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.

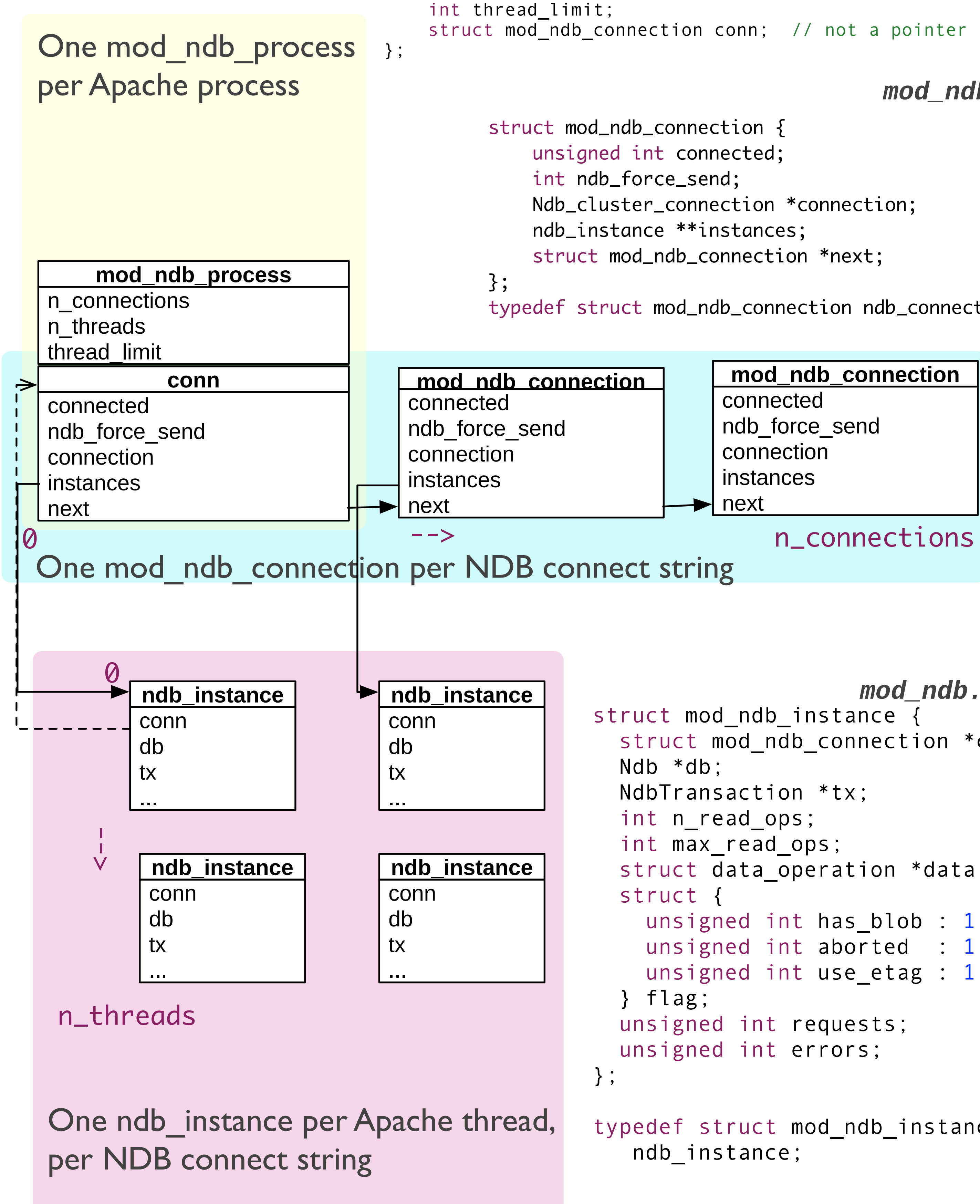
Apache processes and threads in mod_ndb

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;
```



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 to 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 *request_body.cc)*, the names and values are stored in an apache table, *q->form_data*. Later, in *set_up_write() (Query.cc),* 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.

Configuration *mod_ndb_config.h*

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
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 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.

*/
```

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*

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

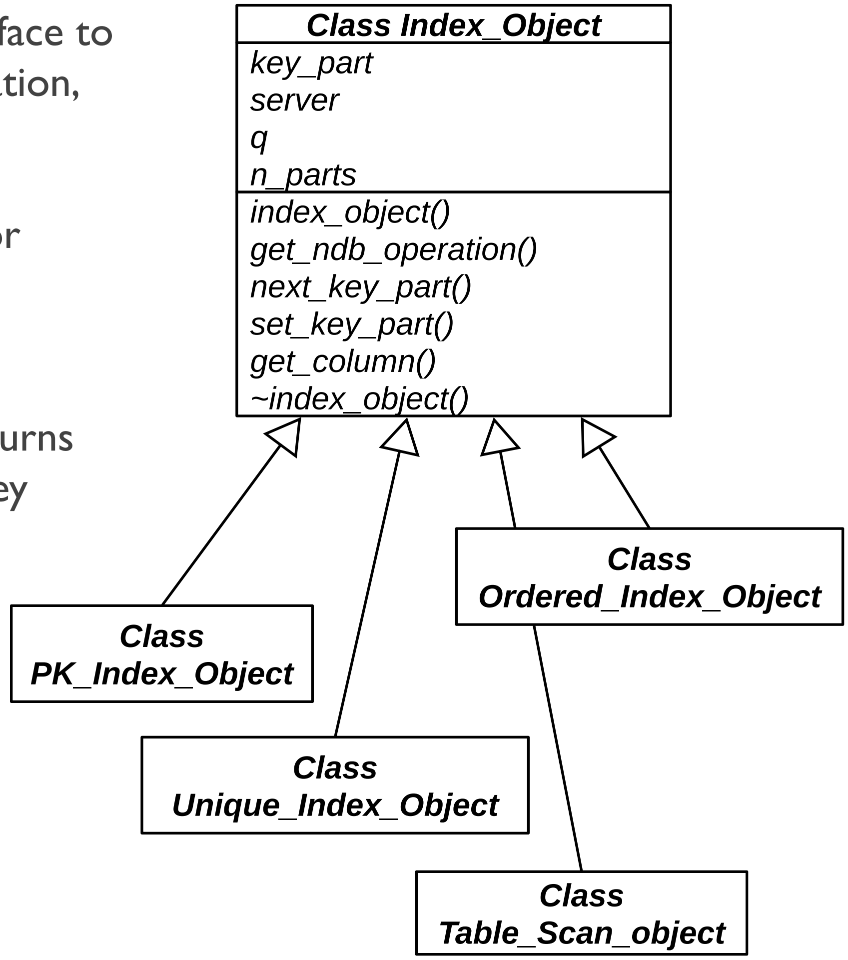
The index_object class hierarchy is defined and implemented entirely in the file "index_object.h"

- 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 {
  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);
};
```

Reading the HTTP Request Body

request_body.cc

• The body of an HTTP request contains the POST or PUT data. It is encoded as specified by the request's Content-type header.

• Mod_ndb 1.0 supports the *application/x-www-form-urlencoded* type of request body. The original *util_read()* code and some other parts of *request_body.cc* were written by Lincoln Stein & Doug MacEachern for mod_perl and published without copyright or restriction.

• Mod_ndb 1.1 adds support for *application/jsonrequest,* as proposed at www.json.org/JSONRequest.html. The JSON parser is implemented using Coco.

• In both cases, the request is presented to mod_ndb as an Apache table.

```
/*
  What gets copied?

  In util_read() the request body is copied from various network buffers
  into a single ap_pcalloc() buffer.

  In read_urlencoded(), ap_getword copies each token into an ap_palloc() buffer.
  ap_table_merge() was making another copy of each token, but now we use
  ap_table_mergen().

  The JSON scanner is initialized with the util_read buffer and its length.
  Coco's Buffer() originally made a copy of the buffer, but I edited
  Scanner.frame to stop that.

  Coco's Buffer::Read() reads an 8-bit char buf[pos] and returns it as a
  32-bit int; everything higher in Coco -- the Scanner and Parser -- treats it
  as a 32-bit wide-char.  Token->val is an array of wchar_t allocated off of a
  HeapBlock, which is maintained by the scanner (and freed in ~Scanner()).

  Coco's token heap is the second copy -- equivalent to the ap_getword() copy
  in the urlencoded case.  But I can't use it; it's an array of wchar_t and
  it still needs to be unquoted and unescaped.  So JSON_string() makes the
  third copy (with escapes) and then unescapes the string in place.
*/
```

Transactions and Operations

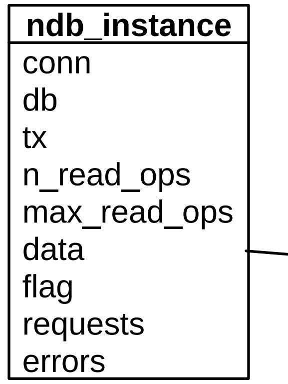
mod_ndb.h

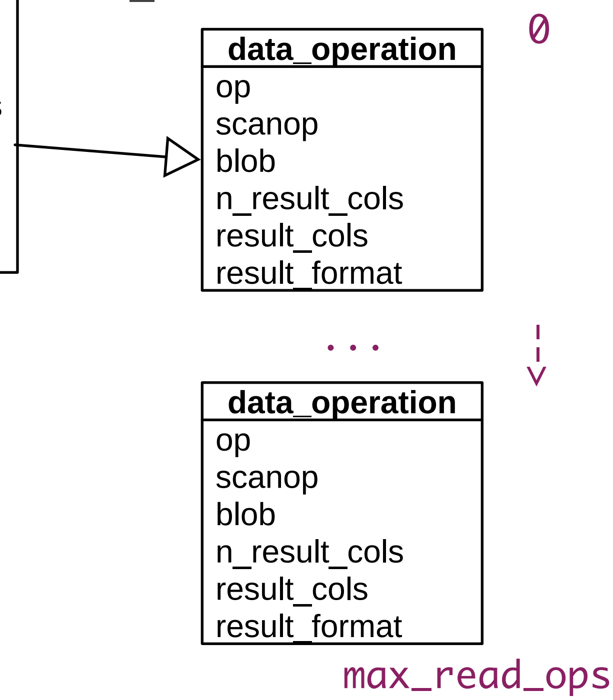
```
struct mod_ndb_instance {
  struct mod_ndb_connection *conn;
  Ndb *db;
  NdbTransaction *tx;
  int n_read_ops;
  int max_read_ops;
  struct data_operation *data;
  struct {
    unsigned int has_blob : 1 ;
    unsigned int aborted  : 1 ;
    unsigned int use_etag : 1 ;
  } flag;
  unsigned int requests;
  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 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 {
  class result;
  void  value(mvalue &, ap_pool *,
              const NdbDictionary::Column *,
              const char *);
};
```

Decoding

MySQL_result.h

- The MySQL::result class represents a value provided by the NDB API ; it provides a common interface to both NdbRecAttr and NdbBlob. MySQL::result::out() provides a text representation of the result.
- Actual decoding is handled by some private functions inside of MySQL_result.cc ...
  - 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

MySQL_value.h

- 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;
```

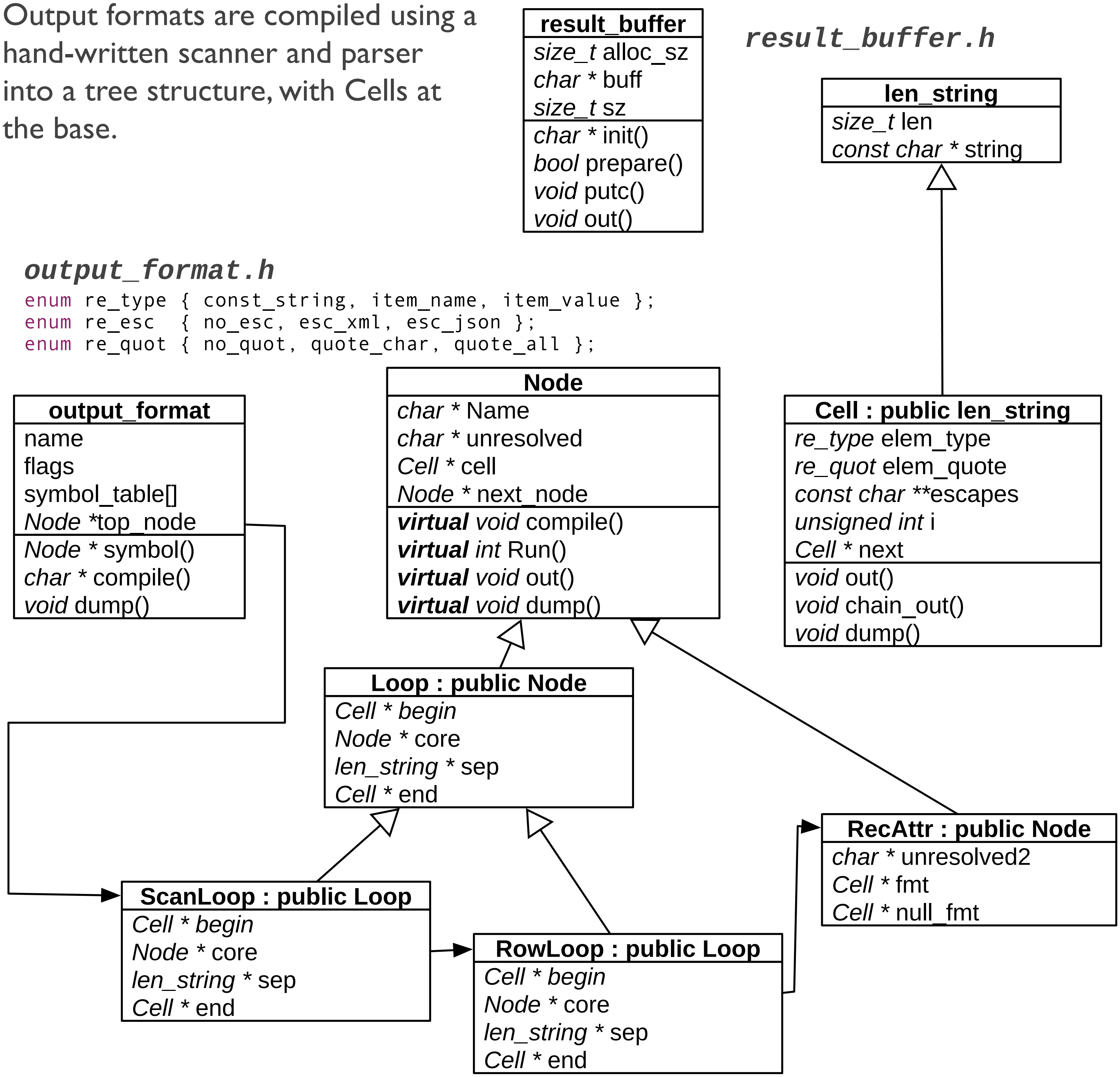
Output Formats and Result Buffers

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

result_buffer.h

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.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.