storage mode
- LRU Storage
- Separate storage formats: key-value (default), document (keys are part of value)

- Optional AMQ Filter (Approximate member query filter) based on Quotient Filter
- Async and sync reads (Callback triggered vs. blocking)
- Upsert: optimized write-only 'Update or Insert' operation
- Consistent Cursors
- Prefix search
- Prefix compression (Using key-part duplicates)
- Point-in-Time Views
- Online/Versional database creation and asynchronous shutdown/drop
- Asynchronous Online/Hot Backup
- Compression (Per region, no-holes, supported: lz4, zstd)
- Compression for Hot and Cold data (distinct compression types)
- Meta-data Compession (By default)
- Optimizations for faster recovery with big databases (Snapshot)
- Easy to use (Minimalistic API)
- Easy to integrate into a DBMS (Native support of using as storage engine)
- Easy to write bindings (FFI-friendly, API designed to be stable in future)
- Easy to built-in (Amalgamated, compiles into two C files)
- Event loop friendly
- Zero-Configuration (Tuned by default)
- Implemented as small *C-written* library with zero dependencies
- Carefully tested
- Open Source Software, *BSD* Licensed

# Build

Sophia has no external dependencies.

Run *make* to generate **sophia.c** and **sophia.h** and build the library.

```
make
```

Following command can be used to compile the library in your project:

```
cc -O2 -DNDEBUG -std=c99 -pedantic -Wall -Wextra -pthread -c sophia.c
```

To build and run tests:

```
cd test
make
./sophia-test
```

# Sophia API

Sophia defines a small set of basic methods which can be applied to any database object.

All API declarations are stored in a separate include file: **sophia.h**

Configuration, Control, Transactions and other objects are accessible using the same methods. Methods are called depending on used objects. Methods semantic may slightly change depending on used object.

All functions return either 0 on success, or -1 on error. The only exception are functions that return a pointer. In that case NULL might indicate an error.

sp_error() function can be used to check if any fatal errors occured leading to complete database shutdown. All created objects must be freed by sp_destroy() function.

All methods are thread-safe and atomic.

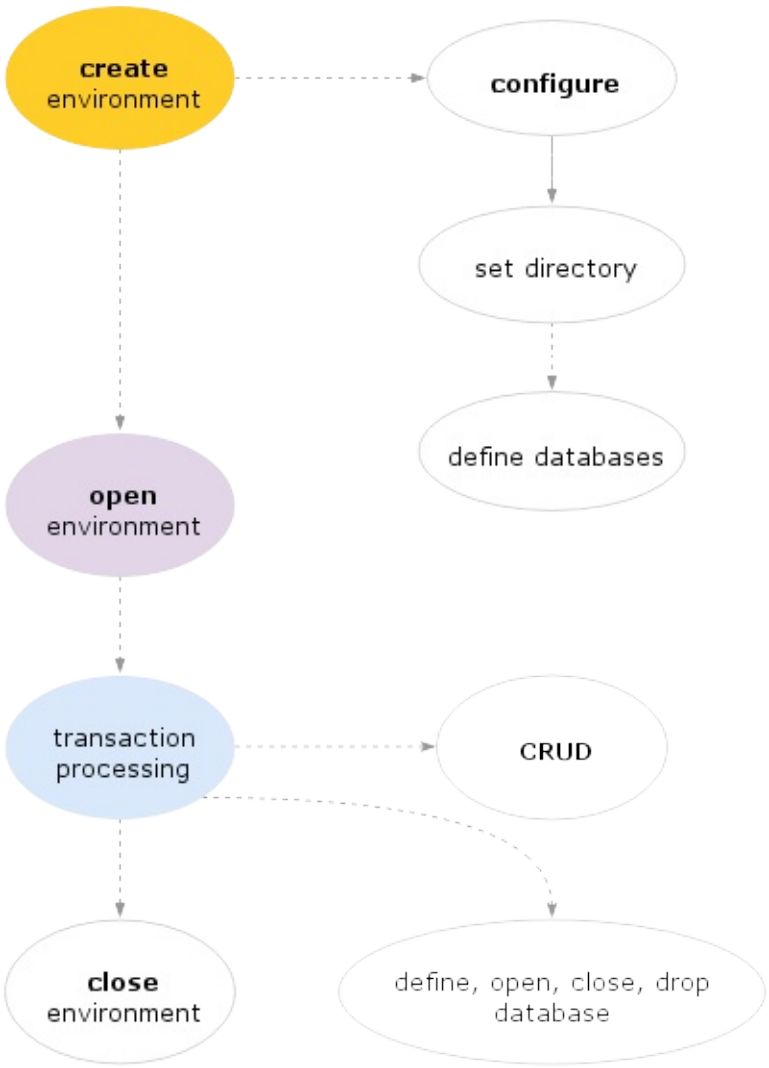
Please take a look at the API manual section for additional details.

- sp_env()
- sp_document()
- sp_setstring()
- sp_setint()
- sp_getobject()
- sp_getstring()
- sp_getint()
- sp_open()
- sp_destroy()
- sp_error()
- sp_poll()
- sp_drop()
- sp_set()
- sp_upsert()
- sp_delete()
- sp_get()
- sp_cursor()
- sp_begin()
- sp_commit()

# Common Workflow

Basic workflow is simple:

1. create sophia environment sp_env()
2. set options using sp_setint(), sp_setstring(), define **sophia.path**
3. define databases
4. sp_open() environment
5. do transaction processing using sp_document(), sp_set(), sp_get(), sp_delete(), sp_upsert(), sp_cursor(), sp_begin(), sp_commit(), sp_destroy()
6. on finish: sp_destroy() the environment object




```
void *env = sp_env();
sp_setstring(env, "sophia.path", "./storage", 0);
sp_setstring(env, "db", "test", 0);
sp_open(env);
void *db = sp_getobject(env, "db.test");
/* do transactions */
sp_destroy(env);
```

# Configuration

Every Sophia configuraton, monitoring, database creation, etc. is done using sysctl-alike interface.

Operations sp_setstring(), sp_getstring(), sp_setint(), sp_getint(), sp_getobject() and sp_cursor() are used to set, get and iterate through configuration fields.

Most of the configuration can only be changed before opening an environment.

Any error description can be accessed through **sophia.error** field.

Set example:

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

Get example:

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

To get a list of all system objects and configuration values:

```
void *cursor = sp_getobject(env, NULL);
void *ptr = NULL;
while ((ptr = sp_get(cursor, ptr))) {
    char *key = sp_getstring(ptr, "key", NULL);
    char *value = sp_getstring(ptr, "value", NULL);
    printf("%s", key);
    if (value)
        printf(" = %s\n", value);
    else
        printf(" = \n");
}
sp_destroy(cursor);
```

# How It Works

To describe how Sophia works internally, we will use a simple case as a guiding example:

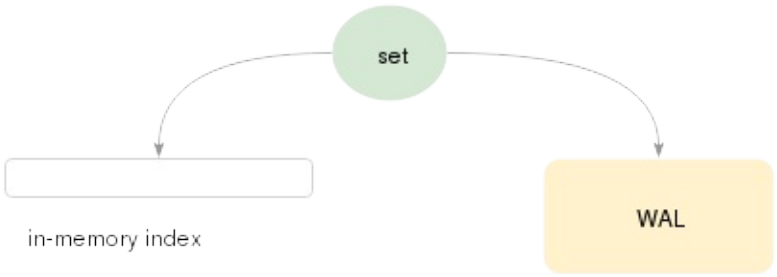
- **a**. fill our empty database with the keys in random order
- **b**. read all stored keys in original order

Please take a look at the Architecture manual section for more details.

# (a) Fill empty database

We will start by inserting 1 million random keys.

During first 200K Set operations, inserted keys first go to the sorted in-memory index. Second, in order to maintain persistence, exact operation information is written to write-ahead log.



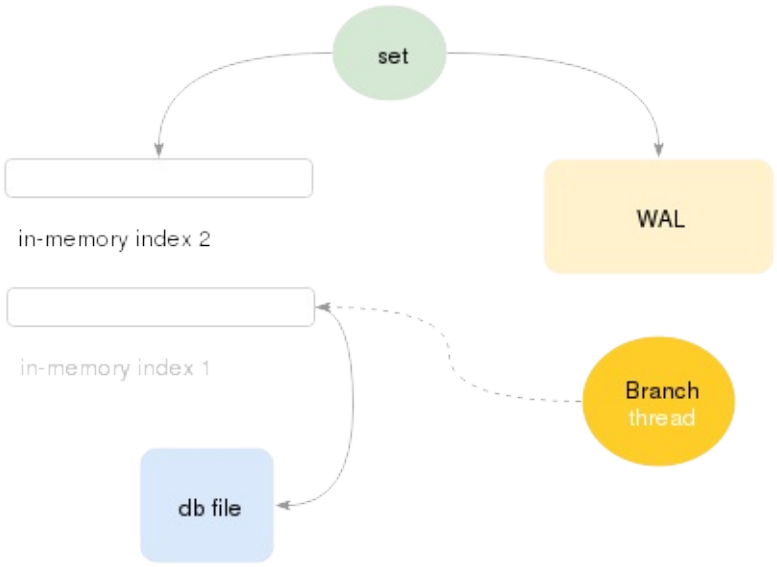

At this point, we have 100K keys stored in-memory and the same amount of keys written to write-ahead log.

We will continue to insert keys from 200K to 500K.

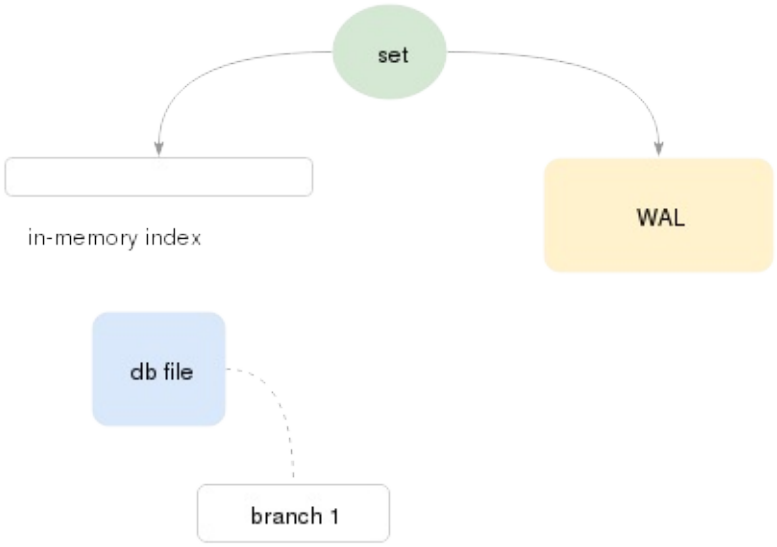
When in-memory index becomes too large, Sophia scheduler makes an attempt to move this index from memory to disk. In-memory index dump stored on-disk is called **Branch**. To save the branch, a new file is created. Let's call it **db file** for now.

Branch creation process is launched in the background by one of the dedicated Sophia worker threads. During the process, Sophia creates a **second in-memory index** to reduce an effect on parallel going Set operations.

Inserts are now silently go to the second index.

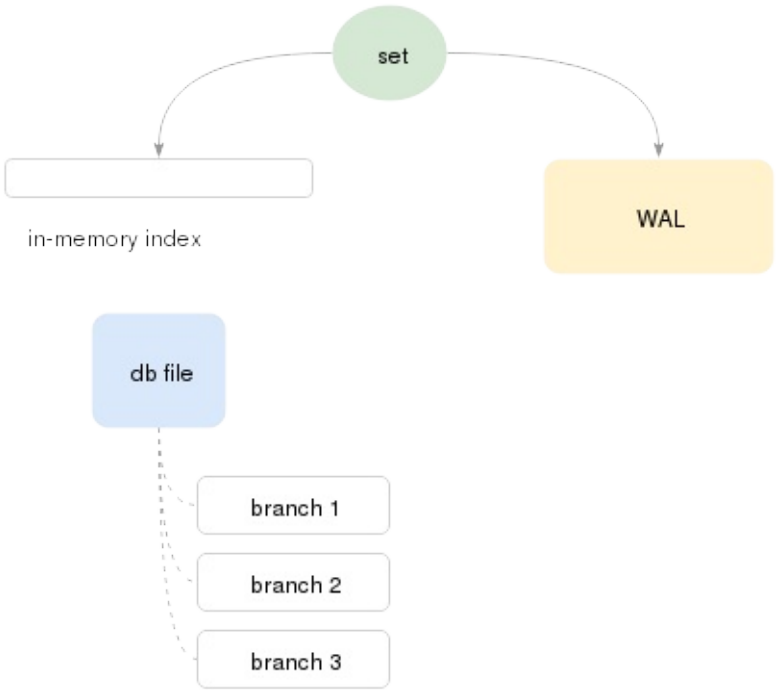



When the Branch creation process is completed, *first in-memory index* got freed.

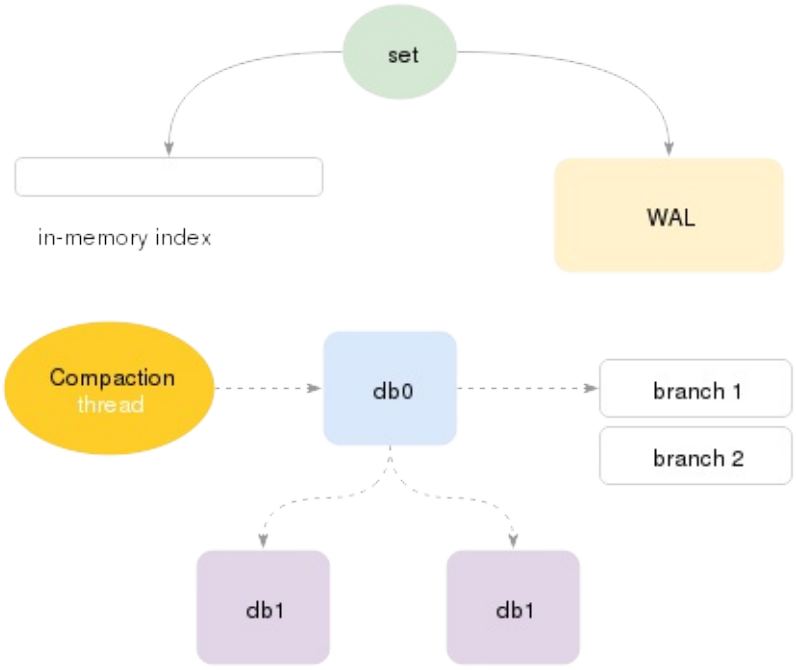



Yet, we still keep on to inserting our keys from 500K to 700K.

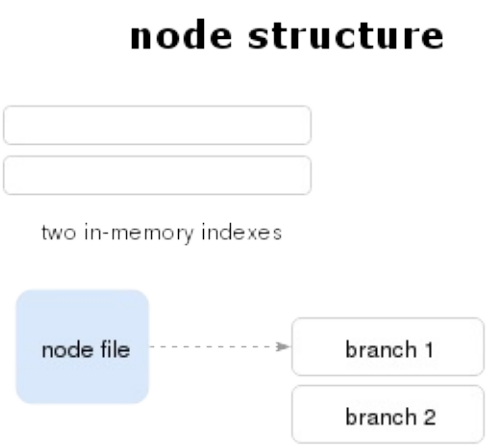
The circle of in-memory index overrun and branch creation continues until the number of created branches becomes too big. All branches are appended to the end of **db file**.




When the number of branches hit some *watermark* number, Sophia scheduler starts **Compaction** process for this **db file**. During the compaction process, all keys stored in each branch are merged, which leads to creation of **one** or **more** new db files.




Now we are ready to introduce new important term: each pair of in-memory indexes and its associated db files is called a **Node** in Sophia.




So basically, a **Node** represents a Sophia database file which stores a range of sorted keys. Those keys are stored in one or more Branches.

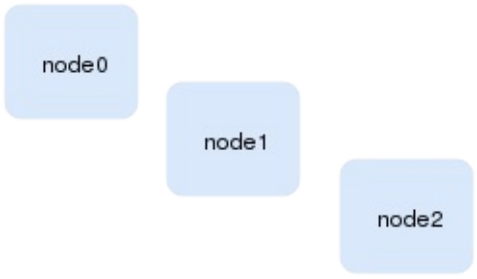
When a Node becomes too big (in terms of Branch numbers), it splits into **two** or **more** Nodes by background thread.

It is important that each Node key is strictly separated from the other keys. It is also crucial that Node in Sophia is a unit of such background operations like: Branch or Compaction, Gc, Backup and so on. Several nodes can be **processed in parallel** by Sophia worker-threads.

But let's return to our case.

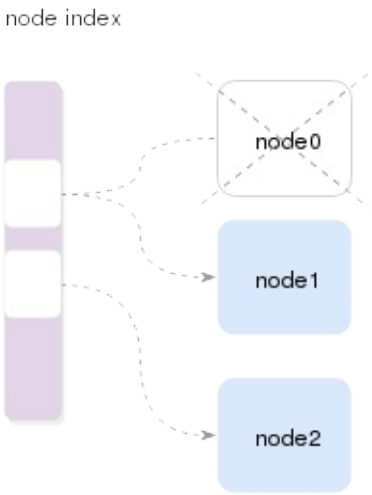
At this point we have **one node** (automatically created during database deploy), which has: (a) in-memory index with some keys in it, (b) several **Branches** (previously sorted in-memory index dumps) are saved in the **node db file**. All operations have been saved in WAL in serial order.

Since the number of branches became too big, Sophia scheduler starts the **Compaction** process and creates **two** new Nodes.

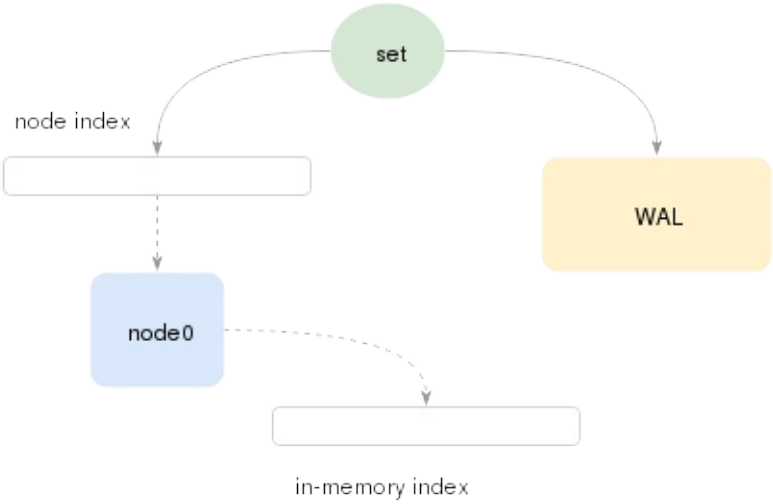



So basically, each new created node represents a half of sorted keys stored in original node. This applies also to in-memory index. Keys inserted into origin in-memory index during compaction must be moved to new node indexes. This process runs in parallel with on-going Sets, so efficient implementation is tricky.

When compaction process is finished, origin node is deleted too, and new created nodes are silently inserted into in-memory **Node Index**.

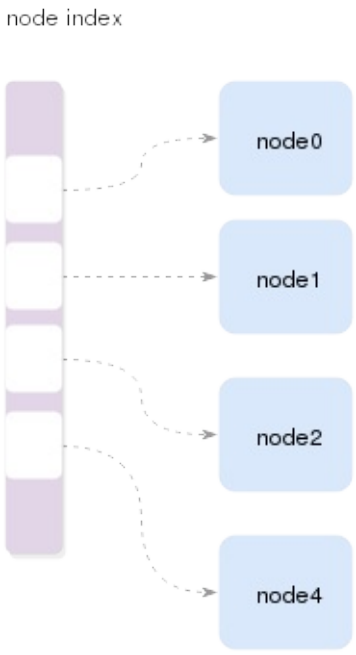



A **Node Index** is used for correct routing during key insert or search. Sophia is aware about min/max of each node, so this information is used during a Node search.




We still continue to insert our keys. Now range varies from 700K to 1M range.

The circle of branch/compaction and node creation continues, and by the end of insertion case our database consists of **four** nodes.



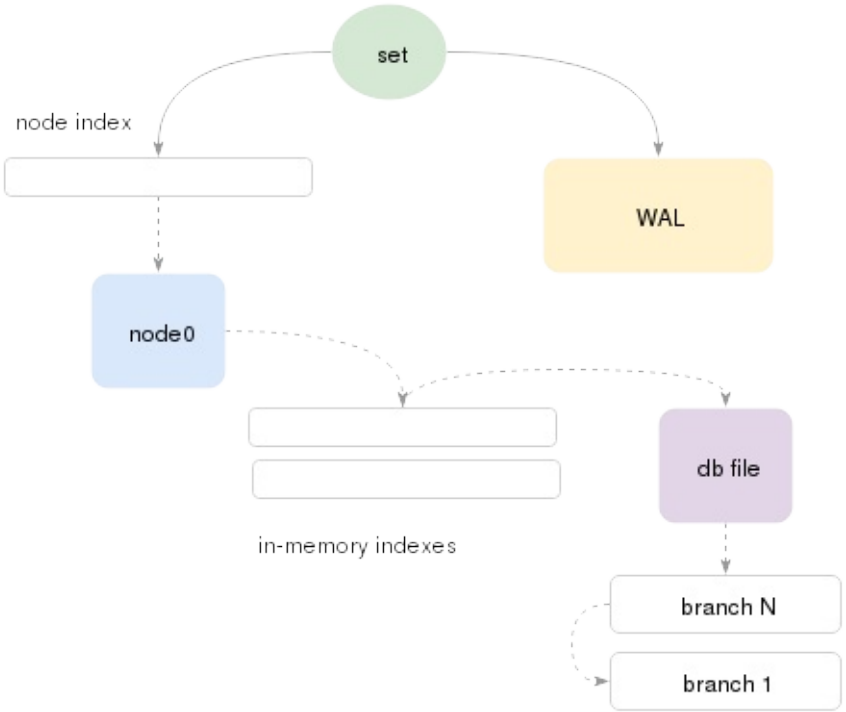

Sophia is designed to efficiently utilize available memory.

If there is more memory available, then branch/compaction operations become more infrequent and system becomes more disk-efficient. Best performance can be obtained with no memory limit set. Sophia is Hard-Drive (and Flash) friendly, since all operations are delayed and executed in large sequential reads and writes, without overwrite.

If there is a memory-limit set, Sophia scheduler is aware about nodes that have biggest in-memory indexes. These are processed first to efficiently free memory.

# (b) Random read

We start to read 1 million keys in origin order, which is random.

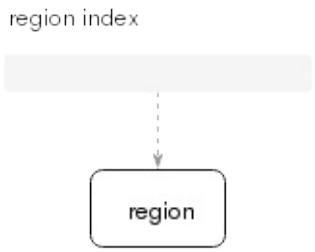


During the Get (search), only branch regions that have min <= key <= max are examined. When the node found, search is performed in following structures:

1. first in-memory index
2. second in-memory index
3. each node branch: strictly starting from the last one

Branch format is highly optimized to reduce disk access during reads. Internally each Branch consist of **Region Index** and **Sorted Regions**.

So basically, this structures splits whole Branch range of keys into smaller regions. Region Index holds information about every Regions stored in the Branch, their min/max values and meta-data.




Region Indexes are loaded into memory during database opening. They allow to find exact region to read from disk during search and to reduce overall search times.

There is a game between available memory, a number of Branches and Search times. Each additional branch says that there is a possible additional disk access during the search. In the same time, it is unable to maintain memory limits without branching, because compaction times are greater than possible rate of incoming data.

Sophia is designed to be read optimized. There is a high possibility that latest created Branches (hot data) are stored in the file system cache. Scheduler is aware about nodes which have largest in-memory Key Index and biggest number of Branches. These are processed first.

Ideally, Sophia scheduler tries to make that each node will have 1 branch. In that case: O(1) disk seek time is guaranteed during search.

# Examples

| name | link |
| --- | --- |
| set, get, delete example | github |
| Transactions example | github |
| Cursor example | github |
| Point-in-Time View example | github |
| Multi-part keys example | github |
| Upsert example | github |

# Sophia License

# Version

Sophia has two versions: *release version* and *storage format version*.

Release version is the current version number. Sophia uses a common practice for version naming: **major**.**minor**.**latest_fix_number**.

Current release version can be read from **sophia.version** variable.

Storage format version follows the same rule. The number of version is equal to the previous Sophia version, that contains modified storage format.

Current storage format version can be read from **sophia.version_storage** variable.

Any Sophia releases are storage format compatible if storage versions are equal.

# Database creation

Database can be created, opened or deleted before or after environment startup. To create a database, new database name should be set to *db* configuration namespace. If no database exists, it will be created automatically.

Sophia v2.1 does not save database scheme information.

By default databases are created in **sophia.path/database_name** folder. It is possible to set custom folder per database using **db.database_name.path** It might be useful to separate databases on different disk drives.

Create or open database before environment start:

```
void *env = sp_env();
sp_setstring(env, "sophia.path", "./storage", 0);
sp_setstring(env, "db", "test", 0);
sp_open(env);
void *db = sp_getobject(env, "db.test");
sp_destroy(env);
```

# Database schema

By default database index type is **string**. Following index key types are supported: **string**, **u32**, **u64**, **u32_rev**, **u64_rev**.

```
void *env = sp_env();
sp_setstring(env, "db.test.index.key", "u32", 0);
```

Sophia supports multi-part keys:

```
void *env = sp_env();
sp_setstring(env, "db.test.index.key", "u32", 0);
sp_setstring(env, "db.test.index", "key_b", 0);
sp_setstring(env, "db.test.index.key_b", "string", 0);
...
void *o = sp_object(db);
sp_setstring(o, "key", &key_a, 0);
sp_setstring(o, "key_b", "hello", 0);
sp_set(db, o);
```

# Online database creation

Create or open database after environment start:

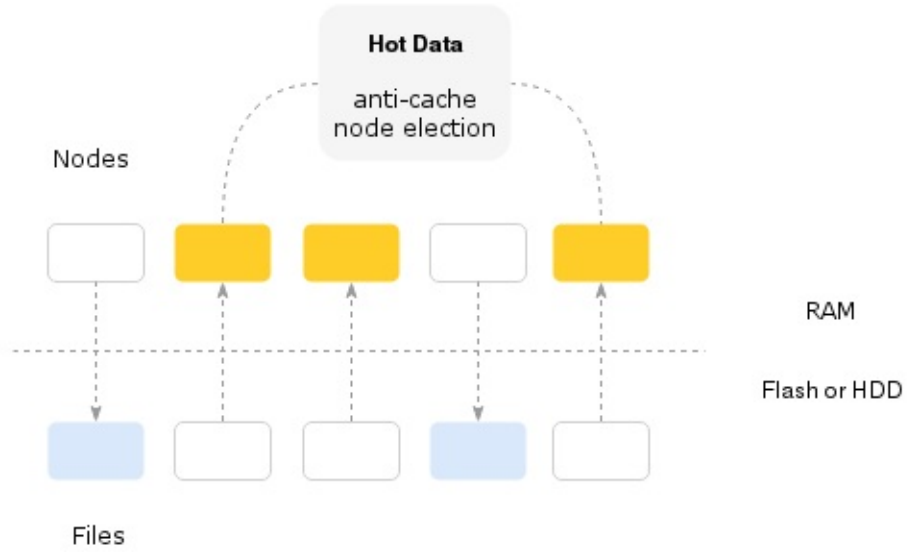
```
void *env = sp_env();
sp_setstring(env, "sophia.path", "./storage", 0);
sp_open(env);
sp_setstring(env, "db", "test", 0);
void *db = sp_getobject(env, "db.test");
sp_open(db);
sp_destroy(db); /* close database */
sp_destroy(env);
```

# Online database close and drop

To close a database the sp_destroy() method should be called on a database object. Note that sp_destroy() does not delete any data.

To schedule a database drop the sp_drop() method should be called on a database object. Actuall drop procedure will be automatically scheduled when a latest transaction completes.

# Anti-Cache Storage




Anti-Cache mode makes RAM a primary storage, while disk becomes a secondary storage.

Sophia implements the mode by introducing so-called index *Temperature*. The Temperature shows which nodes are the most frequently accessed for read (hot). These nodes are periodically elected and promoted to be put **in-memory**. Previously elected nodes are unloaded if they do not fit into memory limit.

Following variable can be set to enable anti-cache mode and set the memory limit: **db.database_name.storage** and **memory.anticache**.

```
/* set 1Gb anti-cache limit (for all databases) */
sp_setint(env, "memory.anticache", 1 * 1024 * 1024 * 1024);
/* switch a test database into anti-cache mode */
sp_setstring(env, "db.test.storage", "anti-cache", 0);
```

The election period is configurable by the following variable: **compaction.zone.anticache_period**.

```
/* schedule anti-cache node election every 5 minutes */
sp_setint(env, "compaction.0.anticache_period", 60 * 5);
```

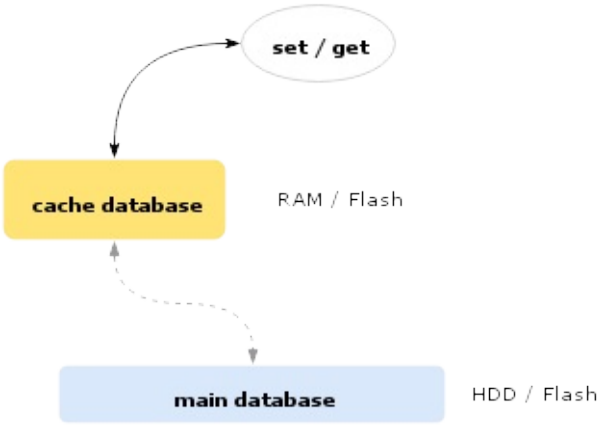
Database **db.database_name.temperature_histogram** can be examined to see current temperature distribution among index.

```
char *temperature = sp_getstring(env, "db.test.temperature_histogram", NULL);
...
free(temperature);
```

Please take a look at Database and Compaction configuration sections for additional details.

For pure in-memory storage mode see Persistent RAM Storage and Memory-Mapped Storage for semi-in-memory storage mode.

# Persistent Caching




Sophia allows to use a database as a *cache* for another one or more databases. Caching database is used to store records which are read from main database. Caching database can be placed in RAM or Flash, while main database can be stored on HDD. Caching database has the same storage format as a main database, it is also involved in Compaction. It must has an exact index keys types as the main database.

Following variable can be set to put a database into a cache mode: **db.database_name.cache_mode**. To assign a caching database for main database **db.database_name.cache** should be set with caching database name.

By enabling LRU Mode user can create Persistent LRU Cache, otherwise caching database will be a copy of the main database.

It is highly advisable to enable AMQ Filter for caching database. The filter will be used to reduce cache usage during search and reduce cache washout factor during invalidation (set, delete).

```
/* put cache db into cache mode */
sp_setint(env, "db.cache.cache_mode", 1);
/* set size of cache as 1 Gb */
sp_setint(env, "db.cache.lru", 1 * 1024 * 1024 * 1024);
/* enable AMQ Filter */
sp_setint(env, "db.cache.amqf", 1);

/* assign cache db as a cache for the main db */
sp_setstring(env, "db.main.cache", "cache", 0);
```

After that, all transactions with main database transparently start to go through caching database. Following logic is used:

- sp_get(main)
  1. attempt to find a key in caching database
  2. if the key is found
     - sp_set(cache, key) to maintain LRU logic
     - return key to user
  3. attempt to find a key in main database (if not 2)
  4. if the key is found
     - sp_set(cache, key)
     - return key to user
  5. not found
- sp_set(main)
  1. insert or replace key in main database
  2. replace (or do nothing if not found) key in caching database

- sp_delete(main)
  1. delete key in main database
  2. delete key in caching database

To update caching database uses sp_get(). **multi-statement** transactions *must* be used.

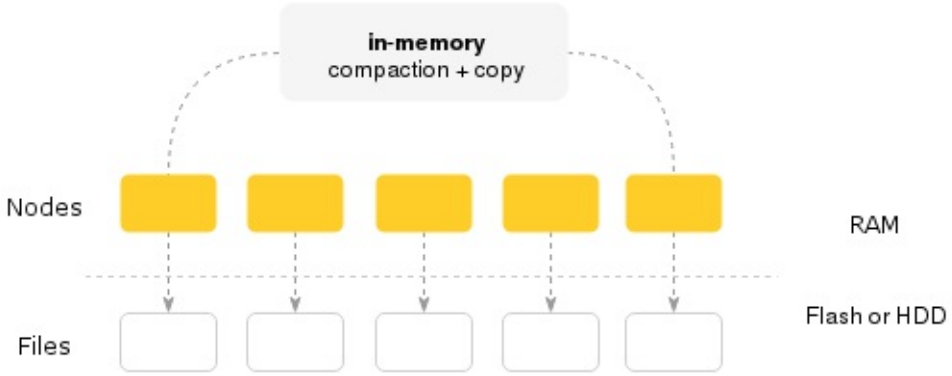
Single statement operation will only make an attempt to find a record, but not to save it back into caching database.

```
/* multi-statement */
void *transaction = sp_begin(env);
void *doc = sp_document(main);
sp_setstring(doc, "key", key, sizeof(key));
void *record = sp_get(transaction, doc);
if (record)
    sp_destroy(record);
/* commit will maintain Cache and LRU logic by
 * inserting record into caching database with higher
 * LSN number. */
sp_commit(transaction);

/* single-statement */
void *doc = sp_document(main);
sp_setstring(doc, "key", key, sizeof(key));
/* record will not be saved in caching database */
void *record = sp_get(main, doc);
if (record)
    sp_destroy(record);
```

Upsert operation sp_upsert() is not supported for the scheme. Cursor operation sp_cursor() does not use caching database and can separately be used with caching or main database.

*Please note*: since sp_get() statements are not involved in Write-Ahead Log, some of a latest updates may be lost in cache database after recovery. These are records which yet remain in-memory and yet being dumped to disk by compaction.

# Persistent RAM Storage




Sophia can store an exact copy of any node file into RAM.

All read requests are handled using RAM storage. The storage data continuously writing on disk using default background compaction procedure and Write Ahead Log.

To enable RAM storage a database storage mode should be set as **in-memory**.

```
sp_setstring(env, "db.test.storage", "in-memory", 0);
```

It is a good idea to switch Sophia into branch-less compaction mode. This mode allows to have an exact two-level model without necessity to create additional branches (index dumps), since all data is already stored in RAM.

```
sp_setint(env, "compaction.0.mode", 1);
```

During recovery, storage data files are read back into memory.

It is possible to enable and combine compression with RAM Storage.

# Memory-Mapped Mode

As an alternate way to use Persistent RAM storage and Anti-Cache modes could be mmap mode. By default Sophia uses pread(2) to read data from disk. Using mmap mode, Sophia handles all requests by directly accessing memory-mapped node files memory.

Following variable can be set to enable or disable mmap mode: **db.database_name.mmap**

```
sp_setint(env, "db.test.mmap", 1);
```

It is a good idea to try this mode, even if your dataset is rather small or you need to handle a large ratio of read request with an predictable pattern.

Disadvantage of mmap mode, in comparison to RAM Storage, is a possible unpredictable latency behaviour and a OS cache warmup period after recovery.

# LRU Mode

LRU stands for 'Least Recently Used' eviction algorithm.

When LRU mode is enabled, Sophia tries to maintain a database size limit by evicting oldest records. LSN number is used for eviction, which happens during compaction.

Following variable can be set to enable and set database size limit: **db.database_name.lru**

```
/* limit database size by 1Gb */
sp_setint(env, "db.test.lru", 1 * 1024 * 1024 * 1024);
```

LRU mode should be used in conjunction with Persistent Caching mode.

# AMQ Filter

AMQF stands for 'Approximate Member Query Filter'. The filter can be turned on to reduce a number of possible disk accesses during point-looks using sp_get() or sp_delete(). The filter is not used for range queries by sp_cursor(), cursor implementation has its own caching scheme.

Following variable can be set to enable or disable AMQF usage: **db.database_name.amqf**

```
sp_setint(env, "db.test.amqf", 1);
```

By default the filter is turned off, because normally there is no need for it. But there are some cases, when it can be useful.

The filter should be used to reduce LRU washout in Persistent Caching.

Sophia uses the Quotient Filter for the AMQF purpose.

# Compression

Following options can be set to enable or disable compression usage: **db.database_name.compression** and **db.database_name.compression_branch**.

It is possible to choose different compression types for Cold and Hot data (in terms of updates).

Most of data are stored in Cold branches (*compression*). While Hot data stored in a recently created branches (*compression_branch*).

```
sp_setstring(env, "db.test.compression", "lz4", 0);
sp_setstring(env, "db.test.compression_branch", "lz4", 0);
```

Supported compression values: **lz4**, **zstd**, **none** (default).

# RAM Compression

It is possible to enable and combine compression and RAM Storage Mode.

# Prefix Compression

Prefix compression is implemented by compressing mulit-part keys duplicates during compaction process.

To enable key compression:

```
sp_setint(env, "db.test.compression_key", 1);
```

# Point-in-Time Views

To create a View, new view name should be assigned to **view** configuration namespace.

It is possible to do sp_get() or sp_cursor() on a view object.

```
sp_setstring(env, "view", "today", 0);
void *view = sp_getobject(env, "view.today");
```

Views are not persistent, therefore view object must be recreated after shutdown before opening environment with latest view LSN number: **view.name.lsn**.

```
sp_setstring(env, "view", "today", 0);
sp_setint(env, "view.today.lsn", 12345);
```

To delete a view, sp_drop() or sp_destroy() should be called on a view object.

# Snapshot

During recovery, Sophia tries to read disk indexes. To reduce recovery time for big databases, Sophia periodically writes index dump to the disk generating a single *snapshot* file. This operation is called **Snapshot**.

Only indexes are saved during the operation, but database records remain untouched.

Snapshot period interval can be set or disabled using **compaction.zone.snapshot_period** variable.

```
/* take snapshot every 10 minutes */
sp_setint(env, "compaction.0.snapshot_period", 360);
```

Another important purpose of Snapshot is saving necessary statistic to distinct Hot and Cold node files used by Anti-Cache storage mode.

# Backup and Restore

Sophia supports asynchronous Hot/Online Backups.

Each backup iteration creates exact copy of environment, then assigns backup sequential number. Sophia v2.1 does not support incremental backup.

**backup.path** must be set with a specified folder which will contain resulting backup folders. To start a backup, user must initiate **backup.run** procedure first. Procedure call is fast and does not block.

```
sp_setint(env, "backup.run", 0);
```

**backup.active** and **backup.last_complete** variables can be examined to see if backup process is in progress or being succesfully completed.

Additionally, **scheduler.event_on_backup** can be enabled which will result in asynchronous notifications using **scheduler.on_event** function and sp_poll(). This might be helpful for a *event loop* integration.

Backups being made as a part of a common database workflow. It is possible to change backup priorities using compaction redzone settings.

To restore from a backup, a suitable backup version should be picked, copied and used as **sophia.path** directory.

# Compaction

Sophia Scheduler is responsible for planning all background tasks depending on current system load and selected profile (redzone).

The schedule handles following tasks: garbage collection, branch compaction, node compaction, log ration, lru, anti-cache election, async reads, and so on.

Sophia has multi-thread scallable compaction. Number of active background workers (threads) can be set using **scheduler.threads** variable.

```
sp_setint(env, "scheduler.threads", 5);
```

Please take a look at the Compaction and Scheduler configuration sections for more details.

# Compaction Configuration

Sophia compaction process is configurable via **redzone**. Redzone is a special value which represents current memory usage. Each redzone defines the background operations' priority, etc.

If no memory limit is set, redzone zero is used (default).

To create a new redzone, write a percent value into **compaction** namespace.

By default only *compaction.0* and *compaction.80* redzones are defined. When 80 percent of the memory usage is reached, checkpoint process starts automatically.

# Checkpoint

It is possible to start incremental asynchronous checkpointing process, which will force branch creation and memory freeing for every node in-memory index. Once a memory index log is free, files also will be automatically garbage-collected.

```
sp_setint(env, "scheduler.checkpoint", 0);
```

Procedure call is fast and does not block. **scheduler.checkpoint_active** and **scheduler.checkpoint_lsn_last** variables can be examined to see if checkpoint process is completed.

Checkpoints are automatically used to ensure a memory limit.

# Monitoring

Database monitoring is possible by getting current dynamic statistics and configuration via sp_env() object.

Performance metrics are described in the Performance configuration section.

To get current memory usage or trace every worker thread see Memory, Performance and Scheduler sections.

To get per-database metrics see Database sections.

# DBMS Integration

Sophia support special work modes which can be used to support external Write-Ahead Log: **log.enable** and **sophia.recover**.

# Two-Phase Recover

In this mode Sophia processes transactions to ensure that they were not already commited.

After second sp_open() Sophia starts environment.

1. env = sp_env()
2. sophia.recover = **2**
3. log.enable = 0
4. sp_open(env) start in recovery mode, compaction not started
    i. start defining and recovering databases
    ii. start replaying transactions from external WAL
    iii. sp_setint(transaction, "lsn", lsn) forge transaction *lsn* before commit
    iv. sp_commit(transaction) every commit ensures that data were not previously written to disk
5. sp_open(env) *second time* starts in default mode

This mode can be helpful for Sophia integration into other database management system, which supports its own Write-Ahead Log.

# N-Phase Recover

This recovery mode allows to switch Sophia into *recovery* mode and back on the fly.

1. env = sp_env()
2. sophia.recover = **3**
3. log.enable = 0
4. sp_open(env) start in **default** mode with thread-pool run.
5. usual transaction processing
6. sp_open(env) switch to *recovery* mode
7. start replaying transactions from external *source*
8. sp_setint(transaction, "lsn", lsn) forge transaction *lsn* before commit
9. sp_commit(transaction) every commit ensures that data were not previously written to disk
10. sp_open(env) *again* switch Sophia back to default mode.

Steps from 4-9 can be repeated any time.

This mode can be helpful for Sophia integration with most of a Replication/JOIN technologies.

# Transactions

Sophia supports fast optimistic single-statement and multi-statement transactions. Transactions are completely isolated from each other under Serializable Snapshot isolation (SSI).

# Single-statement transactions

Single-statement transactions are automatically processed when sp_set(), sp_delete(), sp_upsert(), sp_get() are used on a database object.

As a part of a transactional statement a key-value document must be prepared using sp_document() method.

First argument of sp_document() method must be an database object.

Object must be prepared by setting **key** and **value** fields, where value is optional. It is important that while setting **key** and **value** fields, only pointers are copied. Real data copies only during first operation.

Prepared document is automatically freed on commit.

```
void *db = sp_getobject(env, "db.test");
void *o = sp_document(db);
sp_setstring(o, "key", "hello", 0);
sp_setstring(o, "value", "world", 0));
sp_set(db, o); /* transaction */
o = sp_document(db);
sp_set(o, "key", "hello", 0);
sp_delete(db, o);
```

sp_get(database) method returns an document that is semantically equal to sp_document(database), but is read-only.

Example:

```
void *o = sp_document(db);
sp_set(o, "key", "hello", 0);
void *result = sp_get(db, o);
if (result) {
    int valuesize;
    char *value = sp_getstring(result, "value", &valuesize);
    printf("%s\n", value);
    sp_destroy(result);
}
```

# Multi-statement transactions

Multi-statement transaction is automatically processed when sp_set(), sp_delete(), sp_upsert(), sp_get() are used on a transactional object.

The sp_begin() function is used to start a multi-statement transaction.

During transaction, no updates are written to the database files until a sp_commit() is called. On commit, all modifications that were made are written to the log file in a single batch.

To discard any changes made during transaction operation, sp_destroy() function should be used. No nested transactions are supported.

There are no limit on a number of concurrent transactions. Any number of databases can be involved in a multi-statement transaction.

```
void *a = sp_getobject(env, "db.database_a");
void *b = sp_getobject(env, "db.database_b");

char key[] = "hello";
char value[] = "world";

/* begin a transaction */
void *transaction = sp_begin(env);

void *o = sp_document(a);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

o = sp_document(b);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

/* complete */
sp_commit(transaction);
```

Atransactional status should be checked (both for single and multi-statement):

```
int status = sp_commit(transaction);
switch (status) {
case -1: /* error */
case  0: /* ok */
case  1: /* rollback */
case  2: /* lock */
}
```

*Rollback* status means that transaction has been rollbacked by another concurrent transaction. *Lock* status means that transaction is not finished and waiting for concurrent transaction to complete. In that case commit should be retried later or transaction can be rollbacked. Any error happened during multi-statement transaction does not rollback a transaction.

# Deadlocks

Due to a nature of multi-statement transactions deadlocks are possible. Deadlocks are not automatically handled. Transaction object procedure **deadlock** can be used to check if the transaction is in deadlock.

When a deadlock happens, transactions stays in *Lock* state.

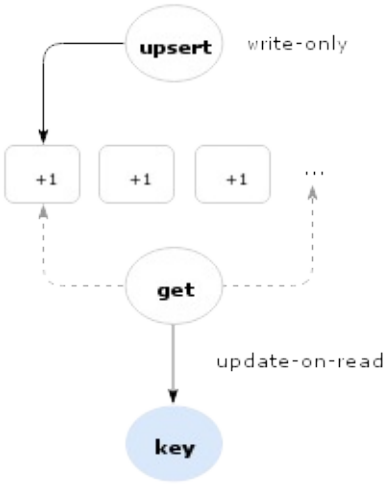
Example:

```
void *db = sp_getobject(env, "db.database");

void *a = sp_begin(env);
void *b = sp_begin(env);

uint32_t key = 7;
void *o = sp_document(db);
sp_setstring(o, "key", &key, sizeof(key));
sp_set(a, o);
key = 8;
o = sp_document(db);
sp_setstring(o, "key", &key, sizeof(key));
sp_set(b, o);
o = sp_document(db);
sp_setstring(o, "key", &key, sizeof(key));
sp_set(a, o);
key = 7;
o = sp_document(db);
sp_setstring(o, "key", &key, sizeof(key));
sp_set(b, o);

sp_commit(a) == 2; /* lock */
sp_commit(b) == 2; /* lock */

sp_getint(a, "deadlock") == 1;
sp_getint(b, "deadlock") == 1;

sp_destroy(a);
sp_getint(b, "deadlock") == 0;

sp_commit(b) == 0; /* ok */
```

# Asynchronous reads

Asynchronous operations are automatically scheduled when using sp_async(database) object instead of database one.
On complete: **scheduler.on_event** callback function is envoked.

# Upsert




Upsert is Update or Insert operation.

Sophia Upsert implementation allows to reduce Read-Modify-Write case to a single Write. Updates are applied by user-supplied callback **db.database_name.index.upsert** during data compaction or upon read request by sp_get() or sp_cursor().

To enable upsert command, a **db.database_name.index.upsert** and optionally **db.database_name.index.upsert_arg** must be set to callback function pointer.

Please take a look at sp_upsert() API for description and an example.

# Cursors

It is possible to do range queries using cursors.

To create a cursor the sp_cursor(), function should be used. Each call to sp_get() on cursor object makes an iteration step according to current iteration order. Second and futher sp_get() calls must using previously get object to continue iteration.

To set iteration order, cursor key object must be prepared by setting **order** field. To do prefix scan **prefix** field must be set using the same argument convention as for key. Supported orders: **>**, **>=**, **<**, **<=** both including or excluding search key. By default key order is set to >=.

If search key is not set, then maximum or minimum key is returned.

Example (traverse a database in increasing order):

```
void *cursor = sp_cursor(env);
void *o = sp_document(db);
sp_setstring(o, "order", ">=", 0);
while ((o = sp_get(c, o))) {
    char *key = sp_getstring(o, "key", NULL);
    char *value = sp_getstring(o, "value", NULL);
    printf("%s = %s\n", key, value);
}
sp_destroy(cursor);
```

Cursors are consistent. It is possible to do iteration and deletions or updates at the same time without any interference with query data or other transactions.

Cursor should be freed using the sp_destroy() function after usage.

# Sophia Environment

| name | type | description |
|---|---|---|
| sophia.version | string, ro | Get current Sophia version. |
| sophia.version_storage | string, ro | Get current Sophia storage version. |
| sophia.build | string, ro | Get git commit id of a current build. |
| sophia.error | string, ro | Get last error description. |
| sophia.path | string | Set current Sophia environment directory. |
| sophia.path_create | int | Fail if **sophia.path** directory is not exists. |
| sophia.recover | int | Recovery mode 1. on-phase (default) 2. - two-phase, 3. - three-phase. |

# Memory Control

Current memory limit is in bytes. This limit applies only to memory used for storing in-memory keys. This does not limit any memory used for node bufferization at the moment.

Memory pager is a part of Sophia that is used for memory allocation as a part of internal slab-allocator. Used for object allocations.

Please consider to read Architecture section about memory limits.

| name | type | description |
|---|---|---|
| memory.limit | int | Set current memory limit in bytes. |
| memory.used | int, ro | Get current memory usage in bytes. |
| memory.anticache | int | Set current memory limit for Anti-Cache mode. |
| memory.pager_pool_size | int, ro | Get current memory pager pool size. |
| memory.pager_page_size | int, ro | Get pager page size. |
| memory.pager_pools | int, ro | Get a number of allocated pager memory pools. |

# Scheduler

| name | type | description |
|---|---|---|
| scheduler.threads | int | Set a number of worker threads. |
| scheduler.id.trace | string, ro | Get a worker trace. |
| scheduler.zone | int, ro | Currently active compaction redzone. |
| scheduler.checkpoint_active | int,ro | Shows if checkpoint operation is in progress. |
| scheduler.checkpoint_lsn | int, ro | LSN of currently active checkpoint operation. |
| scheduler.checkpoint_lsn_last | int, ro | LSN of the last completed checkpoint operation. |
| scheduler.checkpoint | function | Force to start background checkpoint. Does not block. |
| scheduler.anticache_active | int,ro | Shows if anti-cache operation is in progress. |
| scheduler.anticache_asn | int, ro | ASN of currently active anti-cache operation. |
| scheduler.anticache_asn_last | int, ro | ASN of the last completed Anti-Cache operation. |
| scheduler.anticache | function | Force to start background anti-cache. Does not block. |
| scheduler.snapshot_active | int,ro | Shows if snapshot operation is in progress. |
| scheduler.snapshot_ssn | int, ro | SSN of currently active snapshot operation. |
| scheduler.snapshot_ssn_last | int, ro | SSN of the last completed snapshot operation. |
| scheduler.snapshot | function | Force to start background snapshot operation. Does not block. |
| scheduler.on_recover | pointer | Set recover log function. |
| scheduler.on_recover_arg | pointer | on_recover function arg. |
| scheduler.on_event | pointer | Set a callback which will be called on an async operation completion. |
| scheduler.on_event_arg | pointer | on_event function argument. |
| scheduler.event_on_backup | int | Generate async event on a backup completion. |
| scheduler.gc_active | function | Shows if gc operation is in progress. |
| scheduler.gc | function | Force to start background gc. Does not block. |
| scheduler.lru_active | function | Shows if lru operation is in progress. |
| scheduler.lru | function | Force to start background lru operation. Does not block. |
| scheduler.run | function | Run scheduler step in a current thread. For testing purposes. |

# Compaction

| name | type | description |
|---|---|---|
| compaction.redzone | int | To create a new redzone, write a percent value into *compaction* namespace. |
| compaction.redzone.mode | int | Set compaction mode. Mode 1: branch-less mode (strict 2-level storage), 2: checkpoint, 3: branch + compaction (default). |
| compaction.redzone.compact_wm | int | Compaction operation starts when a number of node branches reaches this watermark. Cant be less than two. |
| compaction.redzone.compact_mode | int | Set read-intensive or write-intensive compaction strategy mode. 0 - by number of branches, 1 - by temperature. |
| compaction.redzone.branch_prio | int | Priority of branch operation. Priority is measured by a maximum number of executing workers. |
| compaction.redzone.branch_wm | int | Branch operation starts when a size of in-memory key index reaches this watermark value. Measured in bytes. |
| compaction.redzone.branch_age | int | Branch operation automatically starts when it detects that a node in-memory key index has not been updated in a branch_age number of seconds. |
| compaction.redzone.branch_age_period | int | Scheduler starts scanning for aged node in-memory index every branch_age_period seconds. |
| compaction.redzone.branch_age_wm | int | This watermark value defines lower bound of in-memory index key size which is being checked during branch_age operation. Measured in bytes. |
| compaction.redzone.anticache_period | int | Check for anti-cache node election every anticache_period seconds. |
| compaction.redzone.backup_prio | int | Priority of backup operation. Priority is measured by a maximum number of executing workers. |
| compaction.redzone.gc_wm | int | Garbage collection starts when watermark value reaches a certain percent of duplicates. When this value reaches a compaction, operation is scheduled. |
| compaction.redzone.gc_db_prio | int | Priority of a database async close/drop operation. |
| compaction.redzone.gc_prio | int | Priority of gc operation. Priority is measured by a maximum number of executing workers. |
| compaction.redzone.gc_period | int | Check for a gc every gc_period seconds. |
| compaction.redzone.lru_prio | int | Priority of LRU operation. Priority is measured by a maximum number of executing workers. |
| compaction.redzone.lru_period | int | Run LRU scheduler every lru_period seconds. |
| compaction.redzone.async | int | Asynchronous thread work mode: 1 - reserve thread, 2 - do not reserve thread. |

# Performance

| name | type | description |
|---|---|---|
| performance.documents | int, ro | Number of currently allocated document. |
| performance.documents_used | int, ro | RAM used by allocated document. |
| performance.key | string, ro | Average key size. |
| performance.value | string, ro | Average value size. |
| performance.set | int, ro | Total number of Set operations. |
| performance.set_latency | string, ro | Average Set latency. |
| performance.delete | int, ro | Total number of Delete operations. |
| performance.delete_latency | string, ro | Average Delete latency. |
| performance.upsert | int, ro | Total number of Upsert operations. |
| performance.upsert_latency | string, ro | Average Upsert latency. |
| performance.get | int, ro | Total number of Get operations. |
| performance.get_latency | string, ro | Average Get latency. |
| performance.get_read_disk | string, ro | Average disk reads by Get operation. |
| performance.get_read_cache | string, ro | Average cache reads by Get operation. |
| performance.tx_active_rw | int, ro | Number of active RW transactions. |
| performance.tx_active_ro | int, ro | Number of active RO transactions. |
| performance.tx | int, ro | Total number of completed transactions. |
| performance.tx_rollback | int, ro | Total number of transaction rollbacks. |
| performance.tx_conflict | int, ro | Total number of transaction conflicts. |
| performance.tx_lock | int, ro | Total number of transaction locks. |
| performance.tx_latency | string, ro | Average transaction latency from begin till commit. |
| performance.tx_ops | string, ro | Average number of statements per transaction. |
| performance.tx_gc_queue | int, ro | Transaction manager GC queue size. |
| performance.cursor | int, ro | Total number of Cursor operations. |
| performance.cursor_latency | string, ro | Average Cursor latency. |
| performance.cursor_read_disk | string, ro | Average disk reads by Cursor operation. |
| performance.cursor_read_cache | string, ro | Average cache reads by Cursor operation. |
| performance.cursor_ops | string, ro | Average number of keys read by Cursor operation. |
| performance.req_queue | int, ro | Number of waiting request in async queue. |
| performance.req_ready | int, ro | Number of ready request in async queue. |
| performance.req_active | int, ro | Number of active request in async queue. |
| performance.reqs | int, ro | Current number of request in async queue. |

# Storage Engine Metrics

| name | type | description |
|---|---|---|
| metric.lsn | int | Current log sequential number. |
| metric.tsn | int | Current transaction sequential number. |
| metric.nsn | int | Current node sequential number. |
| metric.ssn | int | Current snapshot sequential number. |
| metric.asn | int | Current anticache sequential number. |
| metric.dsn | int | Current database sequential number. |
| metric.bsn | int | Current backup sequential number. |
| metric.lfsn | int | Current log file sequential number. |

# View

To create a view, new view name should be set to **view** configuration namespace.

| name | type | description |
| --- | --- | --- |
| view.name.lsn | int | Set or get view LSN number. |

# Write Ahead Log

| name | type | description |
|---|---|---|
| log.enable | int | Enable or disable transaction log. |
| log.path | string | Set folder for transaction log directory. If variable is not set, it will be automatically set as **sophia.path/log**. |
| log.sync | int | Sync transaction log on every commit. |
| log.rotate_wm | int | Create new log file after rotate_wm updates. |
| log.rotate_sync | int | Sync log file on every rotation. |
| log.rotate | function | Force to rotate log file. |
| log.gc | function | Force to garbage-collect log file pool. |
| log.files | int, ro | Number of log files in the pool. |

# Database

To create a database, new database name should be set to **db** control namespace. If no database exists, it will be created automatically.

Database can be created, opened or deleted before or after environment startup.

Database has following states: offline, online, recover, shutdown, malfunction. Database sets malfunction status if any unrecoverable error occurs.

| name | type | description |
|---|---|---|
| db.name.name | string, ro | Get database name |
| db.name.id | int | Database's sequential id number. This number is used in the transaction log for the database identification. |
| db.name.status | string, ro | Get database status. |
| db.name.storage | string | Set storage mode: anti-cache, cache, in-memory. |
| db.name.format | string | Set database format: kv, document. |
| db.name.amqf | int | enable or disable AMQ Filter. |
| db.name.path | string | Set folder to store database data. If variable is not set, it will be automatically set as **sophia.path/database_name**. |
| db.name.path_fail_on_exists | int | Produce error if path already exists. |
| db.name.path_fail_on_drop | int | Produce error on attempt to open 'dropped' database directory. |
| db.name.cache_mode | int | Mark this database as a cache. |
| db.name.cache | string | Set name of a cache database to use. |
| db.name.mmap | int | Enable or disable mmap mode. |
| db.name.sync | int | Sync node file on the branch creation or compaction completion. |
| db.name.node_preload | int | Preload whole node into memory for compaction. |
| db.name.node_size | int | Set a node file size in bytes. Node file can grow up to two times the size before the old node file is being split. |
| db.name.page_size | int | Set size of a page to use. |
| db.name.page_checksum | int | Check checksum during compaction. |
| db.name.compression_key | int | Enable or disable prefix (multi-part) compression. |
| db.name.compression | string | Specify compression driver. Supported: lz4, zstd, none (default). |
| db.name.compression_branch | string | Specify compression driver for branches. |
| db.name.lru | int | Enable LRU mode. |
| db.name.lru_step | int | Set LRU accuracy. |
| db.name.branch | function | Force branch creation. |
| db.name.compact | function | Force compaction. |
| db.name.compact_index | function | Force two-level compaction. |

| | | |
|---|---|---|
| db.name.index.memory_used | int, ro | Memory used by database for in-memory key indexes in bytes. |
| db.name.index.size | int, ro | Sum of nodes size in bytes (compressed). This is equal to the full database size. |
| db.name.index.size_uncompressed | int, ro | Full database size before the compression. |
| db.name.index.size_snapshot | int, ro | Snapshot file size. |
| db.name.index.size_amqf | int, ro | Total size used by AMQ Filter. |
| db.name.index.count | int, ro | Total number of keys stored in database. This includes transactional duplicates and not yet-merged duplicates. |
| db.name.index.count_dup | int, ro | Total number of transactional duplicates. |
| db.name.index.read_disk | int, ro | Number of disk reads since start. |
| db.name.index.read_cache | int, ro | Number of cache reads since start. |
| db.name.index.temperature_avg | int, ro | Average index temperature. |
| db.name.index.temperature_min | int, ro | Min index node temperature. |
| db.name.index.temperature_max | int, ro | Max index node temperature. |
| db.name.index.temperature_histogram | string, ro | Index temperature distribution histogram. |
| db.name.index.node_count | int, ro | Number of active nodes. |
| db.name.index.branch_count | int, ro | Total number of branches. |
| db.name.index.branch_avg | int, ro | Average number of branches per node. |
| db.name.index.branch_max | int, ro | Maximum number of branches per node. |
| db.name.index.branch_histogram | string, ro | Branch histogram distribution through all nodes. |
| db.name.index.page_count | int, ro | Total number of pages. |
| db.name.index.upsert | function | Set upsert callback function. |
| db.name.index.upsert_arg | function | Set upsert function argument. |
| db.name.index.key | string | Set index key type (string, u32, u64, u32_rev, u64_rev). See database section for details. |

# Backup

| name | type | description |
|---|---|---|
| backup.path | string | Set backup path. Each new backup will create a **backup.path/id** folder containing complete environment copy. |
| backup.run | function | Start background backup. Does not block. |
| backup.active | int | Shows if backup operation is in progress. |
| backup.last | int | Shows id of the last completed backup. |
| backup.last_complete | int | Shows if the last backup was successful. |

# Version 1.2

There have been major changes in storage architecture since version 1.1.

Version 1.1 defines strict 2-level storage model between in-memory and disk. It gives worst-case guarantee O(1) for any ordered key read in terms of disk access.

This approach had its limitations, since it was unable to efficiently maintain memory limit with required performance. Additionally there was a need for multi-threaded compaction.

Sophia has evolved in a way that expands original ideas. Architecture has been designed to efficiently work with memory and large amount of keys.

In fact, it became even simplier.

# Design

Sophia's main storage unit is a Node.

Each Node represents a single file with associated in-memory region index and two in-memory key indexes. Node file consists of Branches.

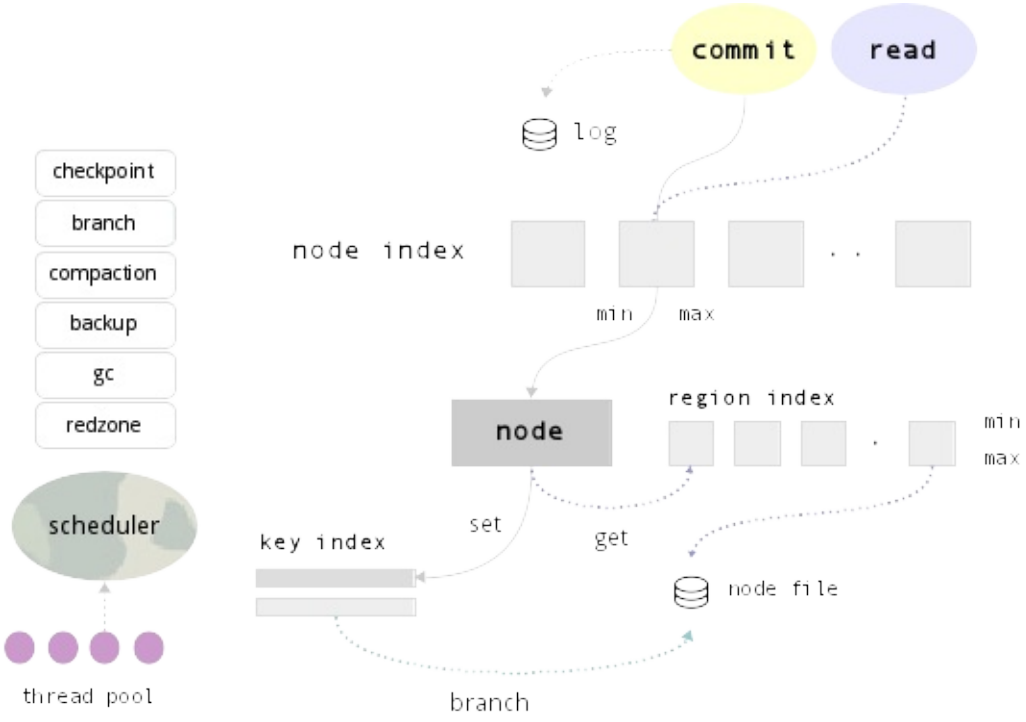
Each Branch consists of a set of sorted Regions and Region Index.

A single Region holds keys with values. It has the same semantical meaning as a B-Tree page, but organized in a different way. It does not have a tree structure or any internal page-to-page relationships and thus no meta-data overhead.

A Region Index is represented by an ordered range of regions with their min and max keys and on-disk reference. Regions never overlap.

A Key Index is very similar to LSM zero-level (memtable), but has a different key lifecycle. All modifications first get into the index and hold up until they are explicitly removed by the merger.

Before getting added to the in-memory Key Index, modifications are first written to the Write-Ahead Log.



# Lifecycle

Database lifecycle is organized in terms of two major operations: Branch and Compaction.

When a Node's in-memory Key Index size reaches a special watermark value or global memory limit, Branch operation is scheduled.

When some Node branch counter reaches a special watermark value, Compaction operation is scheduled.

**Branch operation**

1. rotate in-memory Key Index (use second one empty) (Node updates during Branch goes to second index)
2. create new Regions and Region Index from Key Index
3. create new Node Branch
4. sequentially write Branch to the end of Node file
5. sync
6. free index and rotate

**Compaction operation**

1. sequentially read Node file into memory
2. create iterator for each Branch
3. merge and split key-stream:
    i. create one or more Nodes
    ii. delete Node
4. sequentially write new Node or Nodes
5. sync
6. redistribute online updates between new Nodes
7. remove old Node
8. rename new Node or Nodes when completed

# Optimization Game

All background operations are planned by special scheduler.

There is a game between available memory, a number of Branches and Search times.

Each additional branch says that there is a possible additional disk access during the search. During the search, only branch regions that have min >= key <= max are examined. In the same time, it is unable to maintain memory limits without branching, because compaction times are greater than possible rate of incoming data.

Sophia is designed to be read optimized. There is a high possibility that latest created Branches (hot data) are stored in the file system cache. Scheduler is aware about nodes which have largest in-memory Key Index and biggest number of Branches. These are processed first.

Additionally all operations are planned taking current system state in account, like memory usage statistics, current load profiler (redzone), operations priorities, checkpoint, backup, etc.

Sophia compaction is multi-threaded. Each worker (thread) requests scheduler for a new task. Basic unit of a background task is an operation on a single Node.

Sophia is designed to efficiently utilize available memory. If there is more memory available, then branch/compaction operations become more infrequent and system becomes more disk-efficient. Best performance can be obtained with no memory limit set. Sophia is Hard-Drive (and Flash) friendly, since all operations are delayed and executed in large sequential reads and writes, without overwrite.

# Garbage Collection

Garbage collection (MVCC related) is executed automatically by Compaction task.

Also, scheduler periodically checks if there are any nodes which have large percentage of transactional versions (duplicates) stored per node.

# Algorithmic Complexity

Sophia has following algorithmic complexity (in terms of disk access):

**set** worst case is O(1) write-ahead-log append-only key write + in-memory node index search + in-memory index insert

**delete** worst case is O(1) (delete operation is equal to set)

**get** worst case is amortized O(max_branch_count_per_node) random region read from a single node file, which itself does in-memory key index search + in-memory region search

**range** worst case of full database traversal is amortized O(total_region_count) + in-memory key-index searches for each node

# Version 1.1

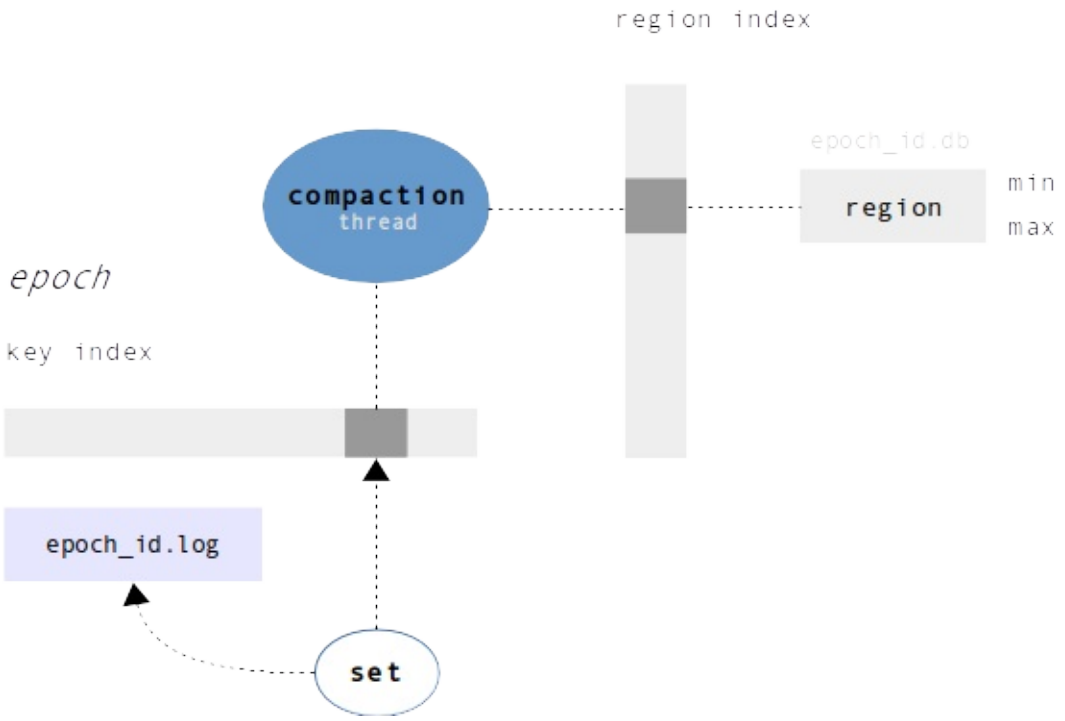
Original Sophia's architecture combines a region in-memory index with an in-memory key index.

A region index is represented by an ordered range of regions with their min and max keys and latest on-disk reference. Regions never overlap.

These regions have the same semantical meaning as the B-Tree pages, but designed differently. They do not have a tree structure or any internal page-to-page relationships, thus no meta-data overhead (specifically to append-only B-Tree).

A single region on-disk holds keys with values. As a B-tree page, region has its maximum key count. Regions are uniquely identified by region id number that makes them trackable in future.

Key index is very similar to LSM zero-level (memtable), but has a different key lifecycle. All modifications first get inserted into the index and then hold up until they are explicitly removed by merger.




# Lifecycle

The database update lifecycle is organized in terms of epochs. Epoch lifetime is set in terms of key updates. When the update counter reaches an epoch's watermark number then the Rotation event happen.

Each epoch, depending on its state, is associated with a single log file or database file. Before getting added to the in-memory index, modifications are first written to the epoch's write-ahead log.

On each rotation event:

1. current epoch, so called 'live', is marked as 'transfer' which leads to a new 'live' epoch creation (new log file)
2. create new in-memory key index and swap it with current one
3. merger thread is being woken up

The merger thread is the core part that is responsible for region merging and garbage collecting of the old regions and older epochs. On the wakeup, the merger thread iterates through list of epochs marked as 'transfer' and starts the merge procedure.

The merge procedure has the following steps:

1. create new database file for the latest 'transfer' epoch
2. fetch any keys from in-memory index that associated with a single destination region
3. start the merge for each fetched key and origin region, then write a new region to the database file
4. on each completed region (current merged key count is less or equal to max region key count):
    i. allocate new split region for region index, set min and max
    ii. first region always has id of origin destination region
    iii. link region and schedule for future commit
5. on origin region update completion:
    i. update destination region index file reference to the current epoch and insert split regions
    ii. remove keys from key index
6. start step (2) until there is no updates left
7. start garbage collector
8. sync database with a disk drive, then, if everything went well, remove all 'transfer' epochs (log files) and gc'ed databases
9. free index memory

# Garbage Collection

The garbage collector has a simple design.

All that you need is to track an epoch's total region count and the count of transfered regions during merge procedure. Thus, if some older epoch database has fewer than 70% (or any other changeable factor) live regions, they just get copied to the current epoch database file while the old one is being deleted.

On database recovery, Sophia tracks and builds an index of pages from the earliest epochs (biggest numbers) down to the oldest. Log files then are being replayed and epochs become marked as 'transfer'.

# Algorithmic Complexity

Sophia has been evaluated as having the following complexity (in terms of disk accesses):

**set** worst case is O(1) append-only key write + in-memory index insert

**delete** worst case is O(1) (delete operation is equal to set)

**get** worst case is O(1) random region read, which itself does amortized O(log region_key_count) key compares + in-memory key index search + in-memory region search

**range** range queries are very fast due to the fact that each iteration needs to compare no more than two keys without a search, and access through mmaped database. Roughly complexity can be equally evaluated as sequential reading of the mmaped file.

### NAME

sp_env - create a new environment handle

### SYNOPSIS

```
#include <sophia.h>

void *sp_env(void);
```

### DESCRIPTION

The sp_env() function allocates new Sophia environment object.

The object is intended for usage by sp_open() and must be configured first. After using, an object should be freed by sp_destroy().

Please take a look at Configuration, and Database administration sections.

Common workflow is described here.

### EXAMPLE

```
void *env = sp_env();
sp_setstring(env, "sophia.path", "./storage", 0);
sp_setstring(env, "db", "test", 0);
sp_open(env);
void *db = sp_getobject(env, "db.test");
/* do transactions */
sp_destroy(env);
```

### RETURN VALUE

On success, sp_env() allocates new environment object pointer. On error, it returns NULL.

### SEE ALSO

Sophia API

### NAME

sp_document - create a document object

### SYNOPSIS

```
#include <sophia.h>

void *sp_document(void *object);
```

### DESCRIPTION

sp_document(**database**): create new document for a transaction on a selected database.

The sp_document() function returns an object which is intended to be used in by any CRUD operations. Document might contain a key-value pair with any additional metadata.

### EXAMPLE

```
void *o = sp_document(db);
sp_setstring(o, "key", "hello", 0);
sp_setstring(o, "value", "world", 0));
sp_set(db, o);
```

### RETURN VALUE

On success, sp_document() returns an object pointer. On error, it returns NULL.

### SEE ALSO

Sophia API

### NAME

sp_setstring, sp_getstring, sp_setint, sp_getint, sp_getobject - set or get configuration options

### SYNOPSIS

```
#include <sophia.h>

int      sp_setstring(void *object, const char *path, const void *ptr, int size);
void    *sp_getstring(void *object, const char *path, int *size);
int      sp_setint(void *object, const char *path, int64_t value);
int64_t  sp_getint(void *object, const char *path);
void    *sp_getobject(void *object, const char *path);
```

### DESCRIPTION

For additional information take a look at the Configuration section.

### EXAMPLE

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_setstring() returns 0. On error, it returns -1.

On success, sp_getstring() returns string pointer. On error or if the variable is not set, it returns NULL.

All pointers returned by sp_getstring() must be freed using free(3) function. Exception is sp_document() object and configuration cursor document.

On success, sp_setint() returns 0. On error, it returns -1. On success, sp_getint() returns a numeric value. On error, it returns -1.

On success, sp_getobject() returns an object pointer. On error or if the variable is not set, it returns NULL.

The database object returned by sp_getobject() increments its reference counter, sp_destroy() can be used to decrement it. This should be considered for online database close/drop cases.

### SEE ALSO

Sophia API

### NAME

sp_setstring, sp_getstring, sp_setint, sp_getint, sp_getobject - set or get configuration options

### SYNOPSIS

```
#include <sophia.h>

int      sp_setstring(void *object, const char *path, const void *ptr, int size);
void    *sp_getstring(void *object, const char *path, int *size);
int      sp_setint(void *object, const char *path, int64_t value);
int64_t  sp_getint(void *object, const char *path);
void    *sp_getobject(void *object, const char *path);
```

### DESCRIPTION

For additional information take a look at the Configuration section.

### EXAMPLE

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_setstring() returns 0. On error, it returns -1.

On success, sp_getstring() returns string pointer. On error or if the variable is not set, it returns NULL.

All pointers returned by sp_getstring() must be freed using free(3) function. Exception is sp_document() object and configuration cursor document.

On success, sp_setint() returns 0. On error, it returns -1. On success, sp_getint() returns a numeric value. On error, it returns -1.

On success, sp_getobject() returns an object pointer. On error or if the variable is not set, it returns NULL.

The database object returned by sp_getobject() increments its reference counter, sp_destroy() can be used to decrement it. This should be considered for online database close/drop cases.

### SEE ALSO

Sophia API

### NAME

sp_setstring, sp_getstring, sp_setint, sp_getint, sp_getobject - set or get configuration options

### SYNOPSIS

```
#include <sophia.h>

int      sp_setstring(void *object, const char *path, const void *ptr, int size);
void    *sp_getstring(void *object, const char *path, int *size);
int      sp_setint(void *object, const char *path, int64_t value);
int64_t  sp_getint(void *object, const char *path);
void    *sp_getobject(void *object, const char *path);
```

### DESCRIPTION

For additional information take a look at the Configuration section.

### EXAMPLE

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_setstring() returns 0. On error, it returns -1.

On success, sp_getstring() returns string pointer. On error or if the variable is not set, it returns NULL.

All pointers returned by sp_getstring() must be freed using free(3) function. Exception is sp_document() object and configuration cursor document.

On success, sp_setint() returns 0. On error, it returns -1. On success, sp_getint() returns a numeric value. On error, it returns -1.

On success, sp_getobject() returns an object pointer. On error or if the variable is not set, it returns NULL.

The database object returned by sp_getobject() increments its reference counter, sp_destroy() can be used to decrement it. This should be considered for online database close/drop cases.

### SEE ALSO

Sophia API

### NAME

sp_setstring, sp_getstring, sp_setint, sp_getint, sp_getobject - set or get configuration options

### SYNOPSIS

```
#include <sophia.h>

int      sp_setstring(void *object, const char *path, const void *ptr, int size);
void    *sp_getstring(void *object, const char *path, int *size);
int      sp_setint(void *object, const char *path, int64_t value);
int64_t  sp_getint(void *object, const char *path);
void    *sp_getobject(void *object, const char *path);
```

### DESCRIPTION

For additional information take a look at the Configuration section.

### EXAMPLE

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_setstring() returns 0. On error, it returns -1.

On success, sp_getstring() returns string pointer. On error or if the variable is not set, it returns NULL.

All pointers returned by sp_getstring() must be freed using free(3) function. Exception is sp_document() object and configuration cursor document.

On success, sp_setint() returns 0. On error, it returns -1. On success, sp_getint() returns a numeric value. On error, it returns -1.

On success, sp_getobject() returns an object pointer. On error or if the variable is not set, it returns NULL.

The database object returned by sp_getobject() increments its reference counter, sp_destroy() can be used to decrement it. This should be considered for online database close/drop cases.

### SEE ALSO

Sophia API

### NAME

sp_setstring, sp_getstring, sp_setint, sp_getint, sp_getobject - set or get configuration options

### SYNOPSIS

```
#include <sophia.h>

int      sp_setstring(void *object, const char *path, const void *ptr, int size);
void    *sp_getstring(void *object, const char *path, int *size);
int      sp_setint(void *object, const char *path, int64_t value);
int64_t  sp_getint(void *object, const char *path);
void    *sp_getobject(void *object, const char *path);
```

### DESCRIPTION

For additional information take a look at the Configuration section.

### EXAMPLE

```
void *env = sp_env()
sp_setstring(env, "sophia.path", "./sophia", 0);
sp_open(env);
```

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_setstring() returns 0. On error, it returns -1.

On success, sp_getstring() returns string pointer. On error or if the variable is not set, it returns NULL.

All pointers returned by sp_getstring() must be freed using free(3) function. Exception is sp_document() object and configuration cursor document.

On success, sp_setint() returns 0. On error, it returns -1. On success, sp_getint() returns a numeric value. On error, it returns -1.

On success, sp_getobject() returns an object pointer. On error or if the variable is not set, it returns NULL.

The database object returned by sp_getobject() increments its reference counter, sp_destroy() can be used to decrement it. This should be considered for online database close/drop cases.

### SEE ALSO

Sophia API

### NAME

sp_open - open or create

### SYNOPSIS

```
#include <sophia.h>

int sp_open(void *object);
```

### DESCRIPTION

sp_open(**env**): create environment, open or create pre-defined databases.

sp_open(**database**): create or open database.

Please take a look at Configuration, and Database administration sections.

Common workflow is described here.

### EXAMPLE

```
void *env = sp_env();
sp_setstring(env, "sophia.path", "./storage", 0);
sp_setstring(env, "db", "test", 0);
sp_open(env);
void *db = sp_getobject(env, "db.test");
/* do transactions */
sp_destroy(env);
```

### RETURN VALUE

On success, sp_open() returns 0. On error, it returns -1.

### SEE ALSO

Sophia API

### NAME

sp_destroy - free or destroy an object

### SYNOPSIS

```
#include <sophia.h>

int sp_destroy(void *object);
```

### DESCRIPTION

The sp_destroy() function is used to free memory allocated by any Sophia object.

### EXAMPLE

```
void *o = sp_document(db);
void *result = sp_get(db, o);
if (result)
    sp_destroy(result);
```

### RETURN VALUE

On success, sp_destroy() returns 0. On error, it returns -1.

### SEE ALSO

Sophia API

### NAME

sp_error - check error status

### SYNOPSIS

```
#include <sophia.h>

int sp_error(void *env);
```

### DESCRIPTION

sp_error(**env**): check if there any error leads to the shutdown.

Additionally, if any sophia error description can be accessed through **sophia.error** field.

```
int error_size;
char *error = sp_getstring(env, "sophia.error", &error_size);
if (error) {
    printf("error: %s\n", error);
    free(error);
}
```

### RETURN VALUE

Returns 1 or 0.

### SEE ALSO

Sophia API

**NAME**

sp_poll - get a completed asynchronous request

**SYNOPSIS**

```
#include <sophia.h>

void *sp_poll(void *env);
```

**DESCRIPTION**

sp_poll(**env**)

For additional information take a look at Asynchronous read section.

**RETURN VALUE**

On success, sp_poll() returns a document handle. If there are no completed requests, returns NULL.

**SEE ALSO**

Sophia API

### NAME

sp_drop - schedule an database drop or an object deletion

### SYNOPSIS

```
#include <sophia.h>

int sp_drop(void *object);
```

### DESCRIPTION

sp_drop(**database**) Schedule database drop.

sp_drop(**view**) Drop a view.

### EXAMPLE

```
void *db = sp_getobject(env, "db.test");
sp_drop(db);
```

### RETURN VALUE

On success, sp_drop() returns 0. On error, it returns -1.

### SEE ALSO

Sophia API

### NAME

sp_set - insert or replace operation

### SYNOPSIS

```
#include <sophia.h>

int sp_set(void *object, void *document);
```

### DESCRIPTION

sp_set(**database**, document): do a single-statement transaction.

sp_set(**transaction**, document): do a key update as a part of multi-statement transaction.

As a part of a transactional statement a key-value document must be prepared using sp_document() method. First argument of sp_document() method must be an database object.

Object must be prepared by setting **key** and **value** fields, where value is optional. It is important that while setting **key** and **value** fields, only pointers are copied. Real data copies only during first operation.

For additional information take a look at sp_document(), sp_begin() and Transactions.

### EXAMPLE

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", "hello world", 0));
sp_set(db, o);
```

### RETURN VALUE

On success, sp_set() returns 0. On error, it returns -1.

Database object commit: (1) rollback or (2) lock.

### SEE ALSO

Sophia API

**NAME**

sp_upsert - common get operation

**SYNOPSIS**

```
#include <sophia.h>

void *sp_upsert(void *object, void *document);

typedef int (*upsert_callback)(char **result,
                               char **key, int *key_size, int key_count,
                               char *src, int src_size,
                               char *upsert, int upsert_size,
                               void *arg);
```

**DESCRIPTION**

sp_upsert(**database**, document): do a single-statement transaction.

sp_upsert(**transaction**, document): do a key update as a part of multi-statement transaction.

As a part of a transactional statement a key-value document must be prepared using sp_document() method. First argument of sp_document() method must be an database object.

Object must be prepared by setting **key** and **value** fields. It is important that while setting **key** and **value** fields, only pointers are copied. Real data copies only during first operation.

Value field should contain user-supplied data, which should be enough to implement custom update or insert logic.

To enable upsert command, a **db.database_name.index.upsert** and optionally **db.database_name.index.upsert_arg** must be set to callback function pointer.

For additional information take a look at sp_document(), sp_begin() and Transactions and Upsert sections.

**EXAMPLE**

See Sophia repository upsert example.

**RETURN VALUE**

On success, sp_set() returns 0. On error, it returns -1.

Database object commit: (1) rollback or (2) lock.

**SEE ALSO**

Sophia API

**NAME**

sp_delete - delete operation

**SYNOPSIS**

```
#include <sophia.h>

int sp_delete(void *object, void *document);
```

**DESCRIPTION**

sp_delete(**database**, document): do a single-statement transaction.

sp_delete(**transaction**, document): do a key deletion as a part of multi-statement transaction.

As a part of a transactional statement a key-value document must be prepared using sp_document() method. First argument of sp_document() method must be an database object.

Object must be prepared by setting **key** fields. Value is not used for delete operation. It is important that while setting **key** fields, only pointers are copied. Real data copies only during first operation.

For additional information take a look at sp_document(), sp_begin() and Transactions.

**EXAMPLE**

```
char key[] = "key";
void *o = sp_document(db);
sp_setstring(o, "key", key, sizeof(key));
sp_delete(db, o);
```

**RETURN VALUE**

On success, sp_delete() returns 0. On error, it returns -1.

Database object commit: (1) rollback or (2) lock.

**SEE ALSO**

Sophia API

### NAME

sp_get - common get operation

### SYNOPSIS

```
#include <sophia.h>

void *sp_get(void *object, void *document);
```

### DESCRIPTION

sp_get(**database**, document): do a single-statement transaction read.

sp_get(**transaction**, document): do a key search as a part of multi-statement transaction visibility.

sp_get() method returns an document that is semantically equal to sp_document(), but is read-only.

For additional information take a look at sp_begin() and Transactions.

### EXAMPLE

```
void *o = sp_document(db);
sp_set(o, "key", "hello", 0);
void *result = sp_get(db, o);
if (result) {
    int valuesize;
    char *value = sp_getstring(result, "value", &valuesize);
    printf("%s\n", value);
    sp_destroy(result);
}
```

### RETURN VALUE

On success, sp_get() returns a document handle. If an object is not found, returns NULL. On error, it returns NULL.

### SEE ALSO

Sophia API

### NAME

sp_cursor - common cursor operation

### SYNOPSIS

```
#include <sophia.h>

void *sp_cursor(void *env);
```

### DESCRIPTION

sp_cursor(**env**): create a cursor ready to be used with any database.

For additional information take a look at Cursor section.

### EXAMPLE

```
void *cursor = sp_cursor(env);
void *o = sp_document(db);
sp_setstring(o, "order", ">=", 0);
while ((o = sp_get(c, o))) {
    char *key = sp_getstring(o, "key", NULL);
    char *value = sp_getstring(o, "value", NULL);
    printf("%s = %s\n", key, value);
}
sp_destroy(cursor);
```

### RETURN VALUE

On success, sp_cursor() returns cursor object handle. On error, it returns NULL.

### SEE ALSO

Sophia API

### NAME

sp_begin - start a multi-statement transaction

### SYNOPSIS

```
#include <sophia.h>

void *sp_begin(void *env);
```

### DESCRIPTION

sp_begin(**env**): create a transaction

During transaction, all updates are not written to the database files until a sp_commit() is called. All updates that were made during transaction are available through sp_get() or by using cursor.

The sp_destroy() function is used to discard changes of a multi-statement transaction. All modifications that were made during the transaction are not written to the log file.

No nested transactions are supported.

For additional information take a look at Transactions and Deadlock sections.

### EXAMPLE

```
void *a = sp_getobject(env, "db.database_a");
void *b = sp_getobject(env, "db.database_b");

char key[] = "hello";
char value[] = "world";

/* begin a transaction */
void *transaction = sp_begin(env);

void *o = sp_document(a);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

o = sp_document(b);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

/* complete */
sp_commit(transaction);
```

### RETURN VALUE

On success, sp_begin() returns transaction object handle. On error, it returns NULL.

### SEE ALSO

Sophia API

### NAME

sp_commit - commit a multi-statement transaction

### SYNOPSIS

```
#include <sophia.h>

int sp_commit(void *transaction);
```

### DESCRIPTION

sp_commit(**transaction**): commit a transaction

The sp_commit() function is used to apply changes of a multi-statement transaction. All modifications that were made during the transaction are written to the log file in a single batch.

If commit failed, transaction modifications are discarded.

For additional information take a look at Transactions and Deadlock sections.

### EXAMPLE

```
void *a = sp_getobject(env, "db.database_a");
void *b = sp_getobject(env, "db.database_b");

char key[] = "hello";
char value[] = "world";

/* begin a transaction */
void *transaction = sp_begin(env);

void *o = sp_document(a);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

o = sp_document(b);
sp_setstring(o, "key", key, sizeof(key));
sp_setstring(o, "value", value, sizeof(value));
sp_set(transaction, o);

/* complete */
sp_commit(transaction);
```

### RETURN VALUE

On success, sp_commit() returns 0. On error, it returns -1. On rollback 1 is returned, 2 on lock.

### SEE ALSO

Sophia API