Collections

When deciding on data structures to use for the application's data, it is important to minimize the amount of data read and written to storage, and the amount of data serialized and deserialized to minimize the cost of transactions. It is important to understand the tradeoffs of data structures in your smart contract because it can become a bottleneck as the application scales, and migrating the state to the new data structures will come at a cost.

The collections withinnear-sdk-js are designed to split the data into chunks and defer reading and writing to the store until needed. These data structures will handle the low-level storage interactions and aim to have a similar API to thenative data structures in JavaScript .

It is important to keep in mind that when using collections, that each time state is loaded, all entries in the data structure will be read eagerly from storage and deserialized. This will come at a large cost for any non-trivial amount of data, so to minimize the amount of gas used the SDK collections should be used in most cases.

The most up to date collections and their examples can be foundin the repository on GitHub .

The following data structures that exist in the SDK are as follows:

SDK Collection Native Equivalent Description Vector Array A growable array type. The values are sharded in memory and can be used for iterable and indexable values that are dynamically sized. LookupMap Map This structure behaves as a thin wrapper around the key-value storage available to contracts. This structure does not contain any metadata about the elements in the map, so it is not iterable. UnorderedMap Map Similar toLookupMap , except that it stores additional data to be able to iterate through elements in the data structure. LookupSet Set A set, which is similar toLookupMap but without storing values, can be used for checking the unique existence of values. This structure is not iterable and can only be used for lookups. UnorderedSet Set An iterable equivalent ofLookupSet which stores additional metadata for the elements contained in the set.

In-memoryMap

vs persistentUnorderedMap

- Map
- keeps all data in memory. To access it, the contract needs to deserialize the whole map.
- UnorderedMap
- keeps data in persistent storage. To access an element, you only need to deserialize this element.

UseMap in case:

- Need to iterate over all elements in the collectionin one function call
- .
- The number of elements is small or fixed, e.g. less than 10.

UseUnorderedMap in case:

- Need to access a limited subset of the collection, e.g. one or two elements per call.
- Can't fit the collection into memory.

The reason isMap deserializes (and serializes) the entire collection in one storage operation. Accessing the entire collection is cheaper in gas than accessing all elements throughN storage operations.

Example ofMap :

import

{

NearBindgen , call , view , near }

from

"near-sdk-js" ;

@ NearBindgen ({ }) export

class

StatusMessage

{ constructor ( )

```
{ this . records

=

new

Map ( ) ; }

@ call ( { } ) set_status ( { message } )

{ let account_id = near . signerAccountId ( ) ; near . log ( account_id } set_status with message { message } ) ; this . records . set (
account_id , message ) ; }

@ view ( { } ) get_status ( { account_id } )

{ near . log (get_status for account_id { account_id } ) ; return

this . records . get ( account_id ) ; } } Example ofUnorderedMap :

import

{

NearBindgen , call , view , near ,

UnorderedMap

}

from

"near-sdk-js" ;

@ NearBindgen ( { } ) export

class

StatusMessage

{ constructor ( )

{ this . records

=

new

UnorderedMap ( "a" ) ; }

@ call ( { } ) set_status ( { message } )

{ let account_id = near . signerAccountId ( ) ; near . log ( account_id } set_status with message { message } ) ; this . records . set (
account_id , message ) ; }

@ view ( { } ) get_status ( { account_id } )

{ near . log (get_status for account_id { account_id } ) ; return

this . records . get ( account_id ) ; }

@ view ( { } ) get_all_statuses ( )

{ return

this . records . toArray ( ) ; } }
```

Error prone patterns

Because the values are not kept in memory and are lazily loaded from storage, it's important to make sure if a collection is replaced or removed, that the storage is cleared. In addition, it is important that if the collection is modified, the collection itself is updated in state because most collections will store some metadata.

Some error-prone patterns to avoid that cannot be restricted at the type level are:

import

{

UnorderedMap , assert }

from

"near-sdk-js" ;

let m =

new

UnorderedMap ("m") ; m . insert (1 ,

"test") ; assert (m . length ( ) ,

1) ; assert (m . get (1) ,

"test") ;

// Bug 1: Should not replace any collections without clearing state, this will reset any // metadata, such as the number of elements, leading to bugs. If you replace the collection // with something with a different prefix, it will be functional, but you will lose any // previous data and the old values will not be removed from storage. let m =

new

UnorderedMap ("m") ; assert (m . length ( ) ,

0) ; assert (m . get (1) ,

"test") ;

// Bug 2: Should not use the same prefix as another collection // or there will be unexpected side effects. let m2 =

new

UnorderedMap ("m") ; assert (m2 . length ( ) ,

0) ; assert (m2 . get (1) ,

"test") ;

LookupMap

vsUnorderedMap

Functionality

- UnorderedMap
- supports iteration over keys and values, and also supports pagination. Internally, it has the following structures:* a prefix value
-
  - a vector of keys
-
  - aLookupMap
-
  - of keys and values
- LookupMap
- only has a prefix, reading values to and from the contract's storage. Without a vector of keys, it doesn't have the ability to iterate over keys.

Performance

LookupMap has a better performance and stores less data compared toUnorderedMap .

- UnorderedMap
- requires2
- storage reads to get the value and storage writes to insert a new entry.
- LookupMap

- requires only one storage read to get the value and only one storage write to store it.

Storage space

UnorderedMap requires more storage for an entry compared to aLookupMap .

- UnorderedMap
- stores the key twice (once in the first vector and once in itsLookupMap
-) and value once.
- LookupMap
- stores key and value once. Edit this page Last updatedonJan 20, 2023 byDennis Was this page helpful? Yes No

Previous NearBindgen Next Public Methods