Solidity CRUD- Part 1

Data Storage With Sequential Access, Random Access and Delete

Author: Rob Hitchens
Date: February, 2017

It seems to be a rite-of-passage to study Solidity and then branch out in search of the database stuff; the hidden assumption being that a modern language must have a generalized storage system that maps more or less to the way we think about tables, collections, indexes and queries.

There's no database. We are responsible for the internal organization of data at a deep level. This implies that we'll need to address details we may not be accustomed to dealing with.

Let's proceed on the assumption that we will frequently need a well-solved general-purpose pattern for handling on-chain table-like storage.

C.R.U.D. – Create, Retrieve, Update, Delete

In this two-part series, we're going to explore a pattern that will help us:

1) organize a single entity ("table" or "collection" if you prefer) with a defined set of fields ("columns", "attributes", or "properties"),
2) insert new records (or "documents") known by a Key ("primary key", "document id"),
3) randomly retrieve records by their keys,
4) retrieve a record count,
5) access a list of all the records that exist,
6) update field(s) in any given record,
7) validate that keys exist, or do not exist, and
8) delete a record while maintaining the internal structure.

And, we want it keep it simple:

1) a simple mental model we won't get lost in,
2) as few lines of code as possible, so we don't get lost in that, and
3) efficient gas consumption, so we aren't discouraged from using it.

Importantly:

1) It should scale.

For many use-cases, simply creating, retrieving and updating is sufficient. In part 1, we'll explore a rugged, flexible and minimalist pattern for these three operations. In part 2, we'll extend the pattern with a Delete function.

The User Example

We'll apply the pattern to an imagined collection of users we want to store. The records have:

1) Wallet addresses we will use as Keys,
2) Email addresses, and
3) Age

The pattern can be adapted to any data structure where the fields are known in advance.

The Basic Tools – Mappings

Mappings are, generally-speaking, distributed hash tables that provide one-move lookups and writes to a massive address space. The address space is exclusive to our contract. Here's a simple example:

```
mapping (address => uint) userBalances;
```

What it says:

1) This will be an address space.
2) The keys will be type `address`, such as `msg.sender`.
3) We're going to store a `uint` (unsigned integer) in each mapped location.
4) We'll call those values `userBalances`.

It's important to know that even though we haven't actually written anything to this address space yet, all possible locations are zeroed out.

You can think of the example mapping as a huge table with a primary key of type `address` in which *all possible addresses now exist*. A second column contains an unsigned integer called `userBalances`, all of which have been initialized to 0[i].

We can set and get values like this:

```
address key = msg.sender;                    // the key is an address
userBalances[key] = 100;                     // set value
uint retrievedBalance = userBalances[key];  // get value
```

There's a vexing limitation here. There is no obvious way to know the keys that have been written to. There is no way to know how many keys exist (because they all exist). There is no way to iterate over the keys. Awkward.

The Basic Tools – Arrays

We can have arrays of defined types and those arrays can be fixed length or dynamic. We'll use a dynamic array to maintain an unordered list of keys so we can know how many records exist and what their keys are.

We define a dynamic array of addresses that we'll call `userIndex`. We're going track the keys we insert.

```
address[] userIndex;
```

We can append a new key to the index with ease:

```
userIndex.push(anAddress);
```

We can get the record count with ease:

```
uint count = userIndex.length;
```

We can retrieve keys from the list by their row numbers:

```
address firstUserAddress  = userIndex[0];
address secondUserAddress = userIndex[1];
```

The Basic Tools – Structs

A struct is a way to create our own variable types. They use named keys that map to other already existing types. For example:

```
struct UserStruct {
     bytes32 userEmail;
     uint    userAge;
}
```

Now we have a *type* called UserStruct. We can use it to define a new variable:

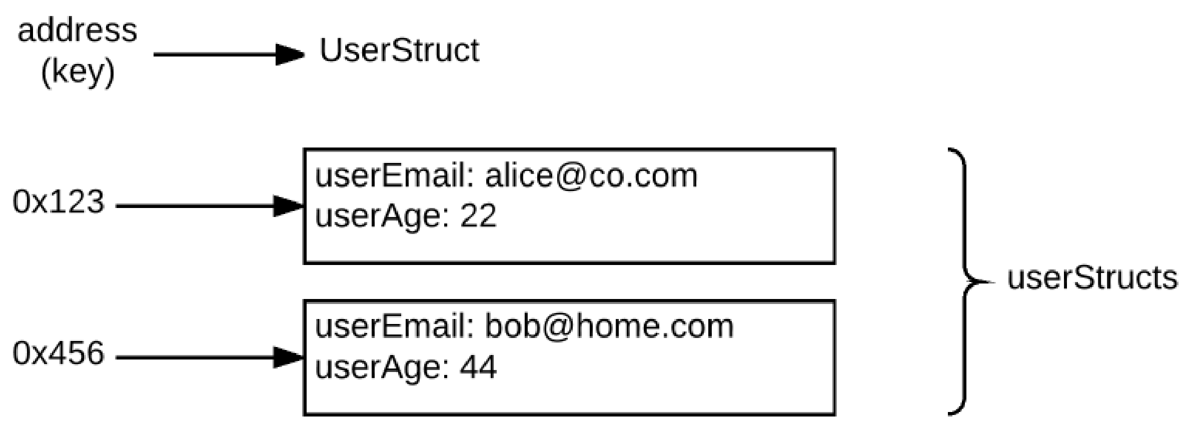
```
UserStruct aUser;
```

We can start treating aUser as something that maps reasonably well to our idea of a record:

```
aUser.userEmail    = emailInHex;
aUser.userAge      = ageAsUnsignedInteger;
```

We can store such structs in a mapping:

```
mapping(address => UserStruct) userStructs;
```



Organized this way, we can randomly access a complete user record and its fields using the key:

```
bytes32 hisEmail = userStructs[hisAddress].email; // get
userStructs[hisAddress].email = newEmailInHex;    // set
```

Putting This Together: Summary of Data Organization

We're going to combine these tools for the desired effect. We'll use:

1) A `struct`, for record details. It will contain all the fields except the key.
2) A `mapping` of keys to struct instances.
3) An array that will be an unordered index of keys. In the example, these are user addresses.

Create / Insert:

1) Store the values in the `struct`.
2) Store the structure in the `mapping`.
3) Append the primary key to the index array. This forms an unordered list of the keys in the system.

Our interface will look like this:

```
function insertUser(address userAddress, bytes32 userEmail, uint userAge) public returns(bool
success) {}
```

We can imagine a very simple process that addresses steps 1 & 2:

```
  function insertUser(
    address userAddress,
    bytes32 userEmail,
    uint    userAge)
    public
    returns(bool success)
  {
    userStructs[userAddress].userEmail = userEmail;
    userStructs[userAddress].userAge   = userAge;
    return true;
  }
```

We can address step 3 by adding the new key to the unordered key list:

```
  userIndex.push(userAddress);
```

This will get us most of the way there, but keep reading. We will elaborate on this slightly later in the paper.

Retrieve

1) Look up the structure in the map using the primary key.

Our interface will look like this:

```
function getUser(address userAddress) public constant returns(bytes32 userEmail, uint userAge) {}
```

For clarity, this function doesn't return the key (`userAddress`) because we obviously already know its value. This is also why we don't store keys in the `UserStructs`. Doing so would be redundant.

Similarly, we can imagine the first formative conception of the function that we will elaborate on shortly:

```
function getUser(address userAddress)
  constant
  returns(bytes32 userEmail, uint userAge)
{
  return(
    userStructs[userAddress].userEmail,
    userStructs[userAddress].userAge);
}
```

Update

1) Set a value in a `struct` stored in the map located by the primary key.

Our interface can be for a single field (as in the Sample Implementation):

```
function updateUserEmail(address userAddress, bytes33 userEmail) public returns(bool success) {}
```

Or, we can opt for a whole record or any useful combination of fields:

```
function updateUser(address userAddress, bytes32 userEmail, uint userAge) public returns(bool
success) {}
```

In this system, any formation of fields can be attached to a key. No changes to keys are allowed. It's not unthinkable to extend the pattern to support that but it implies an unwanted increase in complexity.

Does this key exist?

When we organize data using unique identifiers, the identifiers need to be unique, by definition. We will find it desirable to check for the existence of a key.

A mapping initializes `struct` members to 0, but 0 may have meaning in our application. In other words, `if(value==0)` won't indicate that no data exists in all cases. Perhaps it *is* valid data and the answer is 0. We want widely applicable pattern.

One solution to this problem is to set a Boolean flag to indicate that there is indeed data stored in the `mapping` at a given location. Using this approach, we would add a member to the struct:

```
struct UserStruct {
      bytes32 userEmail;
      uint    userAge;
      bool    isUser;
}
```

We would set this flag to `true` as we add users. We could easily imagine a simple function to check if stored structs contain explicitly set information and not merely defaults:

```
Function isUser(address userAddress) public constant returns(bool isIndeed) {
      Return userStructs[userAddress].isUser;
}
```

This is a valid approach. With this structure in place, we can then embellish the insert, retrieve and update functions with some logical constraints:

- Cannot insert a key that already exists.
- Cannot update a key that doesn't exist.
- Cannot retrieve a key that doesn't exist.

We'll find it advantageous that these functions differentiate between logical and illogical requests. For example, the retrieve function differentiates between an all zero response to a zeroish record that *does* exist, and a `throw` response to a record that simply does not exist.

This system will work in the case that:

- A delete function is not required
- A delete function will *never be required, even in the case of upgraded logic.*

We can overcome this limitation using a *slightly* different strategy for determining the existence/non-existence of a key. The following data structure will support a delete operation if we ever need to add one. The delete procedure will explained in Part 2 of this series.

Instead of adding a Boolean, flag to the `struct`, we'll add a pointer. This pointer will indicate the key's position in the unordered list of keys, called `userIndex`.

```
struct UserStruct {
      bytes32 userEmail;
      uint    userAge;
      uint    index;
}
```

Pointer Logic in Inserts

We need an additional step in the insert process:

1. Set the index pointer in the stored struct. It's always equals the `index.length` because we're always adding to the end of the index. For example:

   ```
   userStruct[newKey].index = userIndex.length;
   ```

   In case this isn't clear, the index starts at 0. Suppose we have 939 records in rows 0-938. `index.length` will be 939. We'll record 939 in the pointer, because when we `push()` the new key into the index in the next step, it will *certainly* land in row 939.

   Even better, since `.push()` returns the new array length, we can use it in concert with this step and reduce our gas cost a little:

   ```
   userStructs[newKey].index = userIndex.push(userAddress)-1;
   ```

Referential Integrity

An astute reader pointed out that if the index ever gets out of sync with the stored records, we will have a serious problem. Smart Contracts are especially good at ensuring such a situation never unfolds if we code it correctly, so we focus on prevention. No provision is made for messy exceptions.

Internal referential integrity is maintained at all times. Inserts and deletes either complete entirely or not at all. The rules are fairly simple. Be careful to strictly enforce these rules and test, test, test your *implementation* of this pattern to be sure you captured all the steps.

1. There is a two-way binding between the mapped structs and the index.
2. Don't allow any operation to break Rule #1. Our operations are atomic. They execute in an "all or nothing" fashion. Operations that don't succeed completely are reverted completely so no incomplete reorganization is ever permitted. Ensure your implementation adheres to this principle.

It's advisable to set the index and mapping to `private` to ensure no accidental overwrites from child contracts.

```
mapping (address => UserStruct) private userStructs;
address[] private userIndex;
```

Invalid Requests Fail Early and Fail Hard

We `throw` on invalid operations to ensure referential integrity is never broken:

1) update a field in a non-existent record (missing key),
2) insert a duplicate key.

There's no good reason to do those things.

We also `throw` if one tries to retrieve details about a key that doesn't exist. That means a deleted record is gone unless additional logic is added to *defeat the design*. It also means an all zero response to an existing key is clearly differentiated from an abortive response to a non-existent key.

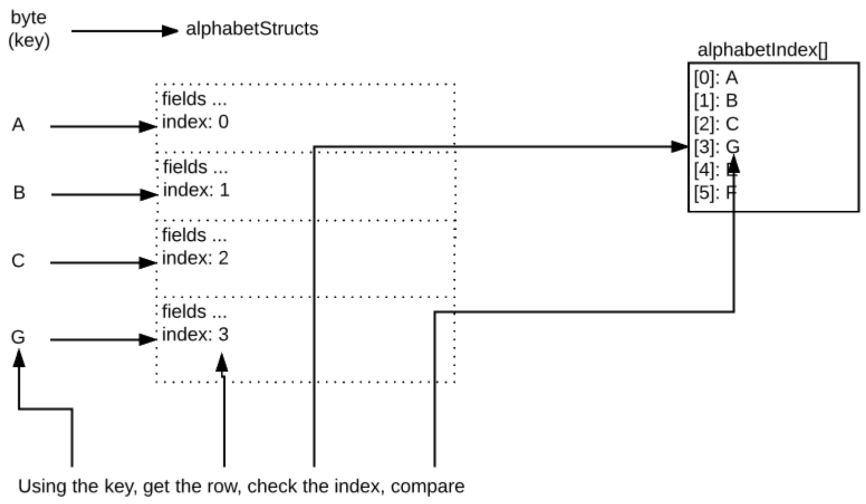
Check If Key Exists

We can validate the existence (or non-existence) of a record given a primary key to test:

1) Get the index pointer from the structure stored in the map at the primary key location.
2) Compare the index at the pointed-to row number with the key we are testing. In practice, we can't access non-existent rows in the index, so in the case that the index is empty, we will know there is no possibility of a match.
3) In the case that there is at least one other Key in the list and the retrieved mapped value refers to it, we compare. Do they match?

If it's a key that exists, say … "G", we would:

1) Notice the list isn't empty.
2) Look up the struct stored in the mapping at "G" and note that it says index row "3".
3) Look up the address stored in index row "3" and note that it says "G".
4) Find that "G" == "G", and conclude that "G" exists.



If the key doesn't exist, say … "Z", we would follow the same process, but it wouldn't pass:

1) If the list is empty, the Key doesn't exist.
2) Look up the structure stored at "Z" and note that it says index row "0" (because of defaults).
3) Look up the address stored in index row "0" and note that it says "A".
4) Find that "A" != "Z", and conclude that "Z" does not exist.

The example code is using `userAddress` for primary keys. It checks if keys do or don't exist in one step, like this:

```
If(userIndex.length == 0) return false; // the list is empty, so "the" key to test isn't present
return(userIndex[userStructs[userAddress].index] == userAddress); // true = exists
```

The `return` is nesting a few arguments to complete the 3-step logic in one statement. If the list is empty, or the round-trip consistency isn't there, the tested key *definitely doesn't* exist. If the round-trip consistency *is* there, then the key *definitely does* exist.

We add a function to check this, and expose the function publicly so clients can use it:

```
function isUser(address userAddress)
  public
  constant
  returns(bool isIndeed)
{
  if(userIndex.length == 0) return false;
  return (userIndex[userStructs[userAddress].index] == userAddress);
}
```

We use this function to check the existence of keys before we permit any changes, and before we return an instance/record. Any illogical request fails hard with `throw`.

```
if(!isUser(userAddress) throw; // missing key
```

The Example Implementation stores two simple non-key fields; `userEmail` and `userAge`. The actual mapped structs you store this way can contain any Solidity data type including arrays and structs. Be aware of storage costs and stack depth, and remember that not everything belongs in on-chain storage.

We can round out the basic functionality of this pattern with some useful functions that are very simple to deploy.

Iterable Interface

We expose the record count …

```
Function getUserCount() public constant returns(uint count) {
        return userIndex.length;
}
```

And a function to support iteration over the list:

```
Function getUserAtIndex(uint index) public constant returns(address userAddress) {
        return userIndex[index];
}
```

A web3 client can:

1) Get the size of the index: `UserCrud.getUserCount.call()*`

```
for(row=0; row < count; row++) {
```

2) Retrieve the keys one by one: `UserCrud.getUserAtIndex.call(row)`
3) Retrieve the details key by key: `UserCrud.getUser.call(theAddressFromStep2)`

```
}

*Truffle style
```

We'll add event emitters so a web3 client can also monitor events that chronicle every state change that has ever taken place and arrive in a timely fashion suitable for live updates.

Event Emitters

It's a best practice to emit events for state changes within a contract. Clients can use these to maintain their own synchronized copies of the stored data. That means clients have two avenues to interact with this on-chain storage:

1) Getter functions that retrieve the current state, and
2) Event emitters that report all state changes

```
event LogNewUser   (address indexed userAddress, uint index, bytes32 userEmail, uint userAge);
event LogUpdateUser(address indexed userAddress, uint index, bytes32 userEmail, uint userAge);
```

We index the `userAddress`es so clients can quickly filter, sort and find relevant information in the event logs. We emit the `index` rows because they are part of the state. We further refine the return value in our create function, substituting `uint index` for `bool success`. Don't worry about returning 0. Row 0 is valid. An invalid operation will `throw` an error and no event will be emitted.

```
returns(uint index);
```

Gas Consumption and Performance

While not formally tested, gas consumption is expected to remain approximately consistent *at any scale* because each write operation proceeds in a step-by-step fashion with no branching or loops.

Representative write costs (Sample implementation):

- insertUser(): 89K
- updateUserEmail(): 8K
- updateUserAge(): 8K

The "constant" functions we use for read-only access are free, but gas is a useful proxy for the workload involved:

- getUserAtIndex(): 700 gas
- getUser(): 1,400 gas
- getUserCount(): 400 gas

These figures are rounded *up* from observed results January, 2017, solc 0.4.9.

Delete

In part 2, we'll expand on this example, to add a `deleteUser()` function. We'll show a simple process for reorganization the internal data structure efficiently, with a consistent gas cost at any scale.

Acknowledgement

I'd like to thank Xavier Lepretre, senior consultant at B9Lab, for his indispensable input and support.

A Note About Security

In most cases you will want to restrict access to the `insert/update` and possibly `get` functions. You may wish to add an "`onlyOwner`" modifier to the `public` functions or change them to `internal` so only child contracts can use them.

There is considerable latitude in the implementation of this pattern. Suffice it to say that you shouldn't overlook the critically important detail of restricting function access to entities that should have it.

Sample Implementation

For clarity, security is intentionally omitted.

Code is available at https://www.github.com/robhitchens/... ** TO DO **

```
pragma solidity ^0.4.6;

contract UserCrud {

  struct UserStruct {
    bytes32 userEmail;
    uint userAge;
    uint index;
  }

  mapping(address => UserStruct) private userStructs;
  address[] private userIndex;

  event LogNewUser   (address indexed userAddress, uint index, bytes32 userEmail, uint userAge);
  event LogUpdateUser(address indexed userAddress, uint index, bytes32 userEmail, uint userAge);

  function isUser(address userAddress)
    public
    constant
    returns(bool isIndeed)
  {
    if(userIndex.length == 0) return false;
    return (userIndex[userStructs[userAddress].index] == userAddress);
  }

  function insertUser(
    address userAddress,
    bytes32 userEmail,
    uint    userAge)
    public
    returns(uint index)
  {
    if(isUser(userAddress)) throw;
    userStructs[userAddress].userEmail = userEmail;
    userStructs[userAddress].userAge   = userAge;
    userStructs[userAddress].index     = userIndex.push(userAddress)-1;
    LogNewUser(
        userAddress,
        userStructs[userAddress].index,
        userEmail,
        userAge);
    return userIndex.length-1;
  }

  function getUser(address userAddress)
    public
    constant
    returns(bytes32 userEmail, uint userAge, uint index)
  {
    if(!isUser(userAddress)) throw;
    return(
      userStructs[userAddress].userEmail,
      userStructs[userAddress].userAge,
      userStructs[userAddress].index);
  }

  function updateUserEmail(address userAddress, bytes32 userEmail)
    public
    returns(bool success)
  {
    if(!isUser(userAddress)) throw;
    userStructs[userAddress].userEmail = userEmail;
    LogUpdateUser(
      userAddress,
      userStructs[userAddress].index,
      userEmail,
      userStructs[userAddress].userAge);
    return true;
  }

  function updateUserAge(address userAddress, uint userAge)
    public
    returns(bool success)
  {
    if(!isUser(userAddress)) throw;
    userStructs[userAddress].userAge = userAge;
    LogUpdateUser(
      userAddress,
      userStructs[userAddress].index,
      userStructs[userAddress].userEmail,
      userAge);
    return true;
  }

  function getUserCount()
    public
    constant
    returns(uint count)
  {
    return userIndex.length;
  }

  function getUserAtIndex(uint index)
    public
    constant
    returns(address userAddress)
  {
    return userIndex[index];
  }

}
```

End Notes

[i] Mappings are initialized to 0 for each key hash in a contract's mapping. In the unlikely event that two keys with identical hashes can be found, called a hash collision, values at the mapped location will not necessarily be initialized to 0. A mapping location may indeed be occupied by unexpected data. Such a collision is statistically improbable. At the time of writing, no hash collision has been discovered using the Keccak-256 cryptographic hash function employed by Ethereum's mappings.