6 Secondary Indexes

This section describes how tables are sorted by indexes using the example of a `voteinfo` table, as well as instructions on how to add an index to a table using the example of an `addressbook` contract.

Table and Index Format

EOS creates a structure for each index that returns a value. In CyberWay, to describe the index, either a separate function is created and `const_mem_fun` is used in the table description, or a field from the `eosio::member` structure is used.

Table format requirement

The table description should conform to the following format:

```
using vote_table = multi_index<N(vote), structures::voteinfo, vote_id_index,
```

Parameters:
`N(vote)` — a primary index;
`vote_table` — a table name;
`voteinfo` — a structure name;
`vote_id_index` — a primary key index;
`vote_messageid_index` — an index identifying the message. For each vote, a message is formed during the voting, which is assigned a number. By this number, a message is searched in the table;
`vote_group_index` — secondary index, composite.

The table should always include:

- primary index (`uint64` type);
- a certain number of secondary indexes, which can be described both by one field using `by_message` , and by several fields.

The description of the secondary index for the field is the following:

```
    uint64_t by_message() const {
        return message_id;
    }
```

The following parameters are indicated when describing a secondary index in several fields:

- index template called `by_` ;
- index name;
- index description:
  - `const_mem_fun` — description in a simple form;
  - `eosio::member` — description in a complex form.

When describing the secondary index in a complex form of `eosio::member` , the `composite_key` , the name of the structure, and the list of fields from which the index is formed are indicated (for example, from the `message_id` and `voter` fields).

```
    eosio::member<structures::voteinfo, uint64_t, &structures::voteinfo::
    eosio::member<structures::voteinfo, name, &structures::voteinfo::vote
```

Requirement for the description of secondary indexes

The description of the secondary indexes should coincide with their description in the ABI-file. In this case, the order of location as well as the type correspondence must be observed. Serialization and deserialization of the transmitted data in binary form must be performed in the same way.

Depending on the purpose of the secondary indexes, there are two ways to describe them:

- a description for work within the contract;

- a description for work outside the contract. In this case, the description of the table with the indexes is in the ABI file

Description of the secondary index for work within the contract

The description of the `name` and `info` fields is the same as in EOS. The difference from EOS is that instead of the index fields `indexed_name` and `indexed_type` used in EOS, CyberWay uses the description of the indexes.

Each individual index is an array in which a name is assigned to every position of elements. This name must match the name specified in the `indexed_by` template. The match is as follows:

- `N(id)` — a coincidence by the ordinal number of the parameter. For example, N1 is the choice of the primary index;
- `N(messageid)` — a match by a sequence number of a message. Each vote is assigned a message number during voting;
- `N(byvoter)` — a coincidence by ordinal number of the voter.

```
using vote_id_index = indexed_by<N(id), const_mem_fun<structures::voteinfo,

using vote_messageid_index = indexed_by<N(messageid), const_mem_fun<structu

using vote_group_index = indexed_by<N(byvoter), eosio::composite_key<structu
      eosio::member<structures::voteinfo, uint64_t, &structures::voteinfo::
      eosio::member<structures::voteinfo, name, &structures::voteinfo::vote
```

Description of the secondary index for work outside the contract

The concept of “unique” is introduced into the concept of the index. It is whether the value of this index for the record is unique or not. The value `unique = true` means uniqueness.

Sorting provides an ordered arrangement of indexes.

```
          "orders": [
              {"field": "id", "order": "asc"}
```

The «orders» field indicates all the fields that will be sorted. The table description for the smart contract is specified in the ABI file. There is an order in which the index fields are described (`messageid` and `voter`). The values of these fields must follow in the order in which they are contained in `composite_key` , namely, `eosio::member` or `const_mem_ func` . The second `order` attribute defines how to build an index within the database. This parameter determines the order of occurrence of elements if these elements are extracted from the table one by one, that is, determines a rule by which the next element in the table is located.

An example of describing a table with indexes in an ABI file:

```
  {
    "name": "vote",
    "type": "voteinfo",
    "indexes": [
       {
          "name": "primary",
          "unique": "true",
          "orders": [
              {"field": "id", "order": "asc"}
          ]
       },
       {
          "name": "messageid",
          "unique": "false",
          "orders": [
              {"field": "message_id", "order": "asc"}
          ]
       },
       {
          "name": "byvoter",
          "unique": "true",
          "orders": [
              {"field": "message_id", "order": "asc"},
              {"field": "voter", "order": "asc"}
          ]
       }
    ]
  },
```

An example of indexes used inside a smart contract

```
    auto votetable_index = vote_table.get_index<"messageid"_n>();
```

```
    auto vote_itr = votetable_index.lower_bound(mssg_itr->id);
    while ((vote_itr != votetable_index.end()) && (vote_itr->message_id == m
        vote_itr = votetable_index.erase(vote_itr);
```

As `messageid` consists of one field, the value `mssg_itr->id` which will be passed to lower_bound, will also consist of one field. `lower_bound` is used to search for the lower boundary by the specified ID. From the found lower limit, the index goes through all the votes, while the vote is considered to be cast for the same post.

An example of using a composite secondary index

```
auto votetable_index = vote_table.get_index<"byvoter"_n>();
    auto vote_itr = votetable_index.find(std::make_tuple(mssg_itr->id, voter
```

`byvoter` — consists of two fields passed by `make_tuple`. Used to search for votes related to the message `mssg_itr->id` of the `voter`, the user who voted.

Example of adding an index to the addressbook contract table

The previous section provides instructions for creating and setting up an `addressbook table`. The following ones are the instructions for adding an index to this table.

6.1 Delete existing entries from the table

The table structure cannot be changed if it contains data. You need to delete both `alice` and `bob` user entries in it.

```
cleos push action addressbook erase '["alice"]' -p alice@active
cleos push action addressbook erase '["bob"]' -p bob@active
```

6.2 Add a new index field to the addressbook.cpp contract

Since the secondary index must be a numeric field, a variable of type `uint64_t` is added.

```
uint64_t age;
```

6.3 Add a secondary index for the addresses table

The field is defined as a secondary index. You need to reconfigure the `address_index` table.

```
typedef eosio::multi_index<"people"_n, person,
indexed_by<"byage"_n, member<person, uint64_t, &person::get_secondary_1>>
 > address_index;
```

`index_by` — the structure that is used to create an instance of the index. `«byage»` — a function call operator that retrieves the value of const as an index key.

A table with the multiple indexes will will position records by the `age` variable.

```
indexed_by<"byage"_n, member<person, uint64_t, &person::get_secondary_1>>
```

6.4 Update upsert function parameters

With all the changes, the function will be:

```
 void upsert(
name user, std::string first_name, std::string last_name, uint64_t age, std
```

Add additional lines to update the `age` field in the `upsert` function as follows:

```
void upsert(name user, std::string first_name, std::string last_name, uint
   require_auth( user );
   address_index addresses( get_first_receiver(), get_first_receiver().valu
   auto iterator = addresses.find(user.value);
   if( iterator == addresses.end() )
   {
     addresses.emplace(user, [&]( auto& row ) {
      row.key = user;
      row.first_name = first_name;
      row.last_name = last_name;
      // -- Add code below --
      row.age = age;
      row.street = street;
      row.city = city;
      row.state = state;
     });
   }
   else {
     std::string changes;
     addresses.modify(iterator, user, [&]( auto& row ) {
       row.key = user;
       row.first_name = first_name;
       row.last_name = last_name;
       // -- Add code below --
       row.age = age;
       row.street = street;
       row.city = city;
       row.state = state;
     });
   }
 }
```

6.5 Compile and delpoy contract addressbook

```
eosio-cpp -o addressbook.wasm addressbook.cpp --abigen
```

6.6 Describe the indexes in the ABI file and execute:

```
cleos set contract addressbook CONTRACTS_DIR/addressbook
```

6.7 Testing

Insert entries into the `alice` and `bob` users' table.

```
cleos push action addressbook upsert '["alice", "alice", "liddell", 9, "123
cleos push action addressbook upsert '["bob", "bob", "is a guy", 49, "doesn
```

Check table data for users `alice` and `bob` by age index. In command lines, use the `--index 2` parameter to specify a query on the secondary index (index No. 2).

```
cleos get table addressbook addressbook people --upper 10 --key-type i64 --
cleos get table addressbook addressbook people --upper 50 --key-type i64 --
```

The search for data in the table by the secondary index is considered as successful if information of the following type is obtained:

```
{
  "rows": [{
      "key": "alice",
      "first_name": "alice",
      "last_name": "liddell",
      "age": 9,
      "street": "123 drink me way",
      "city": "wonderland",
      "state": "amsterdam"
    },{
      "key": "bob",
      "first_name": "bob",
      "last_name": "is a loser",
      "age": 49,
      "street": "doesnt exist",
      "city": "somewhere",
      "state": "someplace"
    }
  ],
  "more": false
}
```