Collections

When deciding on data structures to use for the data of the application, it is important to minimize the amount of data read and written to storage but also 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 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 thestd::collections .

Note Thenear_sdk::collections will be moving tonear_sdk::store and have updated APIs. If you would like to access these updated structures as they are being implemented, enable theunstable feature onnear-sdk . It is important to keep in mind that when usingstd::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 documentation can be foundn the rust docs .

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

SDK collection std equivalent Description LazyOption Option Optional value in storage. This value will only be read from storage when interacted with. This value will beSome when the value is saved in storage, andNone if the value at the prefix does not exist. Vector Vec A growable array type. The values are sharded in memory and can be used for iterable and indexable values that are dynamically sized. LookupMap HashMap 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 HashMap Similar toLookupMap , except that it stores additional data to be able to iterate through elements in the data structure. TreeMap BTreeMap An ordered equivalent ofUnorderedMap . The underlying implementation is based on anAVL tree . This structure should be used when a consistent order is needed or accessing the min/max keys is needed. LookupSet HashSet 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 HashSet An iterable equivalent ofLookupSet which stores additional metadata for the elements contained in the set.

In-memoryHashMap

vs persistentUnorderedMap

- HashMap
- 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.

UseHashMap 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 isHashMap 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 ofHashMap :

/// Using Default initialization.

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, Default)]

pub

struct

Contract

{ pub status_updates :

HashMap < AccountId ,

String

, }

[near_bindgen]

impl

Contract

{ pub

fn

set_status (& mut

self , status :

String)

{ self . status_updates . insert (env :: predecessor_account_id ( ) , status) ; assert! (self . status_updates . len ( )

<=

10 ,

"Too many messages") ; }

pub

fn

clear (& mut

self)

{ // Effectively iterating through all removing them. self . status_updates . clear ( ) ; }

pub

fn

get_all_updates (self)

->

HashMap < AccountId ,

String

{ self . status_updates } } Example ofUnorderedMap :

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

pub

struct

Contract

{ pub status_updates :

UnorderedMap < AccountId ,

String

, }

[near_bindgen]

impl

Contract

{

[init]

pub

fn

new ( )

->

Self

{ // Initializing status_updates with unique key prefix. Self

{ status_updates :

UnorderedMap :: new (b"s" . to_vec ( )) , } }

pub

fn

set_status (& mut

self , status :

String)

{ self . status_updates . insert (& env :: predecessor_account_id ( ) ,

& status) ; // Note, don't need to check size, sinceUnorderedMap doesn't store all data in memory. }

pub

fn

delete_status (& mut

self)

{ self . status_updates . remove (& env :: predecessor_account_id ( )) ; }

pub

fn

get_status (& self , account_id :

AccountId)

->

Option < String

{ self . status_updates . get (& account_id) } }

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:

use

near_sdk :: store :: UnorderedMap ;

let

mut m =

UnorderedMap :: < u8 ,

String

:: new (b"m") ; m . insert (1 ,

"test" . to_string ( )) ; assert_eq! (m . len ( ) ,

1) ; assert_eq! (m . get (& 1) ,

Some (& "test" . to_string ( ))) ;

// 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. m =

UnorderedMap :: new (b"m") ; assert! (m . is_empty ( )) ; assert_eq! (m . get (& 1) ,

Some (& "test" . to_string ( ))) ;

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

UnorderedMap :: < u8 ,

String

:: new (b"m") ; assert! (m2 . is_empty ( )) ; assert_eq! (m2 . get (& 1) ,

Some (& "test" . to_string ( ))) ;

// Bug 3: forgetting to save the collection in storage. When the collection is attached to // the contract state (self in #[near_bindgen]) this will be done automatically, but if // interacting with storage manually or working with nested collections, this is relevant. use

near_sdk :: store :: Vector ;

// Simulate roughly what happens during a function call that initializes state. { let v =

Vector :: < u8

:: new (b"v") ; near_sdk :: env :: state_write (& v) ; }

// Simulate what happens during a function call that just modifies the collection // but does not store the collection itself. { let

mut v :

Vector < u8

=

near_sdk :: env :: state_read ( ) . unwrap ( ) ; v . push (1) ; // The bug is here that the collection itself if not written back }

let v :

Vector < u8

=

near_sdk :: env :: state_read ( ) . unwrap ( ) ; // This will report as if the collection is empty, even though the element exists assert! (v . get (0) . is_none ( )) ; assert! (near_sdk :: env :: storage_read (& [b"v" . as_slice ( ) ,

& 0u32 . to_le_bytes ( )] . concat ( )) . is_some ( )) ;

// Bug 4 (only relevant for near_sdk::store): These collections will cache writes as well // as reads, and the writes are performed on Drop // so if the collection is kept in static memory or something like std::mem::forget is used, // the changes will not be persisted. use

near_sdk :: store :: LookupSet ;

let

mut m :

LookupSet < u8

=

LookupSet :: new (b"l") ; m . insert (1) ; assert! (m . contains (& 1)) ;

// This would be the fix, manually flushing the intermediate changes to storage. // m.flush(); std :: mem :: forget (m) ;

m

LookupSet :: new (b"l") ; assert! (! m . contains (& 1)) ;

Pagination with persistent collections

Persistent collections such as UnorderedMap , UnorderedSet and Vector may contain more elements than the amount of gas available to read them all. In order to expose them all through view calls, we can use pagination.

This can be done using iterators with Skip and Take . This will only load elements from storage within the range.

Example of pagination for UnorderedMap :

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize, PanicOnDefault)]

pub

struct

Contract

{ pub status_updates :

UnorderedMap < AccountId ,

String

, }

[near_bindgen]

impl

Contract

{ /// Retrieves multiple elements from the UnorderedMap. /// - from_index is the index to start from. /// - limit is the maximum number of elements to return. pub

fn

get_updates (& self , from_index :

usize , limit :

usize)

->

Vec < (AccountId ,

String)

{ self . status_updates . iter ( ) . skip (from_index) . take (limit) . collect ( ) } }

LookupMap

vsUnorderedMap

Functionality

- UnorderedMap
- supports iteration over keys and values, and also supports pagination. Internally, it has the following structures:* a map from a key to an index
-
    - a vector of keys
-
    - a vector of values
- LookupMap
- only has a map from a key to a value. 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 and3
- 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 map and once in the vector of keys) and value once. It also has a higher constant for storing the length of vectors and prefixes.
- LookupMap
- stores key and value once.

LazyOption

It's a type of persistent collection that only stores a single value. The goal is to prevent a contract from deserializing the given value until it's needed. An example can be a large blob of metadata that is only needed when it's requested in a view call, but not needed for the majority of contract operations.

It acts like anOption that can either hold a value or not and also requires a unique prefix (a key in this case) like other persistent collections.

Compared to other collections,LazyOption only allows you to initialize the value during initialization.

[near_bindgen]

[derive(BorshDeserialize, BorshSerialize,

PanicOnDefault)]

pub

struct

Contract

{ pub metadata :

LazyOption < Metadata

, }

[derive(Serialize, Deserialize, BorshDeserialize, BorshSerialize)]

[serde(crate =

"near_sdk::serde")] pub

struct

Metadata

{ data :

String , image :

Base64Vec , blobs :

Vec < String

, }

[near_bindgen]

impl

Contract

{

[init]

pub

fn

new (metadata :

Metadata)

->

Self

{ Self

{ metadata :

LazyOption :: new (b"m" ,

Some (metadata)) , } }

pub

fn

get_metadata (& self)

->

Metadata

{ // .get() reads and deserializes the value from the storage. self . metadata . get ( ) . unwrap ( ) }Edit this page Last updatedonDec 9, 2023 bygagdiez Was this page helpful? Yes No

Previous near_bindgen Next Collections Nesting