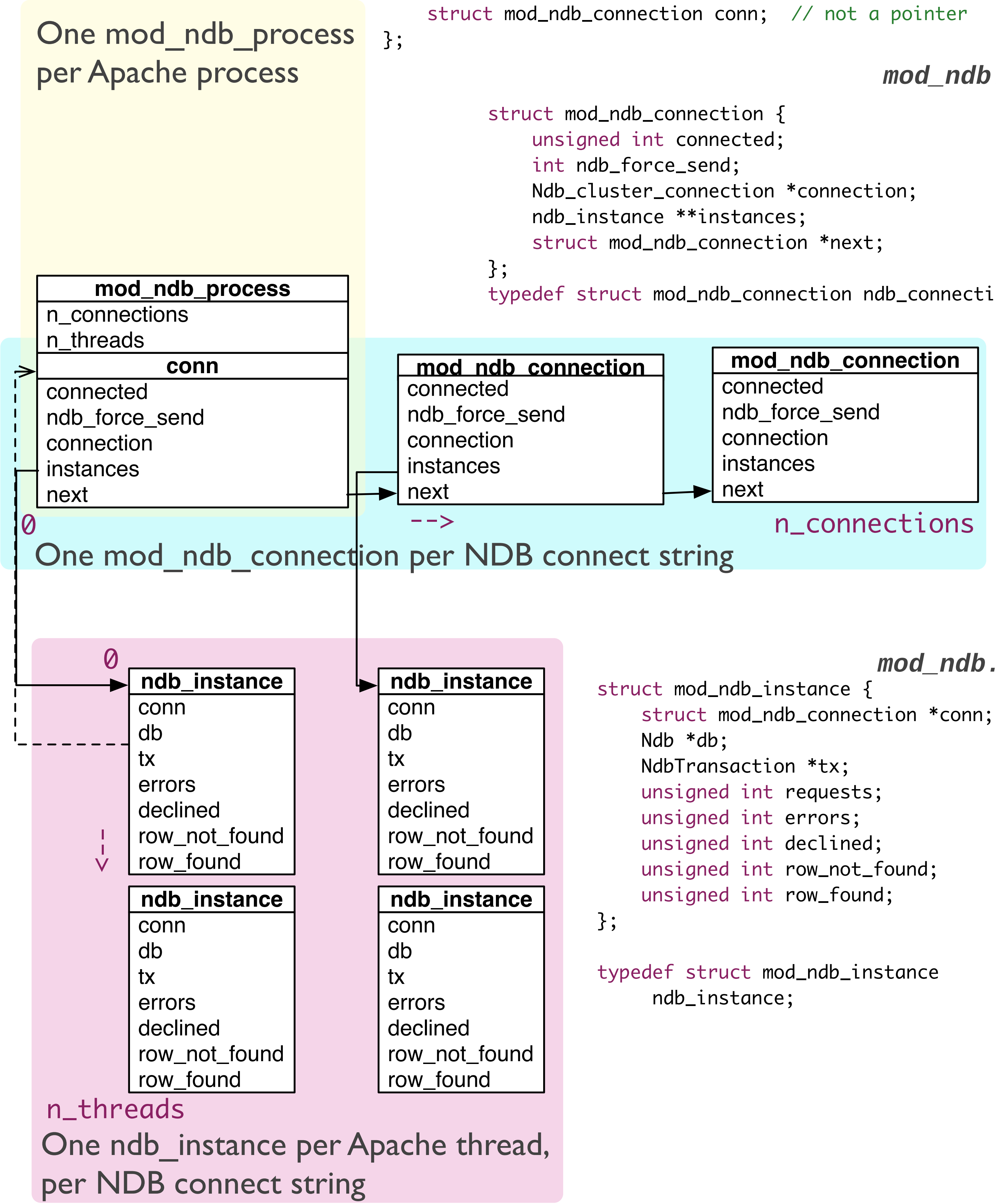
Apache processes and threads in mod_ndb

mod_ndb.h

```
struct mod_ndb_process {
    unsigned short n_connections;
    unsigned short n_threads;
    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
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
errors
declined
row_not_found
row_found

ndb_instance
conn
db
tx
errors
declined
row_not_found
row_found

ndb_instance
conn
db
tx
errors
declined
row_not_found
row_found

ndb_instance
conn
db
tx
errors
declined
row_not_found
row_found

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;
    unsigned int requests;
    unsigned int errors;
    unsigned int declined;
    unsigned int row_not_found;
    unsigned int row_found;
};

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);
```

Configuration

mod_ndb_config.h

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

config::srv
connect_string

```
namespace config {
  /* Apache per-server configuration  */
   struct srv {
     char *connect_string; };
}
```

Apache per-directory config structure

config::dir
database
table
pathinfo_size
pathinfo
allow_delete
results
format_param[]
visible
updatable
indexes
key_columns

```
/* Apache per-directory configuration */
namespace config {
  struct dir {
    char *database;
    char *table;
    int pathinfo_size;
    short *pathinfo;
    int allow_delete;
    result_format results;
    char *format_param[2];
    apache_array<char*> *visible;
    apache_array<char*> *updatable;
    apache_array<config::index> *indexes;
    apache_array<config::key_col> *key_columns;
  };
}
```

Configuration Directives

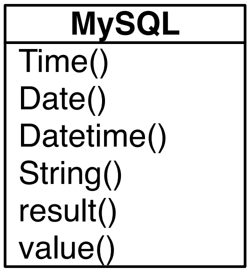
mod_ndb.cc and config.cc

Directive	**Function**	**Data Structure**	**Inheritable**
ndb-connectstring	ap_set_string_slot()	srv->connect_string	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

mod_ndb Architecture

MySQL_Field.h

encoding and decoding of NDB & MySQL data types



MySQL

Time()
Date()
Datetime()
String()
result()
value()

```
namespace MySQL {
  char *Time(pool *p, const NdbRecAttr &rec);
  char *Date(pool *p, const NdbRecAttr &rec);
  char *Datetime(pool *p, const NdbRecAttr &rec);
  char *String(pool *p, const NdbRecAttr &rec, enum ndb_string_packing packing);
  char *result(pool *p,  const NdbRecAttr &rec);
  mvalue value(pool *p, const NdbDictionary::Column *col, const char *val);
};
```

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

- String() can unpack three different sorts of strings packed into NDB character arrays.

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

- result() is a generic "decode" function; it converts an NdbRecAttr to a printable ASCII value

- 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

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

mod_ndb Architecture: Formatting of Results

Results can be formatted in a variety of ways

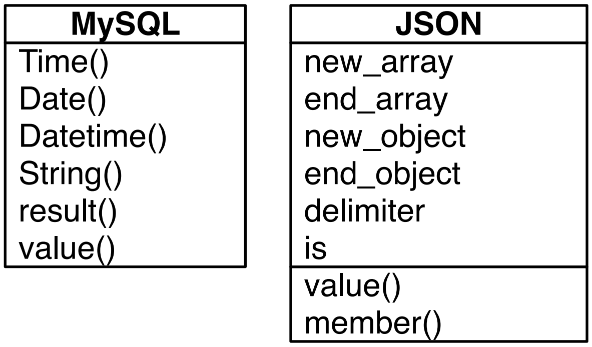
mod_ndb.h

```
enum result_format
{
  json = 1,
  raw,
  xml,
  ap_note
}
```

JSON Result Formatting

JSON.h

```
class JSON {
  public:
    static const char * new_array;     //  "[\n"
    static const char * end_array;     //  "]\n"
    static const char * new_object;    //  "}\n"
    static const char * end_object;    //  "}\n"
    static const char * delimiter ;    //  " , "
    static const char * is        ;    //  " : "
    static char *value(const NdbRecAttr &rec, request_rec *r);
    inline static char *member(const NdbRecAttr &rec, request_rec *r) {
      return ap_pstrcat(r->pool,
                        rec.getColumn()->getName(),
                        JSON::is,
                        JSON::value(rec,r),
                        NULL);
    }
};
```



MySQL

Time()
Date()
Datetime()
String()
result()
value()

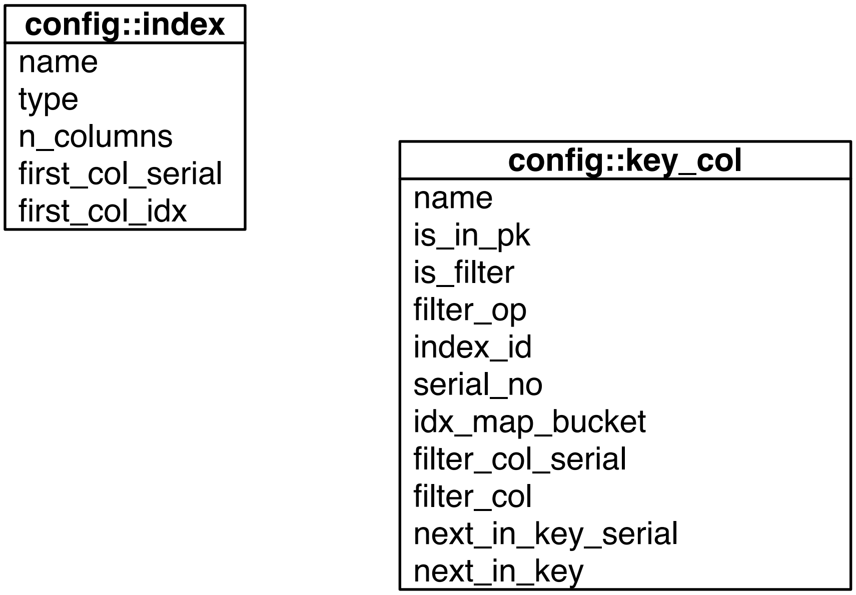
JSON

new_array
end_array
new_object
end_object
delimiter
is

value()
member()

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

Indexes and key columns



config::index

name
type
n_columns
first_col_serial
first_col_idx

config::key_col

name
is_in_pk
is_filter
filter_op
index_id
serial_no
idx_map_bucket
filter_col_serial
filter_col
next_in_key_serial
next_in_key

```
/*
  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.

*/
```