Apache processes and threads in mod_ndb

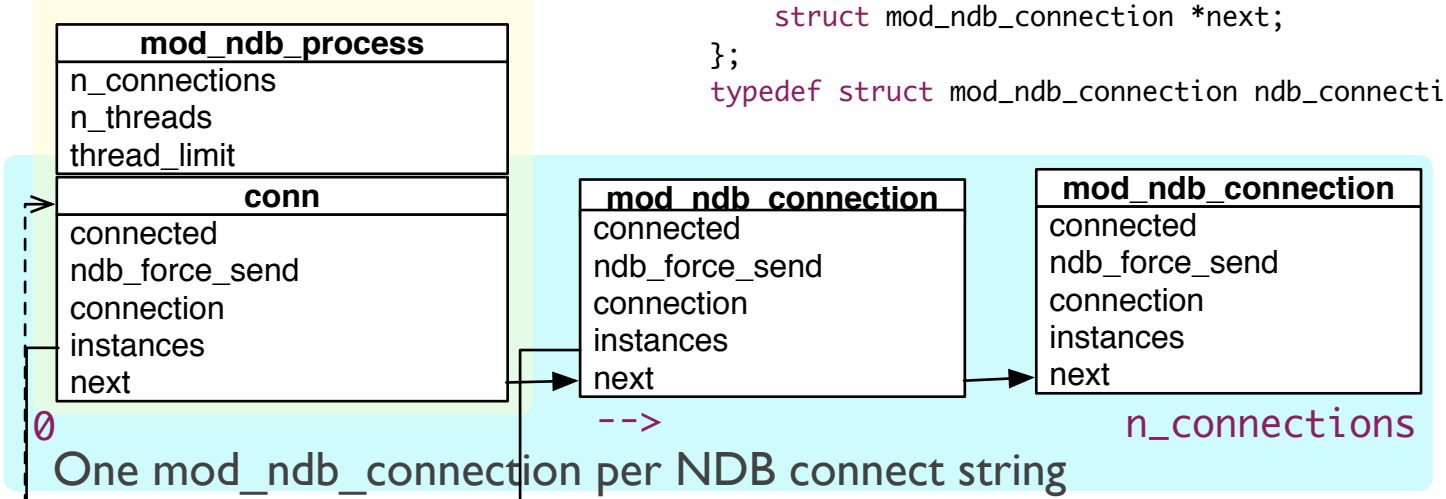
mod_ndb.h

```
struct mod_ndb_process {
    int n_connections;
    int n_threads;
    int thread_limit;
    struct mod_ndb_connection conn;  // not a pointer
};
```

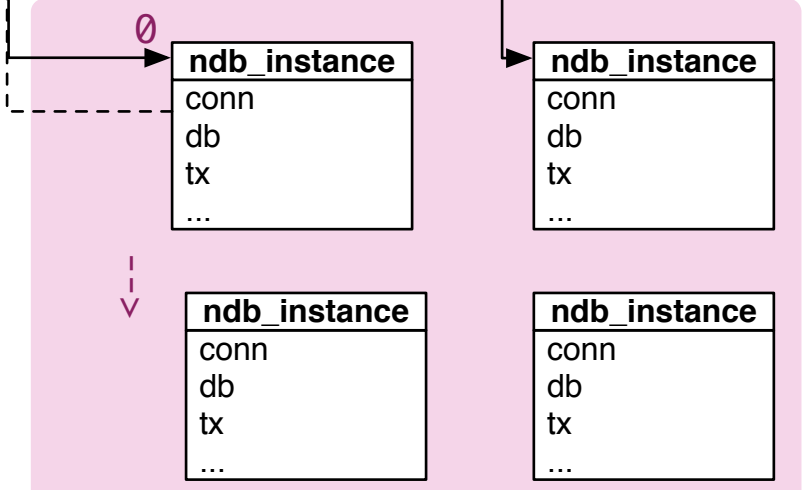
One mod_ndb_process per Apache process

mod_ndb.h

```
struct mod_ndb_connection {
    unsigned int connected;
    int ndb_force_send;
    Ndb_cluster_connection *connection;
    ndb_instance **instances;
    struct mod_ndb_connection *next;
};
typedef struct mod_ndb_connection ndb_connection;
```



One mod_ndb_connection per NDB connect string



One ndb_instance per Apache thread, per NDB connect string

mod_ndb.h

```
struct mod_ndb_instance {
  struct mod_ndb_connection *conn;
  Ndb *db;
  NdbTransaction *tx;
  int n_read_ops;
  int max_read_ops;
  struct data_operation *data;
  struct {
    unsigned int has_blob : 1 ;
    unsigned int aborted  : 1 ;
    unsigned int use_etag : 1 ;
  } flag;
  unsigned int requests;
  unsigned int errors;
};

typedef struct mod_ndb_instance
   ndb_instance;
```

Using C++ class templates above the Apache API

Apache's C-language API relies heavily on void pointers that you can cast to different data types. In C++, though, casting is no fun – the compiler requires you to make every cast explicitly, and casting defeats the type-safe design of the language.

Here are some examples from the array API: array_header->elts is a char * which you cast to an array pointer, and ap_push_array() returns a void pointer to a new element.

httpd/ap_alloc.h

```
typedef struct {
    ap_pool *pool;
    int elt_size;
    int nelts;
    int nalloc;
    char *elts;
} array_header;
```

```
array_header * ap_make_array(pool *p, int nelts, int elt_size);

void * ap_push_array(array_header *);
```

---

mod_ndb.h

```
template <class T>
class apache_array: public array_header {
  public:
    int size()    { return this->nelts; }
    T **handle()  { return (T**) &(this->elts); }
    T *items()    { return (T*) this->elts; }
    T &item(int n){ return ((T*) this->elts)[n]; }
    T *new_item() { return (T*) ap_push_array(this); }
    void * operator new(size_t, ap_pool *p, int n) {
      return ap_make_array(p, n, sizeof(T));
    };
};
```

In mod_ndb, the template apache_array<T> builds a subclass of array_header to manage an array of any type. All of the casting is done here in the template definition, so the code in the actual source files is cleaner:

```
dir->visible     = new(p, 4) apache_array<char *>;
dir->updatable   = new(p, 4) apache_array<char *>;
dir->indexes     = new(p, 2) apache_array<config::index>;


*dir->visible->new_item() = ap_pstrdup(cmd->pool, arg);
```

Per-server (i.e. per-VHOST) config structure

config::srv
connect_string
max_read_operations

```
struct srv {
  char *connect_string;
  int max_read_operations;
};
```

Apache per-directory config structure

config::dir
database
table
pathinfo_size
pathinfo
allow_delete
use_etags
results
sub_results
format_param[]
incr_prefetch
flag.pathinfo_always
flag.has_filters
visible
updatable
indexes
key_columns

```
/* Apache per-directory configuration */
  struct dir {
    char *database;
    char *table;
    int pathinfo_size;
    short *pathinfo;
    int allow_delete;
    int use_etags;
    result_format_type results;
    result_format_type sub_results;
    char *format_param[2];
    int incr_prefetch;
    struct {
      unsigned pathinfo_always : 1;
      unsigned has_filters : 1;
    } flag;
    apache_array<char*> *visible;
    apache_array<char*> *updatable;
    apache_array<config::index> *indexes;
    apache_array<config::key_col> *key_columns;
  };
```

Configuration Directives

Directive	Function	Data Structure	Inheritable
ndb-connectstring	connectstring()	srv->connect_string	Yes
ndb-max-read-subrequests	maxreadsubrequests()	srv-> max_read_operations	Yes
Database	ap_set_string_slot()	dir->database	Yes
Table	ap_set_string_slot()	dir->table	Yes
Deletes	ap_set_flag_slot()	dir->allow_delete	Yes
Format	result_format()	dir->results	Yes
Columns	non_key_column()	dir->visible	No
AllowUpdate	non_key_column()	dir->updatable	No
PrimaryKey	primary_key()	dir->key_columns	No
UniqueIndex	named_index()	dir->key_columns	No
OrderedIndex	named_index()	dir->key_columns	No
PathInfo	pathinfo()	dir->pathinfo	No
Filter	filter()	dir->key_columns	No

Configuration: Indexes and key columns

config::index
name
type
n_columns
first_col_serial
first_col_idx

```
struct index {
    char *name;
    char type;
    unsigned short n_columns;
    short first_col_serial;
    short first_col;
};
```

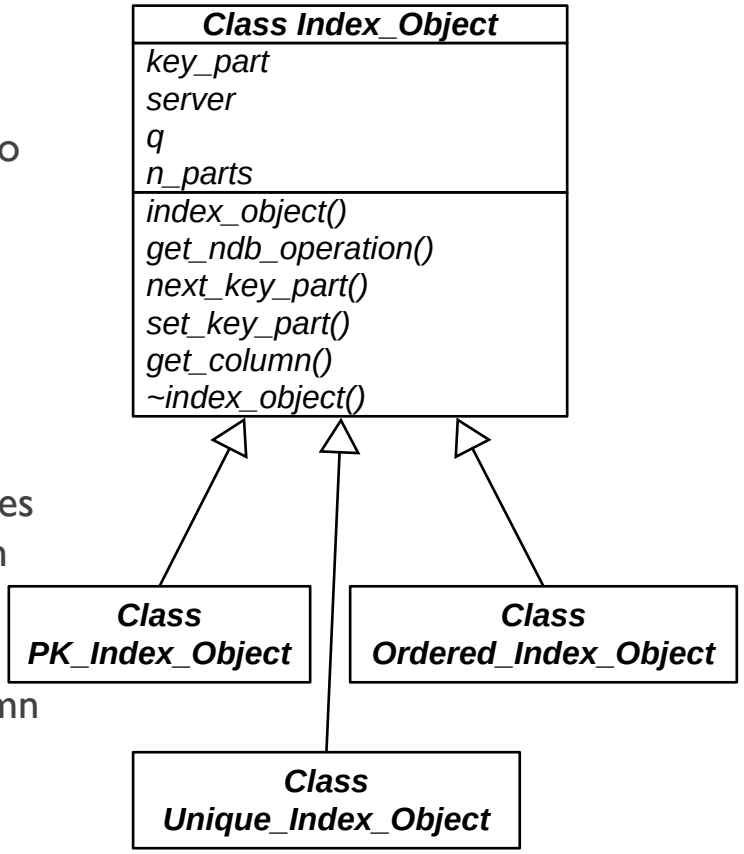
config::key_col
name
index_id
serial_no
idx_map_bucket
filter_col_serial
filter_col
next_in_key_serial
next_in_key
is.in_pk
is.filter
is.alias
is.in_ord_idx
is.in_hash_idx
is.in_pathinfo
filter_op
implied_plan

```
struct key_col {
    char *name;
    short index_id;
    short serial_no;
    short idx_map_bucket;
    short filter_col_serial;
    short filter_col;
    short next_in_key_serial;
    short next_in_key;
    struct {
      unsigned int in_pk       : 1;
      unsigned int filter      : 1;
      unsigned int alias       : 1;
      unsigned int in_ord_idx  : 1;
      unsigned int in_hash_idx : 1;
      unsigned int in_pathinfo : 1;
    } is;
    int filter_op;
    AccessPlan implied_plan;
};
```

```
/*
  Every time a new column is added, the columns get reshuffled some,
  so we have to fix all the mappings between serial numbers and
  actual column id numbers.

  The configuration API in Apache never gives the module a chance to
  "finalize" a configuration structure.  You never know when you're finished
  with a particular directory.  So, we run fix_all_columns() every time we
  create a new column, which, alas, does not scale too well.

  While processing the config file, the CPU time spent fixing columns grows
  with n-squared, the square of the number of columns.  This could be improved
  using config handling that was more complex (a container directive) or less
  user-friendly (an explicit "end" token).

  On the other hand, the design is optimized for handling queries at runtime,
  where some operations (e.g. following the list of columns that belong to an
  index) are constant, and the worst (looking up a column name in the columns
  table) grows at log n.

*/
```

Class index_object:
Standardizing index access in mod_ndb

The index_object class hierarchy is defined and implemented entirely in the file "index_object.h"



- get_ndb_operation() is a single interface to getNdbOperation, getNdbIndexOperation, and getNdbIndexScanOperation.

- set_key_part() is a single interface for op->equal() and scanop->setBound().

- next_key_part() is an iterator that advances the counter *key_part* and returns false when you reach the end of the key

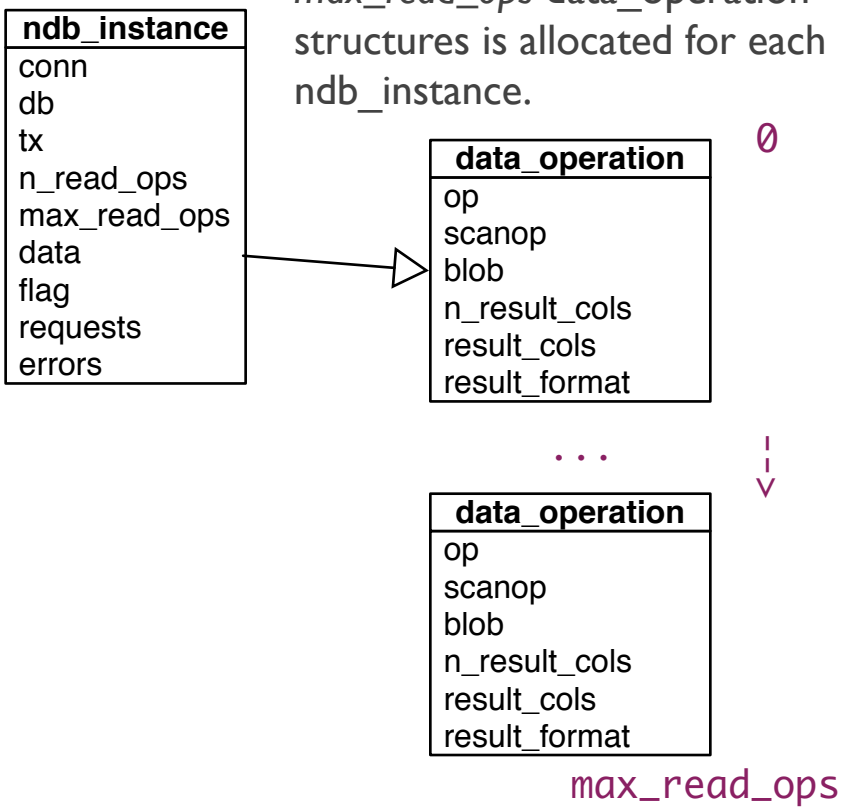
- get_column() maps a key part to its Column in the dictionary

```
class index_object {
public:
    int key_part;
    server_rec *server;
    struct QueryItems *q;
    int n_parts;

    index_object(struct QueryItems *queryitems, request_rec *r) {
      q = queryitems;
      server = r->server;
      key_part = 0;
    };
    virtual ~index_object() {};

    virtual NdbOperation *get_ndb_operation(NdbTransaction *) = 0;
    bool next_key_part() {  return (key_part++ < n_parts); };
    virtual int set_key_part(config::key_col &, mvalue &) = 0;
    virtual const NdbDictionary::Column *get_column() {
      return q->idx->getColumn(key_part);
    };
};
```

Transactions and Operations

mod_ndb.h

```
struct mod_ndb_instance {
  struct mod_ndb_connection *conn;
  Ndb *db;
  NdbTransaction *tx;
  int n_read_ops;
  int max_read_ops;
  struct data_operation *data;
  struct {
    unsigned int has_blob : 1 ;
    unsigned int aborted  : 1 ;
    unsigned int use_etag : 1 ;
  } flag;
  unsigned int requests;
  unsigned int errors;
};

typedef struct mod_ndb_instance
   ndb_instance;

/* An operation */
struct data_operation {
  NdbOperation *op;
  NdbIndexScanOperation *scanop;
  NdbBlob *blob;
  unsigned int n_result_cols;
  const NdbRecAttr **result_cols;
  result_format_type result_format;
};
```



At startup time, an array of *max_read_ops* data_operation structures is allocated for each ndb_instance.

Query.cc

Individual operations are processed in *Query.cc*. The Query() function uses the configuration and the query string to detrermine an "access plan" and create an appropriate NdbOperation.

In a simple (one-operation) request – but not a subrequest – control immediately passes into Execute.cc, and a single result set is returned to the client.

Execute.cc

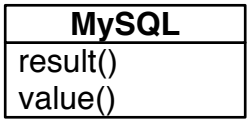
In Execute.cc, we execute the transaction (*ndb_instance::tx)* and then iterate through its read operations, collecting each result set. Each result set is stored as an Apache note.

The request into "/ndb-exec-batch" (the *execute handler)* as a subrequest goes directly into Execute.cc, executing all of the operations created in previous subrequests.

Encoding and decoding NDB & MySQL data types

MySQL_Field.h

```
namespace MySQL {
  void result(result_buffer &, const NdbRecAttr &);
  void  value(mvalue &, ap_pool *,
              const NdbDictionary::Column *,
              const char *);
};
```



Decoding

- result() is a generic "decode" function; it converts an NdbRecAttr to a printable ASCII value
- Decoding is handled by some private functions inside of MySQL_Field.cc, including String(), Time(), Date(), and Datetime()...
  - String() can unpack three different sorts of strings packed into NDB character arrays.

```
enum ndb_string_packing {
  char_fixed,
  char_var,
  char_longvar
};
```

- Time(), Date() and Datetime() decode specially packed mysql data types.

Encoding

- value() is a generic "encode" function; given an ASCII value (from HTTP) and an NdbDictionary::Column (which specifies how to encode the value), it will return an *mvalue* properly enocded for the database.

```
enum mvalue_use {
  can_not_use, use_char,
  use_signed, use_unsigned,
  use_64, use_unsigned_64,
  use_float, use_double,
  use_interpreted, use_null,
  use_autoinc
};

enum mvalue_interpreted {
  not_interpreted = 0,
  is_increment, is_decrement
};
```

mvalues

```
struct mvalue {
  const NdbDictionary::Column *ndb_column;
  union {
    const char *        val_const_char;
    char *              val_char;
    int                 val_signed;
    unsigned int        val_unsigned;
    time_t              val_time;
    long long           val_64;
    unsigned long long  val_unsigned_64;
    float               val_float;
    double              val_double;
    const NdbDictionary::Column * err_col;
  } u;
  size_t len;
  mvalue_use use_value;
  mvalue_interpreted interpreted;
};
typedef struct mvalue mvalue;
```

Formatting of Results

Results can be formatted in a variety of ways:

mod_ndb.h

```
enum result_format
{
  no_results = 0,
  json,
  raw,
  xml
}
```

A result_buffer is a memory region maintained by mod_ndb (and C++), using malloc(), realloc(), and free(). The rbuf.out() method uses realloc() to expand the buffer as needed.

result_buffer.h

```
class result_buffer {
private:
  size_t alloc_sz;

public:
  char *buff;
  size_t sz;
  char *init(request_rec *, size_t );
  void out(const char *fmt, ...);
  void out(size_t, const char *);
  ~result_buffer();
};
```

JSON Result Formatting

JSON.h

```
class JSON {
  public:
    inline static void new_array(result_buffer &rbuf)  { rbuf.out(2,"[\n"); }
    inline static void end_array(result_buffer &rbuf)  { rbuf.out(2,"\n]"); }
    inline static void new_object(result_buffer &rbuf) { rbuf.out(3," { "); }
    inline static void end_object(result_buffer &rbuf) { rbuf.out(2," }") ; }
    inline static void delimiter(result_buffer &rbuf)  { rbuf.out(3," , "); }
    inline static void is(result_buffer &rbuf)         { rbuf.out(3," : "); }

    inline static void put_member(result_buffer &rbuf, const NdbRecAttr &rec)
    {
      rbuf.out("\"%s\"", rec.getColumn()->getName());
      JSON::is(rbuf);
      JSON::put_value(rbuf, rec);
    }
    static void put_value(result_buffer &, const NdbRecAttr &);
};
```

JSON::put_value() – in JSON.cc – is largely a wrapper around MySQL::result(), but strings, dates, and times are all quoted, and NULLs are represented as `"null"`