mod_ndb:

A REST Web Services API for MySQL Cluster

Design goals

- Build a database server that conforms to HTTP 1.1.

- Have a lock-free design, with no mutexes in the mod_ndb code.

- Build mod_ndb for multiple versions of Apache, MySQL, and NDB from a single source tree.

- Do as much work as possible when processing the configuration file, and as little as possible when servicing a request.

- Be able to process configuration files without connecting to a cluster or using the NDB Data Dictionary.

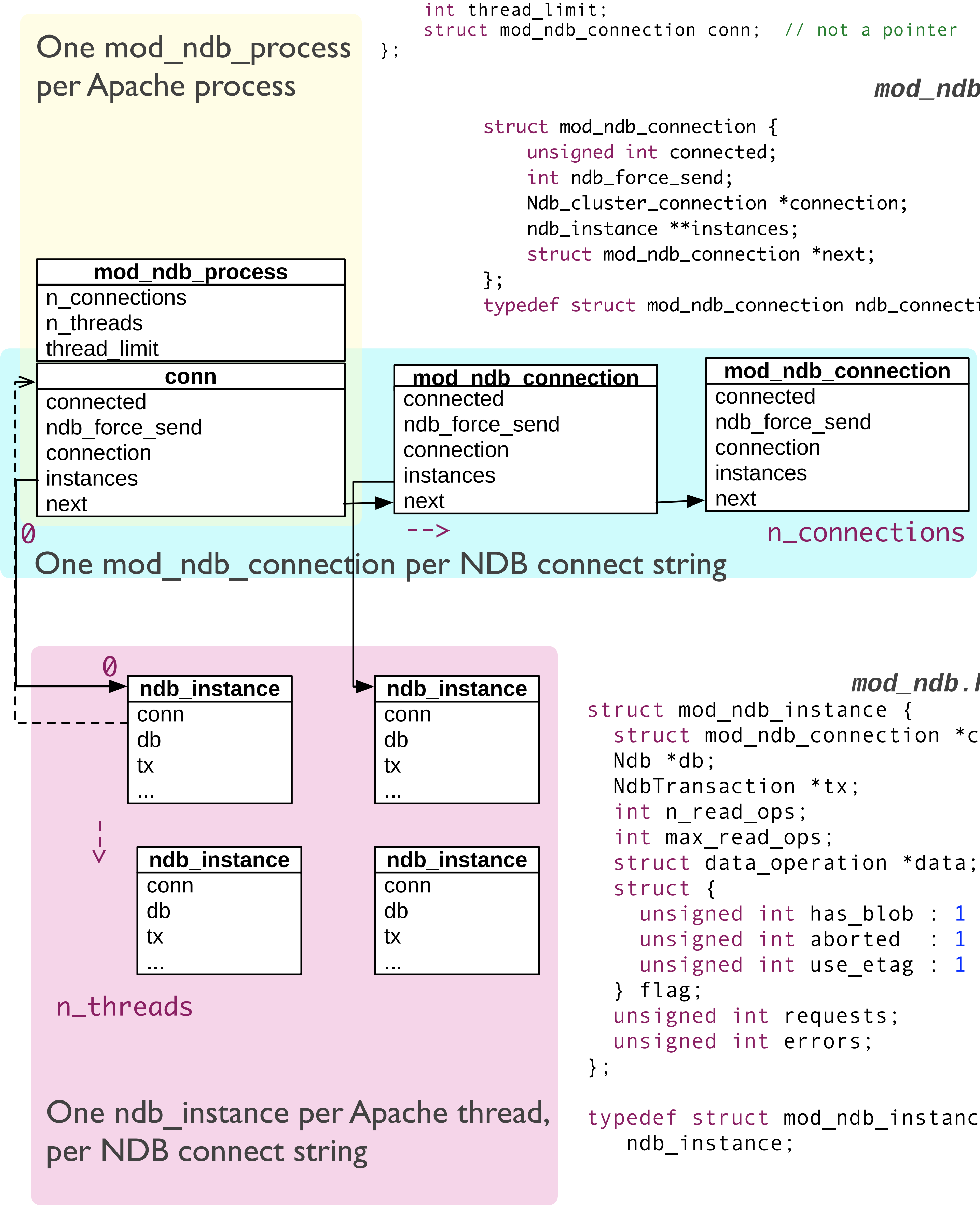
Apache processes and threads in mod_ndb

mod_ndb.h

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

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_process per Apache process

mod_ndb_process
n_connections
n_threads
thread_limit

conn
connected
ndb_force_send
connection
instances
next

mod_ndb_connection
connected
ndb_force_send
connection
instances
next

mod_ndb_connection
connected
ndb_force_send
connection
instances
next

0 --> n_connections

One mod_ndb_connection per NDB connect string

0

ndb_instance
conn
db
tx
...

ndb_instance
conn
db
tx
...

ndb_instance
conn
db
tx
...

ndb_instance
conn
db
tx
...

n_threads

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;
```

Some basics of query execution

- In the configuration for an endpoint, all of the "key columns" -- parameters like "id=4" and "year=2000" that may appear in the query string -- are stored in a sorted list. When the parameters are read from *r->args*, we use a binary search to find each parameter in the key columns.

- Besides named parameters, key columns can also be passed in *r->path_info,* as in the example *http://server/ndb/mytable/2000/4*. Pathinfo configuration is stored as a mapping from the position in the path_info string to the key column's index number in the sorted list – so the value gets associated with a named key column *without* having to use the binary sort.

- Once a key_column is found, *set_key()* in Query.cc determines how to use it. Either it is part of a filter, or it belongs to an index and therefore implies an access plan. If the implied plan is better than the current plan, then use it:

```
if(keycol.implied_plan > q->plan) {
  q->plan = keycol.implied_plan;
  q->active_index = keycol.index_id;
}
```

- The request body – *i.e.* the data sent with a POST request – is handled differently. When the body is read (in *read_http_post.cc)*, the names and values are stored in an apache table, *q->form_data*. Later, in *set_up_write(),* we iterate over the list of updatable columns *dir->updatable* and retrieve each column's new value (if any) from *q->form_data* using *ap_table_get()*.

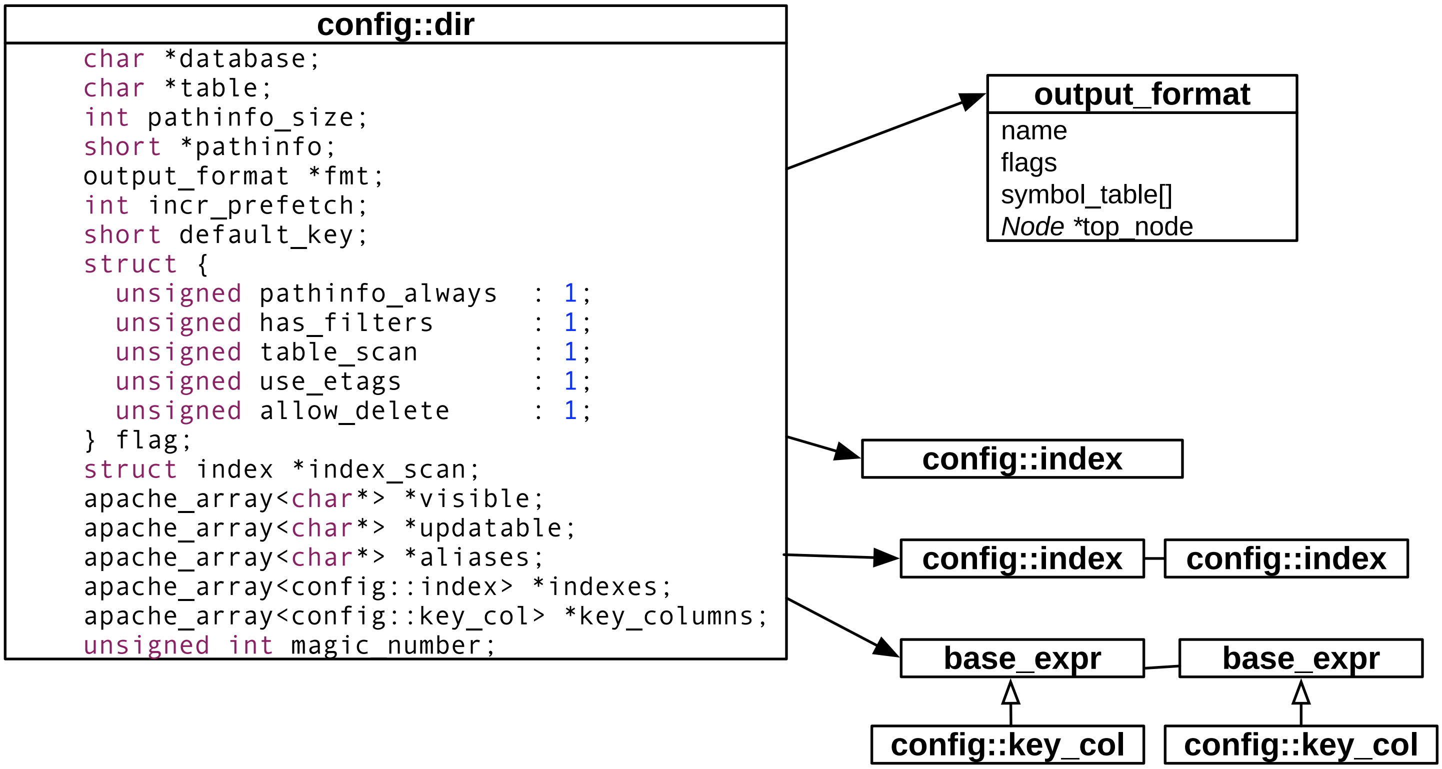
When multipart/form-data is supported, this might change.

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

config::srv

```
char *connect_string;
int max_read_operations;
unsigned int max_retry_ms;
unsigned int force_restart;
unsigned int magic_number;
```

Per-directory configuration structure

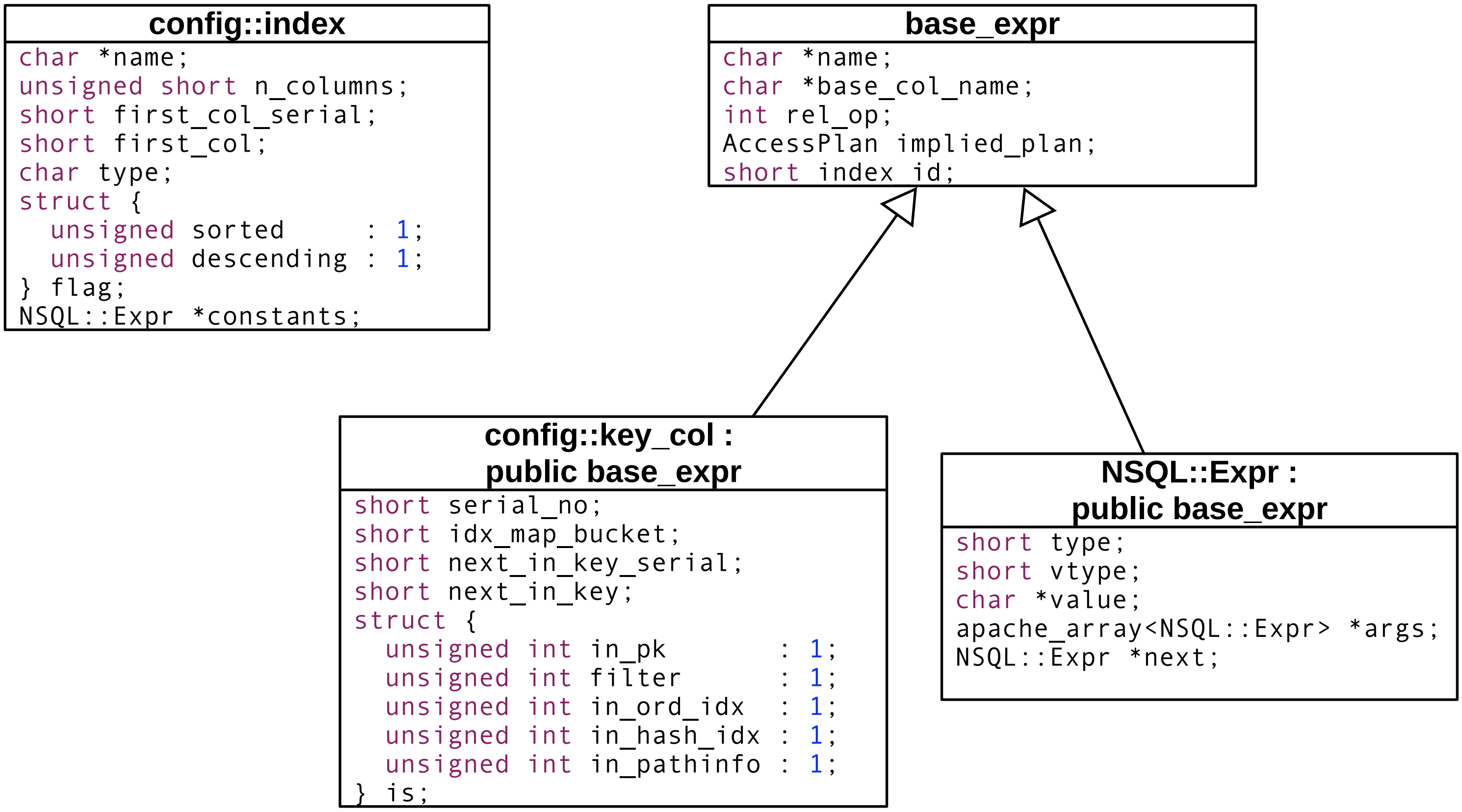


config::dir

```
    char *database;
    char *table;
    int pathinfo_size;
    short *pathinfo;
    output_format *fmt;
    int incr_prefetch;
    short default_key;
    struct {
      unsigned pathinfo_always  : 1;
      unsigned has_filters      : 1;
      unsigned table_scan       : 1;
      unsigned use_etags        : 1;
      unsigned allow_delete     : 1;
    } flag;
    struct index *index_scan;
    apache_array<char*> *visible;
    apache_array<char*> *updatable;
    apache_array<char*> *aliases;
    apache_array<config::index> *indexes;
    apache_array<config::key_col> *key_columns;
    unsigned int magic_number;
```

Configuration: Indexes and key columns

mod_ndb_config.h



config::index

```
char *name;
unsigned short n_columns;
short first_col_serial;
short first_col;
char type;
struct {
  unsigned sorted     : 1;
  unsigned descending : 1;
} flag;
NSQL::Expr *constants;
```

base_expr

```
char *name;
char *base_col_name;
int rel_op;
AccessPlan implied_plan;
short index_id;
```

config::key_col : public base_expr

```
short serial_no;
short idx_map_bucket;
short next_in_key_serial;
short next_in_key;
struct {
  unsigned int in_pk        : 1;
  unsigned int filter       : 1;
  unsigned int in_ord_idx   : 1;
  unsigned int in_hash_idx  : 1;
  unsigned int in_pathinfo  : 1;
} is;
```

NSQL::Expr : public base_expr

```
short type;
short vtype;
char *value;
apache_array<NSQL::Expr> *args;
NSQL::Expr *next;
```

```
/*
  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.

*/
```

N-SQL

The N-SQL language is built using the Coco/R C++ compiler generator from http://www.ssw.uni-linz.ac.at/coco/ -- all basic configuration in the parser is implemented by calls in to the older configuration routines in *config.cc*

The syntax of Apache-style configuration maps very closely to mod_ndb's internal data structres.   The parser is fundamentally a simple translator from the more flexible syntax of SQL to those same structures and configuration routines.

Because mod_ndb has no optimizer, and no access to the data dictionary at configuration time, the parser cannot handle standard SQL. N-SQL still requires the administrator to specify which index to use for every query.

A few features are supported by the N-SQL parser that are not available using Apache-style configuration:

- Non-equal bounds for ordered index scans (less than, greater than, etc.)
- Use of string and numeric *constants* in WHERE clauses

These are represented by the NSQL::Expr object, a cousin of the key_column.

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);
```

Class index_object:
Standardizing index access in mod_ndb

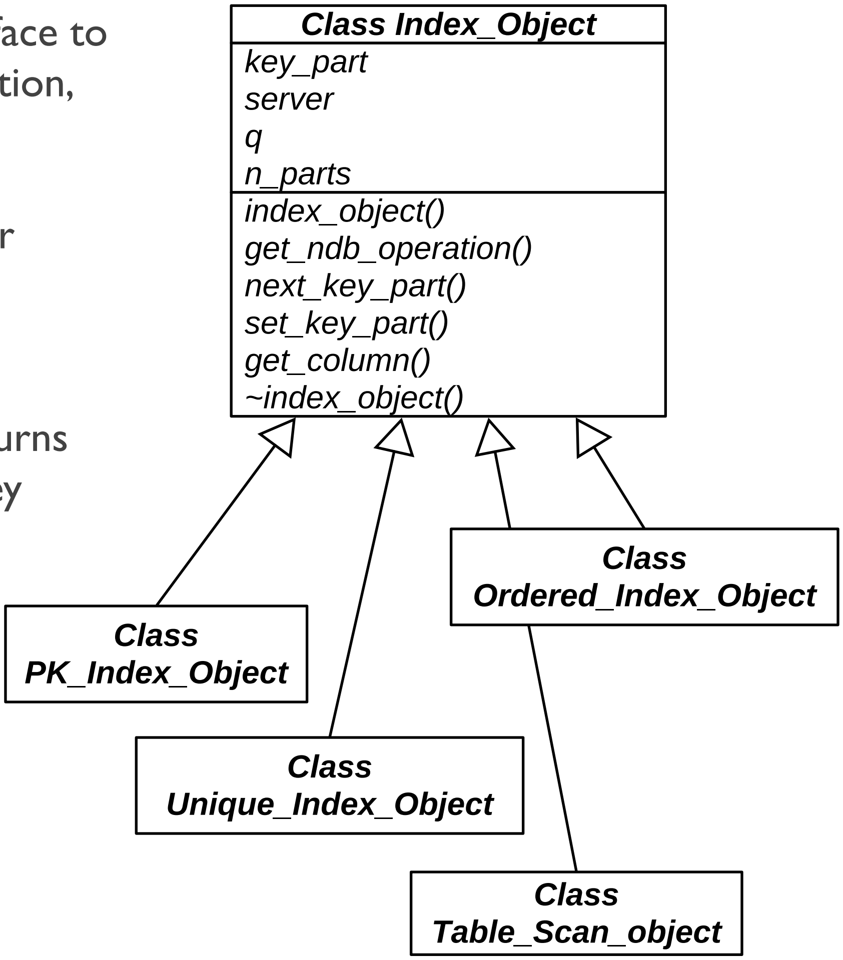
The index_object class hierarchy is defined and implemented entirely in the file "index_object.h"

- 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

key_part
server
q
n_parts

index_object()
get_ndb_operation()
next_key_part()
set_key_part()
get_column()
~index_object()

Class PK_Index_Object

Class Unique_Index_Object

Class Ordered_Index_Object

Class Table_Scan_object

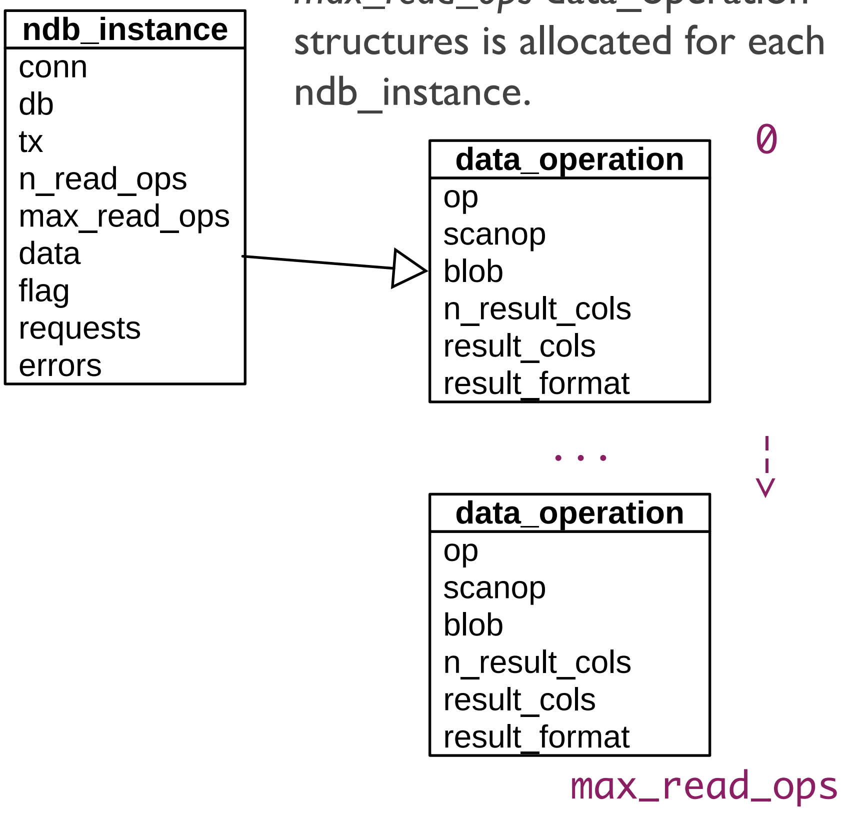
```
class index_object {
  protected:
    int key_part;
    server_rec *server;
    struct QueryItems *q;
    int n_parts;
    int set_key_num(int num, mvalue &mval);

  public:
    index_object(struct QueryItems *queryitems, request_rec *r);
    virtual ~index_object();

    virtual NdbOperation *get_ndb_operation(NdbTransaction *);
    virtual bool next_key_part();
    virtual const NdbDictionary::Column *get_column(base_expr &);
    virtual int set_key_part(int, mvalue &mval);
};
```

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.



ndb_instance
conn
db
tx
n_read_ops
max_read_ops
data
flag
requests
errors

0

data_operation
op
scanop
blob
n_result_cols
result_cols
result_format

. . .

data_operation
op
scanop
blob
n_result_cols
result_cols
result_format

max_read_ops

`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 subrequest, processing ends after Query(), but in a complete request it passes immediately into ExecuteAll().

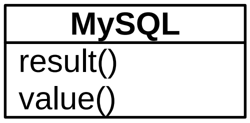
`Execute.cc`

In ExecuteAll() *(Execute.cc)*, we execute the transaction and then collect and format the results. In an ordinary request, a single result page is sent to the client. In a subreqest, though, the final call into "/ndb-exec-batch" (the *execute handler)* calls directly into Execute.cc, executes the transaction, and iterates over the all the operations (from 0 to n_read_ops), storing the results in the Apache notes table.

Encoding and decoding NDB & MySQL data types

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

MySQL_Field.h



MySQL

result()

value()

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.

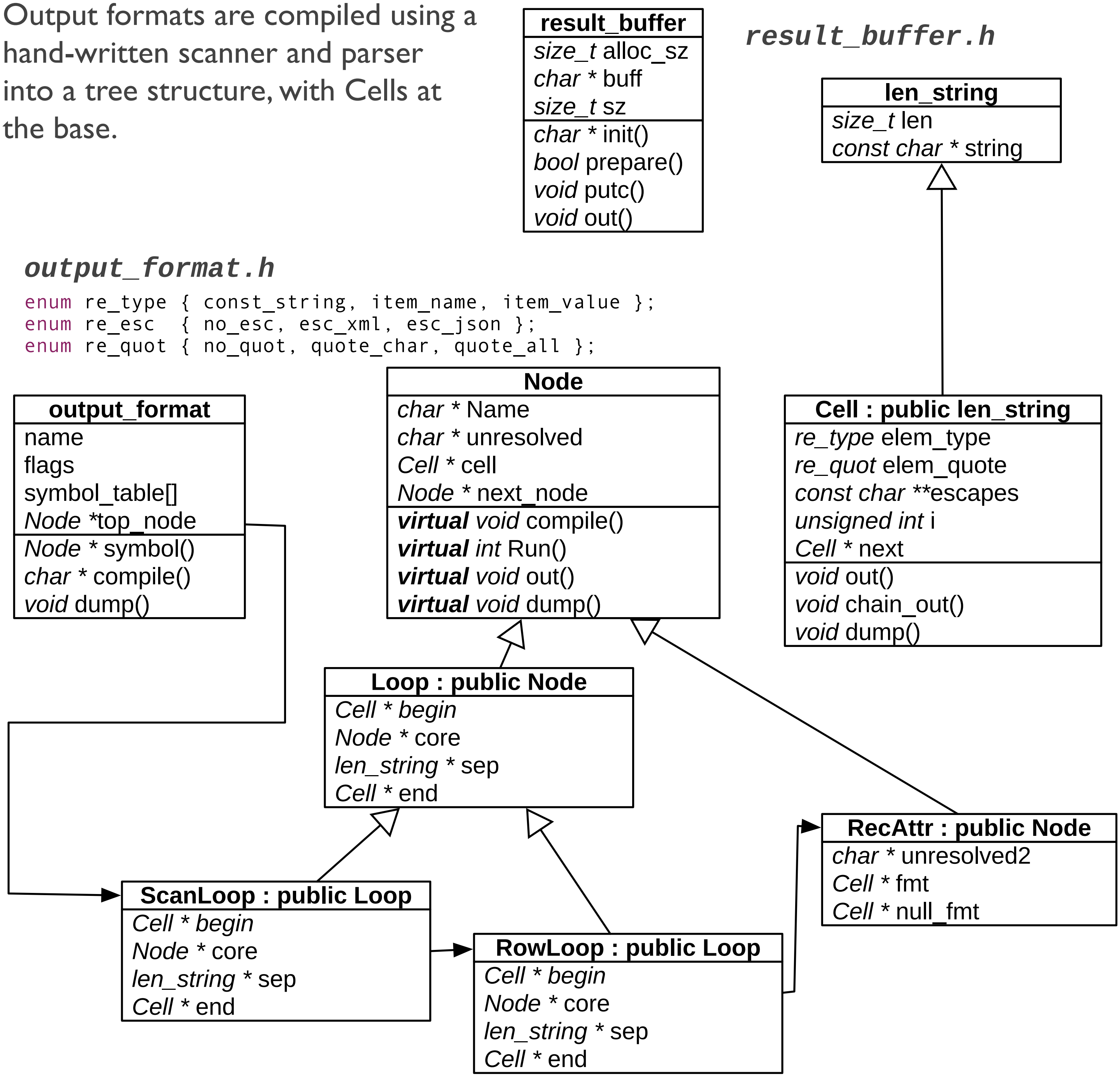
```
enum mvalue_use {
  can_not_use, use_char,
  use_signed, use_unsigned,
  use_64, use_unsigned_64,
  use_float, use_double,
  use_interpreted, use_null,
  use_autoinc
};

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;
```

Output Formats and Result Buffers

Output formats are compiled using a hand-written scanner and parser into a tree structure, with Cells at the base.



result_buffer.h

result_buffer
size_t alloc_sz
char * buff
size_t sz
char * init()
bool prepare()
void putc()
void out()

len_string
size_t len
const char * string

output_format.h

```
enum re_type { const_string, item_name, item_value };
enum re_esc  { no_esc, esc_xml, esc_json };
enum re_quot { no_quot, quote_char, quote_all };
```

output_format
name
flags
symbol_table[]
Node *top_node
Node * symbol()
char * compile()
void dump()

Node
char * Name
char * unresolved
Cell * cell
Node * next_node
virtual *void* compile()
virtual *int* Run()
virtual *void* out()
virtual *void* dump()

Cell : public len_string
re_type elem_type
re_quot elem_quote
const char **escapes
unsigned int i
Cell * next
void out()
void chain_out()
void dump()

Loop : public Node
*Cell * begin*
Node * core
len_string * sep
Cell * end

RecAttr : public Node
char * unresolved2
Cell * fmt
Cell * null_fmt

ScanLoop : public Loop
*Cell * begin*
Node * core
len_string * sep
Cell * end

RowLoop : public Loop
*Cell * begin*
Node * core
len_string * sep
Cell * end

output_format.cc

```
int build_results(request_rec *r, data_operation *data, result_buffer &res) {
  output_format *fmt = data->fmt;
  int result_code;

  if(fmt->flag.is_raw) return Results_raw(r, data, res);
  res.init(r, 8192);
  for(Node *N = fmt->top_node; N != 0 ; N=N->next_node) {
    result_code = N->Run(data, res);
    if(result_code != OK) return result_code;
  }
  return OK;
}
```

In *build_results()*, a query result is built by running the nodes of the output format.