Grove Documentation

Release 0.1.0

Piper Merriam

February 29, 2016

Contents

Grove is an ethereum contract that provides an API for storing and retrieving ordered data in a fast manner.

Grove indexes use an AVL tree for storage which has on average `O(log n)` time complexity for inserts and lookups.

Grove can be used as a library within your own contract, or as a service by interacting with the publicly deployed Grove contract.

The Grove Library can be used at the address `0xd07ce4329b27eb8896c51458468d98a0e4c0394c`.

The Grove contract can be used at `0x8017f24a47c889b1ee80501ff84beb3c017edf0b`.

If you would like to verify the source, it was compiled using version `0.1.5-23865e39` of the `solc` compiler with the `--optimize` flag turned on.

CHAPTER 1

Introduction to Grove

To understand how Grove can be used, we need a situation where we care about the ordering of our data. Lets explore this in the context of a name registrar. Suppose we have a contract which holds some set of key/value mappings.

```
contract namereg {
    struct Record {
        address owner;
        bytes32 key;
        bytes32 value;
        uint expires;
    }
    ...
}
```

One question that someone might want to ask this data is *"Which record will expire next"*. To enable this, the namereg contract would need to feed registrations into and index in grove.

```
contract namereg {
    // Here, Grove might be an abstract solidity contract that allows our
    // namereg contract to interact with it's public API.
    Grove grove = Grove('0x6b07cb54be50bc040cca0360ec792d7b5609f4db');

    struct Record {
        address owner;
        bytes32 key;
        bytes32 value;
        uint expires;
    }
    function register(bytes32 key, bytes32 value) {
        ... // Do the actual registration

        grove.insert("recordExpiration", key, record.expires);
    }
}
```

So, each time someone registers a key, it tells grove about the record, using the `key` as the `id`, and the expiration time of the record as the `value` that grove will use for ordering. Grove will store these records in the index `recordExpiration`.

Someone who wished to query for the next record to expire would do something like the following. We'll assume that we've loaded the Grove ABI into a contract object in web3.

```
// Compute the ID for the index we want to query
> index_id = Grove.getIndexId(namereg.address, "recordExpiration")
```

```
// Query grove for the first record that is greater than 'now'.
> node_id = Grove.query(index_id, ">", now_timestamp)
> record_id = Grove.getNodeId(node_id)
```

In this example, `node_id` would either be `0x0` if there is no record who's expiration is greater than the `now_timestamp` value, or the `bytes32` of the first node in the index that is greater than `now_timestamp`.

Assuming that `node_id` is not `0x0`, we can then fetch the `key` for the record using the `grove.getNodeId` function.

CHAPTER 2

Grove API

2.1 Indexes

In grove, an index represents one set of ordered data, such as the ages of a set of people.

Indexes are namespaced by ethereum address, meaning that any write operations to an index by different addresses will write to different indexes. Each index has an id which can be computed as `sha3(ownerAddress, indexName)`.

You can also use the `computeIndexId(address ownerAddress, bytes32 indexName)` function to compute this for you.

2.2 Nodes

Within each index, there is a binary tree that stores the data. The tree is comprised of nodes which have the following attributes.

- **indexId (bytes32):** the index this node belongs to.
- **id (bytes32):** A unique identifier for the data element that this node represents. Within an index, there is a 1:1 relationship between id and a node in the index.
- **nodeId (bytes32):** a unique identifier for the node, computed as `sha3(indexId, id)`
- **value (int):** an integer that can be used to compare this node with other nodes to determine ordering.
- **parent (bytes32):** the nodeId of this node's parent (0x0 if no parent).
- **left (bytes32):** the nodeId of this node's left child (0x0 if no left child).
- **right (bytes32):** the nodeId of this node's right child (0x0 if no right child).
- **height (bytes32):** the longest path from this node to a child leaf node.

For a given piece of data that you wish to track and query the ordering using Grove, you must be able to provide a unique identifier for that data element, as well as an integer value that can be used to order the data element with respect to the other elements.

2.2.1 Node Properties

- **function getIndexName(bytes32 indexId) constant returns (bytes32)**

  Returns the name for the given index id.

- **function getIndexRoot(bytes32 indexId) constant returns (bytes32)**

  Returns the root node for a given index.

- **function getNodeId(bytes32 nodeId) constant returns (bytes32)**

  Returns the id of the node.

- **function getNodeIndexId(bytes32 nodeId) constant returns (bytes32)**

  Returns the indexId of the node.

- **function getNodeValue(bytes32 nodeId) constant returns (int)**

  Returns the value for the node.

- **function getNodeHeight(bytes32 nodeId) constant returns (uint)**

  Returns the height for the node.

- **function getNodeParent(bytes32 nodeId) constant returns (bytes32)**

  Returns the parent of the node.

- **function getNodeLeftChild(bytes32 nodeId) constant returns (bytes32)**

  Returns the left child of the node.

- **function getNodeRightChild(bytes32 nodeId) constant returns (bytes32)**

  Returns the right child of the node.

2.2.2 Tree Traversal

- **function getNextNode(bytes32 nodeId) constant returns (bytes32)**

  Returns the node ID for the next sequential node in the index. Returns 0x0 if there is not next node.

- **function getPreviousNode(bytes32 nodeId) constant returns (bytes32)**

  Returns the node ID for the previous sequential node in the index. Returns 0x0 if there is no previous node.

2.2.3 Insertion

function insert(bytes32 indexName, bytes32 id, int value) public

You can use the `insert` function insert a new piece of data into the index.

- **indexName:** The name of this index. The index id is automatically computed based on `msg.sender`.
- **id:** The unique identifier for this data.
- **value (int):** The value that can be used to order this node with respect to the other nodes.

Note: If a node with the given **id** is already present in the index, then the

---

2.2.4 Removal (deletion)

function remove(bytes32 indexName, bytes32 id) public

You can use the `remove` function to remove an **id** from the index.

2.2.5 Existence

function exists(bytes32 indexId, bytes32 id) public returns (bool)

You can use the `exists` function to query whether an **id** is present within a given index.

2.2.6 Querying

function query(bytes32 indexId, bytes2 operator, int value) public returns (bytes32)

Each index can be queried using the `query` function.

- **indexId:** The id of the index that should be queried. Note that this is **not** the name of the index.
- **operator:** The comparison operator that should be used. Supported operators are <, <=, >, >=, ==.
- **value:** The value that each node in the index should be compared againt.

For >, >= and == the left-most node that satisfies the comparison is returned.

For < and <= the right-most node that satisfies the comparison is returned.

If no nodes satisfy the comparison, then 0x0 is returned.

2.3 Abstract Solidity Contract

This abstract contract can be used to let your contracts access the Grove API natively.

```
contract GroveAPI {
    /*
     *  Shortcuts
     */
    function computeIndexId(address ownerAddress, bytes32 indexName) constant returns (bytes32);
    function computeNodeId(bytes32 indexId, bytes32 id) constant returns (bytes32);

    /*
     *  Node and Index API
     */
    function getIndexName(bytes32 indexId) constant returns (bytes32);
    function getIndexRoot(bytes32 indexId) constant returns (bytes32);
    function getNodeId(bytes32 nodeId) constant returns (bytes32);
    function getNodeIndexId(bytes32 nodeId) constant returns (bytes32);
    function getNodeValue(bytes32 nodeId) constant returns (int);
    function getNodeHeight(bytes32 nodeId) constant returns (uint);
    function getNodeParent(bytes32 nodeId) constant returns (bytes32);
    function getNodeLeftChild(bytes32 nodeId) constant returns (bytes32);
    function getNodeRightChild(bytes32 nodeId) constant returns (bytes32);

    /*
     *  Traversal
     */
    function getNextNode(bytes32 nodeId) constant returns (bytes32);
    function getPreviousNode(bytes32 nodeId) constant returns (bytes32);

    /*
     *  Insert and Query API
     */
    function insert(bytes32 indexName, bytes32 id, int value) public;
```

```
    function query(bytes32 indexId, bytes2 operator, int value) public returns (bytes32);
    function exists(bytes32 indexId, bytes32 id) constant returns (bool);
    function remove(bytes32 indexName, bytes32 id) public;
}
```

2.4 Contract ABI

The contract can be accessed via web3.js with

```
var Grove = web3.eth.contract([{"constant":true,"inputs":[{"name":"indexId","type":"bytes32"},{"name"
```

CHAPTER 3

Library Usage

Grove can be used as a solidity library, allowing use of all of Groves functionality using your own contracts storage.

The primary Grove contract itself is actually just a wrapper around the deployed library contract.

3.1 Example

In this example, we will use a fictional contract `Members` which tracks information about some set of members for an organization.

```
library GroveAPI {
    struct Index {
            bytes32 id;
            bytes32 name;
            bytes32 root;
            mapping (bytes32 => Node) nodes;
    }

    struct Node {
            bytes32 nodeId;
            bytes32 indexId;
            bytes32 id;
            int value;
            bytes32 parent;
            bytes32 left;
            bytes32 right;
            uint height;
    }

    function insert(Index storage index, bytes32 id, int value) public;
    function query(Index storage index, bytes2 operator, int value) public returns (bytes32);
}

contract Members {
    Grove.Index ageIndex;

    function addMember(bytes32 name, uint age) {
        ... // Do whatever needs to be done to add the member.

        // Update the ageIndex with this new member's age.
        Grove.insert(ageIndex, name, age);
    }
```

```
    function isAnyoneThisOld(uint age) constant returns (bool) {
        return Grove.query("==", age) != 0x0;
    }
}
```

The `Members` contract above interfaces with the Grove library in two places.

First, within the `addMember` function, everytime a member is added to the organization, their age is tracked in the `ageIndex`.

Second, the `isAnyoneThisOld` allows checking whether any member is a specific age.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search