The Ferret Toolkit

Reference Manual

The Ferret Team
Department of Computer Science, Princeton University

Short Contents

Table of Contents

1 Introduction

2 Installation

3 The Ferret Library

3.1 Using the Ferret Library

After successful installation, you only need to put the following line to your code before the first reference to the Ferret Library data types or routines.

```
#include <cass.h>
```

To link against the Ferret Library, add the option `-lcass` to the command line of `ld` or `gcc`.

3.2 Portability

For portability purpose, the following primitive types are used in the Ferret Library, so that the compile to the same size on all machines.

```
int32_t
uint32_t
uint64_t
uchar
```

The database files are portable between 32-bit and 64-bit machines, but not between machines of different endians for now. To make it portable between different endians, only the file IO module (`cass_file.c`) needs to be modified.

3.3 Error Handling

Most of the Ferret Library routines return 0 for success and a non-zero error code if some error happens. Following is a list of possible error codes.

```
CASS_ERR_OUTOFMEM
CASS_ERR_MALFORMATVEC
CASS_ERR_PARAMETER
CASS_ERR_IO
CASS_ERR_CORRUPTED
```

The following function converts an error code into a string for display.

Prototype: `const char* cass_strerror (int err);`

3.4 Utility Routines

This section documents the general utility data structures and routines that are not perticularly related to nearest neighbor search, but will make program easier. You do not need to initialize the library to use the routines documented in this section.

3.4.1 Dynamic arrrays

Maintaining a dynamic (growable) array is always a pain in the plain C language. The Ferret Library provides a set of macros to ease the programmer's life. A dynamic array of type *T* is declared as the following struct:

```
struct {
        cass_size_t inc;
        cass_size_t size;
        cass_size_t len;
```

```
        T *data
} array;
```

where `len` is the number of elements actually in the array and `size` is the size of the memory allocated, pointed to by `data`. When the array needs to grow, `size` is incremented by `inc`, and the memory is reallocated. All `inc`, `size` and `len` are numbers of elements instead of actual bytes. To access the `i`-th element of *`array`*, use *`array`*`.data[i]`.

Any structure declared following the above template can be handled by the macros documented in this section, despite the type of *`T`* and the actual name of the struct. Using the array declaration macro `ARRAY_TYPE`, the declaration of an array can be as easy as

```
ARRAY_TYPE(T) array;
```

The following small example shows the operations of an array of `char`.

```
ARRAY_TYPE(char) s;
ARRAY_INIT(s);

ARRAY_APPEND(s, 'A');
ARRAY_APPEND(s, 'B');
ARRAY_GET(s, 0);        /* should be 'A' */
ARRAY_LEN(s);           /* should be 2 */

/* enumerate the array */
ARRAY_BEGIN_FOREACH(s, c)
{
        putchar(c);
} ARRAY_END_FOREACH(s, c);

ARRAY_CLEANUP(array);
```

The following items are to be noticed.

- All macros accepts the array variable instead of pointer to the variable.
- The type of array elements can be obtained by the GCC extension `typeof` *`array`*`.data[0]`
- The details of the array struct are considered exposed, and any changes made to the struct are assumed safe, so long as the following holds:

```
   len <= size
&& inc > 0
&& (size == 0 || data != NULL)
```

- The array struct can safely contain other elements and they will not be touched by the macros. To add extra elements to the struct, the user need to explicitly declare the struct, the `ARRAY_TYPE` macro will not help.
- The user is responsible for managing the dynamically allocated memory pointed to by the array element, should they be pointers.

3.4.1.1 Declaration

Macro: `ARRAY_TYPE(type)` *`array`*`;`

`array` `= ARRAY_WRAPPER(data, len);`

Description:

`ARRAY_TYPE` expands to the type declaration of a struct for a dynamic array of the given type. `ARRAY_WRAPPER` makes an array variable out of a chunk of

memory allocated by `malloc`, specified by the pointer to the first element, `data`, and the number of element, `len`.

3.4.1.2 Initialization

Macro: `ARRAY_INIT(array)`

`ARRAY_INIT_SIZE(array, initial_size)`

Description:

Initialize an array. The latter set the size of array to `initial_size` and allocates the memory of this size.

3.4.1.3 Cleanup

Macro: `ARRAY_CLEANUP(array)`

Description:

Release the dynamically allocated memory of the array.

3.4.1.4 Field access

Macro: `ARRAY_INC(array)`

`ARRAY_SET_INC(array, inc)`

`ARRAY_LEN(array)`

`ARRAY_RAW_SIZE(array)`

`ARRAY_SIZE(array)`

`ARRAY_EXPAND(array, new_size)`

Description:

These macros access the various fields of the array struct. The macro `ARRAY_RAW_SIZE` returns the raw size of data in bytes actually present in the array, which is equal to `array.len * sizeof(array.data[0])`.

`ARRAY_EXPAND` will grow the array if necessary so that the size of the array is at least `new_size`.

3.4.1.5 Element access

Macro: `ARRAY_GET(array, i)`

`ARRAY_SET(array, i, elem)`

`ARRAY_APPEND(array, elem)`

`ARRAY_APPEND_UNSAFE(array, elem)`

`ARRAY_TRUNC(array) /* same as array.len = 0 */`

`ARRAY_MERGE(array, array2)`

`ARRAY_MERGE_RAW(array, data, len)`

Description:

For `ARRAY_GET` and `ARRAY_SET`, you need to ensure that `i < array.len`.

`ARRAY_APPEND` will automatically grow the array when space is not enough, but `ARRAY_APPEND_UNSAFE` will not. The latter is faster though, and can be used when total size of the array is known and fixed at initialization.

`ARRAY_MERGE` appends all the elements in `array2` to `array`, growing (array) when necessary. `ARRAY_MERGE_RAW` is similar, but the second array is given by the pointer to the first element of the array and the number of elements.

3.4.1.6 Element enumeration

Macro:

```
ARRAY_BEGIN_FOREACH(array, cursor) {

      ...your code...

} ARRAY_END_FOREACH(array, cursor);


ARRAY_BEGIN_FOREACH_P(array, cursor_p) {

      ...your code...

} ARRAY_END_FOREACH_P(array, cursor_p);
```

Description:

These two sets of macros enumerate the elements in an array. `cursor(_p)` does not need to be declared outside the block (actually, any declaration of `cursor(_p)` outside the block will be overridden). The former pair of macros passes the element in value and the latter in reference, which can reduce the overhead of copying the elements when they are large. You can use any name for `cursor(_p)` so long as they match in the `..._BEGIN_...` and `..._END_...` macros.

3.4.2 Heaps

The Ferret Library provides binary heap support on top of the dynamic array routines described in the previous section. A heap is just a dynamic array, except that the elements in the array follow certain order. The heap routines described below directly operate on array structs.

Prototype:

```
HEAP_EMPTY(heap) /*boolean, whether the heap is empty */
HEAP_HEAD(heap) /*the smallest element in the heap, in value */
HEAP_ENQUEUE(heap, elem, ge)
HEAP_DEQUEUE(heap, ge) /* does not return the element */
HEAP_ENQUEUE_UPDATE(heap, elem, ge, update)
HEAP_DEQUEUE_UPDATE(heap, elem, ge, update)
```

Description:

Both `HEAP_ENQUEUE` and `HEAP_DEQUEUE` require an extra parameter `ge`, the comparator, which can be either a function pointer or a macro. For any two variable `a` and `b` of the type of heap element, `ge(&a,&b)` should return 1 if `a >= b` and 0 if not.

Sometimes, the elements of the heap need to known their own index in the heap. The situation is complicated because the enqueue and dequeue operations need to reorder some of the heap elements. The macros `HEAP_ENQUEUE_UPDATE`

and `HEAP_DEQUEUE_UPDATE` allow the user to track the offset of heap elements. The extra parameter `update` and be a function pointer or a macro, and for each element `a` which is rellocated to the new offset `i`, `update(&a, i)` will be invoked.

For `HEAP_DEQUEUE(_UPDATE)`, the head of the heap is thrown away silently. You need to call `HEAP_HEAD` in advance if you need to keep the dequeued element.

All the heap macros described here assume the elements of the array are in binary heap order. You can safely access the heap data by other array macros in read-only manner, but updates other than `ARRAY_EXPAND` are not advised because they may destroy the heap order.

3.4.3 Maintaining top-K elements

Maintaining a list of the best K structs with respect to certain field is a common job and the Ferret Toolkit provides a set of macros to do this efficiently.

The macros documented below works on an array of `K` structs with a comparable field `key`.

Prototype: `TOPK_INIT (array, key, K, init)`

Description:

Initialize the `key` field of each of the `K` element to value `init`. For `K` maximal values, `init` should be minimal value possible, and for the minimal values the opposite.

Prototype: `TOPK_INSERT_MIN (array, key, K, elem)`

Description:

Keep the `K` minimal element in `array` with regard to the field `key`. If `elem` is smaller than the existing largest element in `array`, it replace that largest element.

Prototype: `TOPK_INSERT_MAX (array, key, K, elem)`

Description:

Similar to `TOPK_INSERT_MIN`, except for it keeps the `K` maximal elements.

Prototype: `TOPK_SORT_MIN (array, key, K)`

Description:

The above two macros use heap for high performance and the result is that the result array is not sorted according to `key`. This macro sorts the array in descending order according to `key`. Note that this macro can only sort arrays prepared with `TOPK_INSERT_MIN`.

Prototype: `TOPK_INSERT_MIN_UNIQ (array, key, K, elem)`

Description:

Similar to `TOPK_INSERT_MIN`, except it assures that every one in `array` has a unique `index` field. For now, the name of the field `index` is not customizable. This is to be fixed fixed in later versions of the toolkit. To keep the uniqueness, the naive array insertion instead of heap is used, and the result is sorted.

Following is a small example. Given the query point id $codeq and the set of data point id `data[]`, we want to get the `K` points in `data[]` that is closest to `q`.

```
#define K 10

typedef struct {
        float key;
        int index;
} foo_t;

foo_t min[K], foo;

TOPK_INIT(min, key, K, MAXFLOAT);

for (i = 0; i < data_len; i++)
{
       foo.key = dist(q, data[i]);
       foo.index = i;
       TOPK_INSERT_MIN(min, key, K, foo);
}

/* sort the result*/
TOPK_SORT_MIN(min, key, K);
```

3.4.4 Portable file IO

Prototype:

```
typedef struct ... CASS_FILE;
CASS_FILE *cass_open (const char *path, const char *mode);
void cass_close (CASS_FILE *);
/* type can be one of { int32, uint32, uint64, size, float, double,
char */
int cass_read_type (type *, size_t nmemb, CASS_FILE *);
int cass_write_type (const type *, size_t nmemb, CASS_FILE *);
char *cass_read_pchar (CASS_FILE *);
int cass_write_pchar (const char *, CASS_FILE *);
```

Description:

These routines intend to be a portable way of doing I/O, so that the file produced from a machine of one architecture can be read by a machine of another architecture. For now, the portability is not realized and these routines are just a wrapper of stdio. `CASS_FILE` is same as `FILE` for now.

3.4.5 Timer

Prototype:

```
typedef struct ... stimer_t;
void stimer_tick (stimer_t *timer);
float stimer_tuck (stimer_t *timer, const char *msg);
```

Description:

These routines are used to measure the wall time used by certain piece of code. `stimer_tick` initialize the `timer` struct and record the start time, `stimer_tuck` stop timing and returns the elapsed time. If `msg` is not `NULL`, the elapsed time is printed to `stdout` together with `msg`.

3.4.6 Replacement of `malloc`, `calloc` and `realloc`

Prototype: `type * type_alloc (type);`

`type * type_calloc (type, cass_size_t len);`

`type * type_realloc (type, type *ptr, cass_size_t len);`

Description:

These macros are just wrappers of `malloc`, `calloc` and `realloc`. They accept `type` instead of `sizeof(type)`, and return pointer of type `type *` instead of `void *`.

3.4.7 Vectors and Matrices

Two dimensional array, or matrix, of size `M*N` are stored in memory in the following way. First, a chunk of memory to hold `M*N` elements are allocated. Then a chunk of memory to hold `M` pointer to the rows are allocated, and initialized to the start memory address of each row. `matrix3` is the 3-dimensional array.

Prototype:

`type ** type_matrix_alloc ((type), cass_size_t row, cass_size_t col);`

`void matrix_free (type **matrix);`

`type *** type_matrix3_alloc ((type), cass_size_t num, cass_size_t row, cass_size_t col);`

`void matrix3_free (type ***matrix);`

Description:

Allocate/free memory for the matrix.

Prototype: `int type_matrix_load_stream (type, FILE *stream, cass_size_t *row, cass_size_t *col, type ***matrix);`

`int type_matrix_load_file (type, const char *path, cass_size_t *row, cass_size_t *col, type ***matrix);`

`int type_matrix_dump_stream (type, FILE *stream, cass_size_t row, cass_size_t col, type **matrix);`

`int type_matrix_dump_file (type, const char *path, cass_size_t row, cass_size_t col, type **matrix);`

Description:

Read and write matrix, either with a stream or a file. These routines are not portable and are to be modified to use `CASS_FILE` instead.

Prototype: `int type_matrix_map_file (type, const char *path, cass_size_t *row, cass_size_t *col, type ***matrix);`

`int matrix_unmap_file (void **matrix);`

Description:

Memory mapping, similar to `type_matrix_load_file`.

3.5 Library Initialization and Cleanup

Prototype: `int cass_init (void);`

`int cass_cleanup (void);`

Description:

Call `cass_init` before using any of the Ferret Library routines, and call `cass_cleanup` at the end of your program.

3.6 Database Environment

3.6.1 Open and close

Prototype: `int cass_env_open (cass_env_t **env, char *db_home, uint32_t flags);`

`int cass_env_close(cass_env_t *env, uint32_t flags);`

Description:

The function `cass_env_open` opens a Ferret database for future use and send back the environment pointer through the first argument. `db_home` should contain the path to the directory in which contains the Ferret database. `flags` can be one or more of the following flags.

`CASS_READONLY`
: Open the database read-only. Without this flag specifiled, the database will be opened read-write.

`CASS_EXCL`
: If the directory does not exist, create the database; otherwise fail.

The function `cass_env_close` closes the database provided as the first argument. Currently there's no flags supported and the user should always pass in 0.

3.6.2 Logging

Prototype: `int cass_env_errmsg(cass_env_t *env, int error, const char *fmt, ...);`

`int cass_env_panic(cass_env_t *env, const char *msg);`

Description:

The function `cass_env_errmsg` works like `fprintf`, except it writes the message to the database log file.

The function `cass_env_panic` kills the application after printing out the error message.

3.6.3 Checkpointing

Prototype:

`int cass_env_checkpoint(cass_env_t *env);`

Description:

Flush any changes of the database since last checkpoint into disk. This function may lock the database and cause significant delay to the panding modifications.

3.6.4 Describing

Prototype: `int cass_env_describe (cass_env_t *env, CASS_FILE *);`

Description:

This function writes the textual description of the database to the provided stream.

3.7 Mapping

3.8 Vector, Vecset and Dataset

3.8.1 Data structures

A vecset is a set of vectors, and a dataset is a set of vecsets. A vecset usually contains the feature vectors corresponding to an object, which is segmented into several part, with each part described by a vector in the vecset. A Ferret table is a dataset, which contains multiple vecsets. The three structures are defined as follows.

```
/* vector */
typedef struct {
        float weight;
        cass_vecset_id_t parent;
        union {
                uchar data[0];
                int32_t int_data[0];
                float float_data[0];
                chunk_t bit_data[0];
        };
} cass_vec_t;

/* vecset */
typedef struct {
        uint32_t num_regions;
        cass_vec_id_t start_vecid;
} cass_vecset_t;

/* dataset */
typedef struct {
        uint32_t  flags;
        uint32_t loaded;
        cass_size_t             vec_size;
        cass_size_t             vec_dim;
        cass_vec_id_t           max_vec;
        cass_vec_id_t           num_vec;
        void                    *vec;
        cass_vecset_id_t        max_vecset;
        cass_vecset_id_t        num_vecset;
        cass_vecset_t           *vecset;
} cass_dataset_t;
```

`cass_vec_t` and `cass_vecset_t` are not standalone; they depend on information in `cass_dataset_t` and even See [`cass_vecset_cfg`], page 17, to determine how to interprete there fields. The fields of the three structs are explained below.

First, `cass_dataset_t`.

`flags` A table can contain either vectors or vecsets, or both, and this field specifies what is contained in the dataset. It can be one of the following values.

`CASS_DATASET_VEC`
: The datset contains only vector data.

`CASS_DATASET_VECSET`
: The dataset contains only vecset data.

`CASS_DATASET_BOTH`
: The dataset contains both vectors and vecsets.

`loaded` Whether the feature data (vectors and/or vecsets) are in memory and can be accessed through `vec`/`vecset` field.

`vec_size` Size of a vector in bytes, including a 32-bit weight and the real feature data.

`vec_dim` The dimension of the feature vector.

`max_vec` The maximum number of vectors the memory allocated in `vec` can hold.

`num_vec` The real number of vectors in the dataset.

`vec` Pointer to the memory holding the vectors.

`max_vecset`
: The maximum number of vecsets the memory allocated in `vecset` can hold.

`num_vecset`
: The real number of vecsets in the dataset.

`vecset` Pointer to the memory holding the vecsets.

Second, `cass_vecset_t`.

`num_regions`
: Number of vectors in this vecset.

`start_vecid`
: The offset of the first vector in the dataset. The vectors in dataset that belong to this vecset are those numbered from `start_vecid` to `(start_vecid + num_regions -1)`. Thoses numbers are only meaningful to the dataset that contains the vecset.

Finally, `cass_vec_t`.

`weight` The weight of the vector inside the vecset.

`parent` The id of the vecset which contains the vector.

`[int_|float_|bit_]data`
: This is the real data of the vector. Different field of the union should be used according to the configuration of the vecset. Also, because the dimension is variable, these fields works as a place holder.

`cass_vec_t` requires a bit more explanation. This struct actually works as a placeholder, and when the dimensionality of the vector is determined, the real data will always be larger than the struct in size. In other words, `sizeof(cass_vec_t)` should never be used except for taking the size of the vector without the real feature data. The variable size of `cass_vec_t` makes it a little tricky to locate a perticular vector in a dataset. The following piece of code illustrates how to do the job.

```
/* Get the i-th vector in the dataset */
cass_vec_t *vec = (void *)dataset->vec + dataset->vec_size * i;
```

3.8.2 Dataset routines

The definition of all the above defined structures are considered explicit, and are safe to be modified by the user. The following routines make common operations on datasets easier.

Prototype: `int cass_dataset_init (cass_dataset_t *ds, cass_size_t vec_size, cass_size_t vec_dim, uint32_t flags);`

Description:

Initialize a dataset, without allocating memory to hold the feature data.

Prototype: `int cass_dataset_grow (cass_dataset_t *ds, cass_size_t num_vecset, cass_size_t num_vec);`

Description:

(Re-)allocate memory of the dataset so that it can hold at least `num_vecset` vecsets and `num_vec` vectors.

Prototype: `int cass_dataset_release (cass_dataset_t *ds);`

Description:

Release the memory of the feature data. Not actually frees the `ds` pointer.

Prototype: `int cass_dataset_merge (cass_dataset_t *ds, const cass_dataset_t *src, cass_vecset_id_t start, cass_vecset_id_t num, cass_dataset_map_t map, void *map_param);`

Description:

Merge the `num` vecsets start from `start` of the dataset `src` to the dataset `ds`, grow `ds` when necessary. If the dataset contains both vectors and vecsets, the `parent` field of vectors are updated according to the new position of the corresponding vecsets. However, if the dataset contains vector only, the `parent` field is directly copied.

Prototype: `int cass_dataset_checkpoint (cass_dataset_t *ds, CASS_FILE *);  int cass_dataset_restore (cass_dataset_t *ds, CASS_FILE *);`

Description:

Read/write the meta data.

Prototype: `int cass_dataset_load (cass_dataset_t *ds, CASS_FILE *in); int cass_dataset_dump (cass_dataset_t *ds, CASS_FILE *out);`

Description:

Read/write the feature data.

Description:

3.9 Class, Instance and Registry

The Ferret Toolkit is designed such that many components work as plugins and can be customized. For example, there are different index/sketch plugins that meet different requirements of precision, speed and storage overhead. One specific index algorithm can be viewd as a class, and it is instanciated when a real index structure is created upon a dataset. The vecset distance and vector distance also follow this paradigm. For example, a L2 distance algorithm that only evaluate with a certain subset of all the dimensions is a class, and it is instanciated when the specific interested subset of the dimensions to use is provided. A class is defined with a number of properties and a set of method, and a instance of a class also has a set of properties and a pointer to the class. Following is an example showing the structs describing the vector distance class and vector distance instances.

```
typedef cass_dist_t (*cass_vec_dist_func_t) (cass_size_t n, void *, void *, void *);

/* this is the class */
typedef struct {
        char                    *name;
        cass_vec_type_t          vec_type;
        cass_vec_dist_type_t     type;
        cass_vec_dist_func_t     dist;
        int (*describe) (void *, CASS_FILE *);
        int (*construct) (cass_vec_dist_t **, const char *);
        int (*checkpoint) (void *, CASS_FILE *);
        int (*restore) (void **, CASS_FILE *);
        void (*free) (void *);
} cass_vec_dist_class_t;

/* this is the instance */
typedef struct {
        uint32_t        refcnt;
        char           *name;
        cass_vec_dist_class_t *class;
        /* private data... */
} cass_vec_dist_t;
```

Given a struct `dist_class` of the class, an instance can be created with `dist_class->construct`, which returns the constructed instance through its first argument. Because different class accept different parameters, a string is used to encode the parameters. The parameter string could be something like `"-M 128 -W 3.2"`.

See See [Extending Ferret], page 36, for more information.

Both class and instance are identified by an internal ID and an external textual name, and a registry is used to map between the ID, name and the pointer to the struct. Each type of class/instance has its own name space, and the namespaces of class and instance are seperate. For example, you can create a distance instance "trivial" from the distance class "trivial".

Following are the registry routines.

Prototype: `typedef struct ... cass_reg_t;`

Description:

This is the registry type.

Prototype: `int cass_reg_init (cass_reg_t *reg);`

`int cass_reg_init_size (cass_reg_t *reg, cass_size_t size);`

Description:
Initialize a registry. The latter initialize the registry to hold `size` entries, and is used when the size of registry is known and fixed in advance.

Prototype: `int cass_reg_cleanup (cass_reg_t *reg);`

Description:
Cleanup the registry, release the memory.

Prototype: int32_t cass_reg_lookup (cass_reg_t *, const char *name);

Description:
Map name to ID. If the name does not exist, -1 is returned.

Prototype: int32_t cass_reg_find (cass_reg_t *, const void *ptr);

Description:
Map pointer to ID. If the pointer does not exist in the registry, -1 is returned.

Prototype: void *cass_reg_get (cass_reg_t *, uint32_t i);

Description:
Map ID to pointer.

Prototype: int cass_reg_add (cass_reg_t *, const char *name, void *);

Description:
Add an item to the registry, return the ID of the newly added item.

3.10 Vector Distance and Vecset Distance

To enable customization, the vector distance and vecset distance come with a parameter instead of simply a pointer to function, and the parameter and pointer to function are wrapped up in a struct. Following are the relative declarations of vector distance. Both vector/vecset distance follow the class/instance model.

Following are declarations relative to vector distance.

```
typedef cass_dist_t (*cass_vec_dist_func_t) (cass_size_t n, void *v1, void *v2,
cass_vec_dist_t *);

typedef struct _cass_vec_dist_class
        char *name;
        cass_vec_type_t          vec_type;
        cass_vec_dist_type_t     type;
        cass_vec_dist_func_t     dist;

        int (*describe) (void *, CASS_FILE *);
        int (*construct) (void **, const char *param);
        int (*checkpoint) (void *, CASS_FILE *);
        int (*restore) (void **, CASS_FILE *);
        void (*free) (void *);
 cass_vec_dist_class_t;

typedef struct
        uint32_t refcnt;
        char *name;
```

```
        cass_vec_dist_class_t *class;
        /* private data... */
    cass_vec_dist_t;
```

Following are the explanation of some of the fields of `cass_vec_dist_class_t`.

`vec_type` The type of vector on which this distance is defined.

`type` Type of the distance. Can be one of the following values.

`CASS_VEC_DIST_TYPE_TRIVIAL`
`CASS_VEC_DIST_TYPE_ID`
`CASS_VEC_DIST_TYPE_L1`
`CASS_VEC_DIST_TYPE_L2`
`CASS_VEC_DIST_TYPE_MAX`
`CASS_VEC_DIST_TYPE_HAMMING`
`CASS_VEC_DIST_TYPE_COS`

Note that the distance type itself do not determine the distance. For example, L1 distance evaluate using all the dimensions and those evaluated using a subset of the dimensions are all of the type `CASS_VEC_DIST_TYPE_L1`.

`dist` The pointer to the function.

`describe` Writes textual describing information to the stream.

`construct`
Construct the distance instance using the given textual parameter.

`checkpoint`
Write the meta data of distance into a stream.

`restore` Construct a distance instance from a stream.

`free` Free the distance instance.

The fields of `cass_vec_dist_t` actually do not need explanation. One thing to note that `cass_vec_dist_t` is a basic type, and for different distance classes, there can be more specific types which can hold private information. For example, if only certain subset of all dimensions are interested, then the distance can be defined as follows:

```
typedef struct
{
cass_vec_dist_t base;
ARRAY_TYPE(int) dim; /* array of interested dimensions */
} cass_l1_dim_t;
```

One can then safely cast `cass_l1_dim_t *` to `cass_vec_dist_t *`.

Following are routines to access the distance classes. They are just specialized version of the vector distance class registry.

Prototype:
```
int32_t cass_vec_dist_class_lookup (const char *n);
int32_t cass_vec_dist_class_find (const cass_vec_dist_class_t *p);
cass_vec_dist_class_t *cass_vec_dist_class_get (uint32_t i);
```

The declaration of vecset distance is similar.

```
typedef cass_dist_t (*cass_vecset_dist_func_t) (cass_dataset_t *, cass_vecset_id_t,
cass_dataset_t *, cass_vecset_id_t , cass_vec_dist_t *vec_dist,
cass_vecset_dist_t *);

typedef struct _cass_vecset_dist_class
char *name;
cass_vecset_type_t vecset_type;
cass_vecset_dist_type_t type;
cass_vecset_dist_func_t dist;
/* void ** is actually cass_vecset_dist_t ** */
int (*describe) (void *, CASS_FILE *);
int (*construct) (void **, const char *param);
int (*checkpoint) (void *, CASS_FILE *);
int (*restore) (void **, CASS_FILE *);
void (*free) (void *);
/* private data... */
 cass_vecset_dist_class_t;

typedef struct
uint32_t refcnt;
char *name;
int32_t vec_dist;
cass_vecset_dist_class_t *class;
 cass_vecset_dist_t;
```

There are also routines to access the vecset distance class registry.

Prototype: `int32_t cass_vecset_dist_class_lookup (const char *n);`

`int32_t cass_vecset_dist_class_find (const cass_vecset_dist_class_t *p);`

`cass_vecset_dist_class_t *cass_vecset_dist_class_get (uint32_t i);`

3.11 Configurations and Tables

Just like a table in a relational database has a scheme, a table in a Ferret database has a configuration, which is specified by the user when the table is created. Following is the definition of the configuration struct.

```
typedef struct {
        uint32_t                refcnt;
        char                   *name;
        cass_vecset_type_t      vecset_type;
        cass_vec_type_t         vec_type;
        cass_size_t             vec_dim;
        cass_size_t             vec_size;
        uint32_t                flags;
} cass_vecset_cfg_t;
```

The fields are explained below.

`refcnt` The reference count of the structure. Unlike the scheme in the relational database, which is always associated with a specific table, a configuration in a Ferret

database is a independent object, and can be shared among more than one tables. The reference count is used to maintain the number of tables, or other objects that keep a pointer to the configuration struct.

`name` The name of the configuration. The memory of the string is owned by the struct, and will be freed when the configuration is destroyed.

`vecset_type`
Type of the vecset, can be one of the following values:

`CASS_VECSET_SINGLE`
Any vecset in the table always has one and only one vector.

`CASS_VECSET_SET`
A vecset can have more than one vectors. This is the most common case.

`CASS_VECSET_NONE`
The table do not keep data for vecset, but only keep data for vectors. This is used for sketch methods which disregard the vecset and only deal with vectors.

`vec_type` The type of the vector, can be one of the following values:

`CASS_VEC_INT`
32-bit integer, or `int32_t`.

`CASS_VEC_FLOAT`
32-bit floating point number, or `float`.

`CASS_VEC_BIT`
The feature vector is a bit vector.

`CASS_VEC_QUANT`
The feature vector is some kind of quantization. Used for certain index method.

`vec_dim` The dimension of the feature vector. For bit vectors, it means the number of bits.

`vec_size` The size of the feature vector. Which include the feature vector and a weight.

`flags` For now always set 0.

3.11.1 Table Creation

Prototype: `int cass_table_create (cass_table_t **table, cass_env_t *env, char *table_name, uint32_t opr_id, int32_t cfg_id, int32_t parent_id, int32_t parent_cfg_id, int32_t map_id, char *param);`

Description:
This routine creates a new table. Note that this routine does not associate the file with its parent. To do this, use See [`cass_table_associate`], page 19.

`table` The pointer to the newly created table is return through this pointer.

`env` The environment in which the table is created.

`table_name`
Name of the table.

`opr_id` The ID of the table class. See See [Index and Sketch Reference], page 26, for possible values.

`cfg_id` The ID of the table configuration. For tables do not need a configuration, like index, `cfg_id` can be -1.

`parent_id`
The ID of the parent table. For tables without a parent, use -1.

`parent_cfg_id`
The ID of the parent table's configuration. Can be -1.

`map_id` The ID of the map object to use.

`param` Extra parameters to pass to the detail implementation. See See [Index and Sketch Reference], page 26, for details.

3.11.2 Free a Table

Prototype: `int cass_table_free(cass_table_t *table);`

Description:
This routine frees the table from the main memory. The on-disk table is not destroyed.

3.11.3 Describe a Table

Prototype: `int cass_table_describe (cass_table_t *, CASS_FILE *);`

Description
This routine writes textual description of the table to the provided stream.

3.11.4 Import and Export Data

Prototype: `int cass_table_import_data(cass_table_t *table, char *fname);`
`int cass_table_export_data(cass_table_t *table, char *fname);`

Description:
These two routines import/export a table from/to a external text file. See See [Data File Format], page 29, for details of the text file format.

3.11.5 Table Association

Prototype: `int cass_table_associate (cass_table_t *table, int32_t child);`
`int cass_table_disassociate (cass_table_t *table, int32_t child);`
`cass_table_t *cass_table_parent (cass_table_t *table);`

Description:
`cass_table_associate` associates table `child` (the ID) to `table`, so `child` becomes a child of `table`. When there are insertion to a table, the same data are automatically inserted to its children.

`cass_table_disassociate` revokes the relationship.

`cass_table_parent` returns the parent of the table, or `NULL` when the table has no parent.

3.11.6 Load and Release Feature Data

Prototype:

```
int cass_table_load (cass_table_t *table);
int cass_table_release (cass_table_t *table);
```

Description:

When the database is opened, the meta data of all the tables, as well as other objects are loaded into memory. However, the real feature data, (or index/sketches) are not. In this way, the system knows the structural information of the database without too much memory overhead. The feature data are only loaded when necessary and are released when not needed. If the feature data are modified, they are automatically dumped to disk before being released. You can refer to the `loaded` field of `cass_table_t` to determine if the feature data are loaded. The feature data must be loaded before data insertions or queries.

3.11.7 Data Insertion

Prototype: `int cass_table_batch_insert (cass_table_t *table, cass_dataset_t *dataset, cass_vecset_id_t start, cass_vecset_id_t end);`

Description:

This routine inserts the vecsets in `dataset` indexed by the range [`start`, `end`] into `table`.

3.12 Queries

There are two types of queries in the Ferret toolkit – range queries and K-NN queries. K-NN queries are well supported and there are index/sketches for various distance measures. For each query, the user specifies one query vecset and the required range/# nearest neighbors, and the system tries the best to return the results. Queries can be in two levels – vecset level or vector level. In vecset level, the vecsets in the table that are closest to the query vecset or within the specified range are returned. In vector level, a queries is carried out for each vector in the query vecset, and the results are returned in multiple sets. Also, in vector level queries, the dataset is the union of all the vecsets in the table, and the vecset-vector relationship is disregarded. That means, the more than one vectors in the result set can belong to the same vecset. Further more, the result sets can either be returned as an array or as a bitmap.

To issue a query, the user need to prepare a query data structure to specify the query information, and provide a result structure to hold the return value. The related structures are described below.

```
typedef struct {
        cass_id_t               id;
        cass_dist_t             dist;
```

```
} cass_list_entry_t;

typedef ARRAY_TYPE(cass_list_entry_t) cass_list_t;

typedef struct {
        uint32_t                flags;

        union {
         bitmap_t        bitmap;
                cass_list_t     list;
                ARRAY_TYPE(bitmap_t) bitmaps;
                ARRAY_TYPE(cass_list_t) lists;
        };
} cass_result_t;

typedef struct {
        uint32_t                flags;

        cass_dataset_t          *dataset;
        cass_id_t                vecset_id;

        cass_size_t              topk;
        cass_dist_t              range;

        char                    *extra_params;
        cass_result_t           *candidate;

        int32_t                  vec_dist_id;
        int32_t                  vecset_dist_id;
} cass_query_t;
```

The `cass_result_t` is a union plus a `flags`, which also appears in `cass_query_t` and gives information on how to interprete the union and other requirements. `flags` in `cass_query_t` specifies the user requirement and `flags` in `cass_result_t` specify the system response. The possible values of the two are essentially the same. Specifically, the user can specify one of the following four values in the `flags` of the `cass_query_t` structure, to specify which representation of the result in `cass_result_t` is required.

`CASS_RESULT_BITMAP`
: The `bitmap` field of the union is used. The query should be in vecset level.

`CASS_RESULT_LIST`
: The `list` field of the union is used. The query should also be in vecset level.

`CASS_RESULT_BITMAPS`
: The `bitmaps` field of the union is used. The query should be in vector level.

`CASS_RESULT_LISTS`
: The `lists` field of the union is used. The query should also be in vector level.

Note that for vector level queries, even if there is always only one vector in the vecset (the vecset is of the type `CASS_VECSET_SINGLE`), either `bitmaps` or `lists` should be used.

The `flags` can also have one or more of the following ORed to its value.

`CASS_RESULT_MALLOC`
: The memory to hold the real data in `list`/`lists` have not been allocated by the user and should be allocated by the system.

`CASS_RESULT_USERMEM`
: The user has allocated memory in `list`/`lists` and the system should use the user provided memory. This is useful when there are multiple contiguous queries, allowing the user to allocate memory once and reuse for the following queries .

`CASS_RESULT_REALLOC`
: The user should not set this flag, but if it appears in the return value, it means that the user allocated memory is not enough and the system has reallocated the memory.

`CASS_RESULT_SORT`
: The user requires that the result list to be sorted.

`CASS_RESULT_DIST`
: The user requires that the `dist` field of `cass_list_entry_t` be filled if `list` or `lists` is used. This value means the distance of the corresponding vector/vecset to the query vector/vecset.

It is important to note that all the above flags are treated as suggestions rather than mandatories. The user should always check the `flags` of `cass_result_t` and interpret the result accordingly. For example, some of the index/sketch algorithm do not refer to the original feature data, and have no way to figure out the distance between the data vector/vecset and the query vector/vecset, so even if the user requires `CASS_RESULT_DIST`, it will not be returned. Some of the index/sketch algorithm keep the K-NN's in a heap, and it will be cheap to sort the heap with heap sort than regular quicksort. In that case, if the user requres `CASS_RESULT_SORT`, the results will be sorted. For other algorithms, if sorting within the query procedure is no faster than outside it by the user, the algorithm will disregard the sorting requirement. Finally, some of the index method, like LSH, will always return bitmaps instead of lists.

The other fields of the `cass_query_t` are explained below.

`dataset`

`vecset_id`
: The vecset in `dataset` specified by `vecset_id` is used as the query vecset.

`topk`

`range`
: If `topk > 0`, the system does the K-NN query; otherwise, the system does the range query which is specified by `range`.

`extra_params`
: The extra parameters, as a string, passed to the query algorithm. The string is interpreted differently by different query index/sketch algorithms.

`candidate`
: If `candidate != NULL`, then the query is carried out within the data provided by `candidate` instead of the whole dataset. The often happens when the user wants to refine the result of a fast but inaccurate index algorithm with a more accurate one. In that case, `candidate` can directly point to the previous query result. In the case of vector level query, the order of lists/bitmaps in `candidate` should be the same as the order of vectors in the query vecset.

`vec_dist_id`
`vecset_dist_id`
: The vector distance and vecset distance to be used. If it is a vector level query, `vecset_dist_id` is disregarded.

The following routines are used to issues queries to tables.

Prototype: `int cass_table_query (cass_table_t *table, cass_query_t *query, cass_result_t *result);`

`int cass_table_batch_query (cass_table_t *table, uint32_t count, cass_query_t **queries, cass_result_t **results);`

Description:
: The function of the batch query is same as doing single query multiple times. But for some index/sketch algorithm, doing multiple queries in a batch allows higher performance. But this is not always the case.

3.13 Multiple Modality Support

3.14 Vector Distance Reference

3.14.1 Trivial distance

Name: trivial

Type: `CASS_VEC_DIST_TYPE_TRIVIAL`

Vector type:
: `CASS_ANY`

Parameters:
: (none)

Description:
: The trivial distance measure always return 0 for any pair of vecsets. This vecset distance is only for debugging purpose.

3.14.2 Integer L1 distance

Name: L1_int

Type: `CASS_VEC_DIST_TYPE_L1`

Vector type:
: `CASS_VEC_INT`

Parameters:
: (none)

Description:

3.14.3 Integer L2 distance

Name: L2_int

Type: `CASS_VEC_DIST_TYPE_L2`

Vector type:
`CASS_VEC_INT`

Parameters:
(none)

Description:

3.14.4 Float L1 distance

Name: L1_float

Type: `CASS_VEC_DIST_TYPE_L1`

Vector type:
`CASS_VEC_FLOAT`

Parameters:
(none)

Description:

3.14.5 Float L2 distance

Name: L2_float

Type: `CASS_VEC_DIST_TYPE_L2`

Vector type:
`CASS_VEC_FLOAT`

Parameters:
(none)

Description:

3.14.6 Cosine distance

Name: cosine

Type: `CASS_VEC_DIST_TYPE_COS`

Vector type:
`CASS_VEC_FLOAT`

Parameters:
(none)

Description:
This is actually the dot product of two vectors. If the vectors are not unit vectors, then it's actually not the cosine distance.

3.14.7 Hamming distance

Name: hamming

Type: `CASS_VEC_DIST_TYPE_HAMMING`

Vector type:
`CASS_VEC_BIT`

Parameters:
(none)

Description:

3.15 Vecset Distance Reference

3.15.1 Trivial distance

Name: trivial

Type: `CASS_VECSET_DIST_TYPE_TRIVIAL`

Vecset type:
`CASS_ANY`

Parameters:
(none)

Description:
The trivial distance measure always return 0 for any pair of vecsets. This vecset distance is only for debugging purpose.

3.15.2 Single distance

Name: single

Type: `CASS_VECSET_DIST_TYPE_SINGLE`

Vecset type:
`CASS_ANY`

Parameters:
(none)

Description:
Take the first vector from each of the vecsets and return the distance between the vectors.

3.15.3 EMD distance

Name: emd

Type: `CASS_VECSET_DIST_TYPE_EMD`

Vecset type:
`CASS_ANY`

Parameters:
(none)

Description:
The Earth Mover's Distance.

3.16 Index and Sketch Reference

3.16.1 Raw Table

Name: raw

Type: `CASS_DATA`

Vector type:
`CASS_VEC_ANY`

Vecset type:
`CASS_ANY`

Vector distance type:
`CASS_ANY`

Vecset distance type:
`CASS_ANY`

Parameters:
Description:
This is the raw table holding the real feature data and is not actually an index or sketch.

3.16.2 Locality Sensitive Hashing

Name: LSH

Type: `CASS_VEC_INDEX`

Vector type:
`CASS_VEC_FLOAT`

Vecset type:
`CASS_ANY`

Vector distance type:
`CASS_VEC_DIST_TYPE_L1`

Vecset distance type:
`CASS_ANY`

Parameters:
Description:

3.16.3 Sketch for L1 Distance

Name: sketch1

Type: `CASS_VEC_SKETCH`

Vector type:
`CASS_VEC_FLOAT`

Vecset type:
`CASS_ANY`

Vector distance type:
`CASS_VEC_DIST_TYPE_L1`

Vecset distance type:
`CASS_ANY`

Parameters:

`-B #bit` Size of sketch in bits.

`-H #xor` How many bits to XOR in order to generate one sketch bit.

`-min min_array`
The minimal value of each dimension in the dataset.

`-rng range_array`
The range of each dimension in the dataset

`-weight weight_array`
The weight of each dimension in the dataset. This parameter is optional and has a default value of `(1,1,...,1)`.

Description:

3.16.4 Sketch for L2 Distance

Name: sketch2

Type: `CASS_VEC_SKETCH`

Vector type:
`CASS_VEC_FLOAT`

Vecset type:
`CASS_ANY`

Vector distance type:
`CASS_VEC_DIST_TYPE_L2`

Vecset distance type:
`CASS_ANY`

Parameters:

`-M #bit` Size of sketch in bits.

`-W window` Window size.

Description:

3.16.5 Sketch for Cosine Distance

Name: sketch3

Type: `CASS_VEC_SKETCH`

Vector type:
`CASS_VEC_FLOAT`

Vecset type:
`CASS_ANY`

Vector distance type:
`CASS_VEC_DIST_TYPE_COS`

Vecset distance type:
`CASS_ANY`

Parameters:

`-M #bit` Size of sketch in bits.

Description:

4 Data Layout

4.1 Memory Data Layout

4.2 Directory Layout

4.3 Data File Format

4.4 Benchmark File Format

5 The Ferret Server

6 Utility Programs

6.1 cass_init

Synopsis: `cass_init <path>`

Description:
Creat a database in the given directory.

Options:

path	The path to where to create the database.

6.2 cass_describe

Synopsis: `cass_describe <path>`

Description:
Dump textual description of the database.

Options:

path	The database directory.

6.3 cass_add_vec_dist

Synopsis: `cass_add_vec_dist <path> <name> <class> <param>`

Description:
Add a vector distance to the databse.

Options:

path	The database directory.
name	The name of the distance to add.
class	The name of the class of the distance.
param	Parameters to the construct method of the class.

6.4 cass_add_vec_set_dist

Synopsis: `cass_add_vec_set_dist <path> <name> <class> <param>`

Description:
Add a vecset distance to the database.

Options:

path	The database directory.
name	The name of the distance to add.
class	The name of the class of the distance.
param	Parameters to the construct method of the class.

6.5 cass_add_cfg

Synopsis: `cass_add_cfg <path> <cfg name> <vecset type> <vec type> <dim>`

Description:

Add a vecset configuration to the database.

Options:

path The database.

cfg name Name of the configuration to add.

vecset type

The vecset type, can be the following values.

`1` For `CASS_VECSET_SINGLE`.

`2` For `CASS_VECSET_SET`.

vec type The vector type, can be the following values.

`int`

`float`

`bit`

dim Dimension of feature vector.

6.6 cass_add_map

Synopsis: `cass_add_map <path> <map name> <param>`

Description:

Add a map to the database.

Options:

path The database.

map name Name of the map to add.

param Can be one of the following values.

`direct` The external ID are sequential integers start from 0 and do not require mapping.

`indirect` The external ID are strings and require to be converted to sequential integers for internal usage.

6.7 cass_add_table

Synopsis: `cass_add_table <path> <table name> <cfg> <map>`

Description:

Add a table to the database.

Options:

path The database.

table name
: Name of the table.

cfg
: Name of the configuration to use for the table.

map
: Name of the map to use for the table.

6.8 cass_add_sketch

Synopsis: `cass_add_sketch`

Description:

Options:

6.9 cass_query

Synopsis: `cass_query <path> <table> <benchmark> <#query> <K> <T1> <T2> [extra1] [extra2]`

Description:

The table parameter should be like "shape@sketch2", which indicates the database to use the sketch. If "shape" is used, then sketch will not be used and linear scan will be carried out.

The query algorithm works in the following way:

1. *T1* candidates are found using the sketch algorithm. If *extra1* presents, it is passed to the sketch query algorithm. 2. The candidates are further filtered using the sketch algorithm, with the best *T2* candidates left. If *extra2* presents, it is passed to the sketch query algorithm.

The benchmark files use true K-NN's found by linear scan as ground truth.

Examples are

../cass_run_query all shape@sketch2 ~/data/query_shape 100 100 2000 1000 "" "-asym"

This will first use the sketch to find 2000 candidate, and then use asymmetric method to refine the candidate set to 1000 candidates.

For now, the benchmark file has 10000 queries, each with 200 nearest neighbors, so you need to make sure that <# query> <= 10000 && <top k> <= 200.

Options:

6.10 cass_import

Synopsis: `cass_import <path> <table> <input>`

Description:

Import external data to a table

Options:

path
: The database.

table
: The table to import data to.

input
: The path to the input file.

6.11 cass_export

Synopsis: `cass_export <path> <table> <ouput>`

Description:

Dump the data of a table to a text file.

Options:

path	The database.
table	The table to export.
output	The output file name.

7 Feature Extraction

7.1 Image

8 Extending Ferret

8.1 Adding New Vector Distances

```
typedef cass_dist_t (*cass_vec_dist_func_t) (cass_size_t n, void *, void *, void *);

typedef struct _cass_vec_dist_class {
        char *name;
        cass_vec_type_t vec_type;
        cass_vec_dist_type_t type;
        cass_vec_dist_func_t dist;
        /* void ** is actually cass_vec_dist_t ** */
        int (*describe) (void *, CASS_FILE *);
        int (*construct) (void **, const char *);
        int (*checkpoint) (void *, CASS_FILE *);
        int (*restore) (void **, CASS_FILE *);
        void (*free) (void *);
} cass_vec_dist_class_t;

typedef struct {
        uint32_t refcnt;
        char *name;
        cass_vec_dist_class_t *class;
        /* private data... */
} cass_vec_dist_t;
```

8.2 Adding New Vecset Distances

```
typedef cass_dist_t (*cass_vecset_dist_func_t) (cass_dataset_t *, cass_vecset_id_t,
                cass_dataset_t *, cass_vecset_id_t , cass_vec_dist_t *vec_dist, void *);

typedef struct _cass_vecset_dist_class{
        char *name;
        cass_vecset_type_t vecset_type;
        cass_vecset_dist_type_t type;
        cass_vecset_dist_func_t dist;
        /* void ** is actually cass_vecset_dist_t ** */
        int (*describe) (void *, CASS_FILE *);
        int (*construct) (void **, const char *);
        int (*checkpoint) (void *, CASS_FILE *);
        int (*restore) (void **, CASS_FILE *);
        void (*free) (void *);
        /* private data... */
} cass_vecset_dist_class_t;

typedef struct {
        uint32_t refcnt;
        char *name;
        int32_t vec_dist;
        cass_vecset_dist_class_t *class;
} cass_vecset_dist_t;
```

8.3 Adding New Indices and Sketches

```
typedef struct _cass_table_opr cass_table_opr_t;
```

```
/*    table operations */

typedef int cass_table_init_private_t (cass_table_t *table,  // See management section
                                const char *param);

typedef int cass_table_opr_describe_t (cass_table_opr_t *table, CASS_FILE *out);

typedef int cass_table_checkpoint_private_t (cass_table_t *table, CASS_FILE *out);

typedef int cass_table_restore_private_t (cass_table_t *table, CASS_FILE *in);

typedef int cass_table_load_t (cass_table_t *table);  // bring data in mem

typedef int cass_table_release_t (cass_table_t *table);  // release in-mem vecsets.

typedef int cass_table_batch_insert_t (cass_table_t *table, cass_dataset_t *dataset,
                cass_vecset_id_t start, cass_vecset_id_t end);

typedef int cass_table_query_t(cass_table_t *table,
        cass_query_t *query, cass_result_t *result);

typedef int cass_table_batch_query_t(cass_table_t *table,
        uint32_t count, cass_query_t **queries, cass_result_t **results);

typedef int cass_table_drop_t(cass_table_t *table);

typedef int cass_table_free_private_t(cass_table_t *table);

typedef char *cass_table_tune_t(cass_table_t *parent, char *extra_input);

struct _cass_table_opr {
    char *name;
    // CASS_DATA or CASS_SKETCH or CASS_INDEX, CASS_OUTOFCORE, CASS_SEQUENTIAL
    int type;
    cass_vecset_type_t               vecset_type;
    cass_vec_type_t                  vec_type;
    cass_vecset_dist_type_t          dist_vecset;
    cass_vec_dist_type_t             dist_vec;

    cass_table_tune_t               *tune; // a set of function pointers.
    cass_table_init_private_t       *init_private;
    cass_table_batch_insert_t       *batch_insert;
    cass_table_query_t              *query;
    cass_table_batch_query_t        *batch_query;
    cass_table_load_t               *load; // load data
    cass_table_release_t            *release; // release data
    cass_table_checkpoint_private_t *checkpoint_private;
    cass_table_restore_private_t    *restore_private;
    cass_table_free_private_t       *free_private;
    cass_table_opr_describe_t       *describe; // load data
};
```

8.4 Supporting New Data Types

Index

(Index is nonexistent)