**XODE TRANSACTION FEES**

**Weight​**

As is also the case with Substrate, pallet-contracts use weightV2 to charge execution fees. It is composed of refTime and proofSize :

* **refTime:** The amount of computational time that can be used for execution, in picoseconds.
* **proofSize:** The size of data that needs to be included in the proof of validity in order for the relay chain to verify transaction's state changes, in bytes. So access storage assumes that it will grow the gas fees.

**Information**

Gas = Weight = (refTime, proofSize) Transaction Weight in Substrate Documentation

**Storage Rent​**

Storage rent, also called as Automatic Deposit Collection is a mechanism that ensures security of the chain by preventing on-chain storage spamming. It prevents malicious actors from spamming the network with low-value transactions and to ensure that callers have a financial stake when storing data on-chain.

Users will be charged for every byte stored on-chain and the call will transfer this fee from the free balance of the user to the reserved balance of the contract. Note that the contract itself is unable to spend this reserved balance (but it can expose a function that removes on-chain storage and the caller will get the funds) . It also incentives users to remove unused data from the chain by getting rent fees back. Any user can get back the rent fees if they remove on-chain data (not specifically the user that was first charged for). It's up to the contract developers and users to understand how and if they can get their storage deposit back.

**Storage Rent Calculation​**

This fee is calculated with the price set for each storage item DepositPerItem, and for each byte of storage DepositPerByte.

Per Storage Byte: 0.000001 XON

Per Storage Item: 0.00004 XON

**Calculation​**

When a user stores a new key/value in a Mapping field, one DepositPerItem is charged. The length in bytes of the value is also added to the fee (so bytes length x DepositPerByte).

For example, if a user stores a new entry in a Mapping<u32, AccountId> (AccountId is 32 bytes) it will be charged DepositPerItem + 32 x DepositPerByte.

**For users​**

The first call to a dApp (one or several smart contracts) will usually be more expensive than the following ones. This is because the first call will create a lot of new entries for the user (most of the time it is data related to the user AccountId like a Mapping of Balances). From the second call it should be way cheaper (or free) because it will just update those items.

If the consecutive calls only modify the existing database entry, the caller is only charged for the extra bytes they add to the entry. In the case they reduce the size of the DB entry, they will get storage rent back. What this means in practice is that users can increase their free balance after interacting with a smart contract!

If a user wants to get it back, it should remove on-chain data. It is only possible if the smart-contract exposes a function that removes data from the chain (like remove\_mapping\_entry in the example below).

**For smart-contracts developers​**

As the only way for users to get back their reserved balance is to remove on-chain data, it is important to make sure that the smart-contract exposes functions that allow users to do so.

If the contracts don't expose such functions, there will be no way to remove on-chain data used by the contract and the users will not be able to get their reserved balance back (as it will be a reserved balance on the contract account).

**StorageDepositLimit​**

When doing a contract call one of the arguments is StorageDepositLimit. This value is the maximum amount of storage rent that can be charged for a single call.

**Important**

If StorageDepositLimit is set to None, it allows contracts to charge an arbitrary amount of funds to be drained from the caller's account.

So it is necessary to set a limit (first dry-run the call to get the storage deposit amount) to prevent malicious contracts from draining funds from a user's account. This especially applies for front end applications that trigger contracts calls or for calls sent from contracts UI (like contracts-UI or polkadot-js UI).

Users are responsible for ensuring gas limit & storage deposit limit. This is the same as for EVM smart contracts, but instead of having only non-refundable gas, you also have to take note of StorageDepositLimit.

**Contract example on XODE**

#[ink::contract]

mod rent {

use ink::storage::Mapping;

#[ink(storage)]

pub struct Rent {

map: Mapping<AccountId, u32>,

int: u32,

bool: bool,

}

impl Rent {

#[ink(constructor)]

pub fn new() -> Self {

Self { map: Default::default(), int: 0, bool: false }

}

#[ink(message)]

pub fn update\_32(&mut self, i: u32) {

self.int = i

}

#[ink(message)]

pub fn flip\_bool(&mut self) {

self.bool = !self.bool

}

#[ink(message)]

pub fn add\_mapping\_entry(&mut self) {

let caller = self.env().caller();

// Insert one item to storage. fee = 1 \* PricePerItem (0.0004XON) =

// Value of the mapping is a u32, 4 bytes. fee = 4 \* PricePerByte (0.00002XON) = 0.00008XON

// Total fee = 0.00408XON

self.map.insert(caller, &1u32);

}

#[ink(message)]

pub fn remove\_mapping\_entry(&mut self) {

let caller = self.env().caller();

// Clears the value at key from storage.

// Remove one item from storage. fee = 1 \* PricePerItem (0.0004XON) =

// Remove the value of the mapping u32, 4 bytes. fee = 4 \* PricePerByte (0.00002XON) = 0.00008XON

// Total reserve repatriated by caller = 0.00408XON

self.map.remove(caller);

}

#[ink(message)]

pub fn remove\_entry\_account\_id(&mut self, who: AccountId) {

// Clears the value at key from storage.

// Remove one item from storage. fee = 1 \* PricePerItem (0.0004XON) =

// Remove the value of the mapping u32, 4 bytes. fee = 4 \* PricePerByte (0.00002XON) = 0.00008XON

// Total reserve repatriated by caller = 0.00408XON

self.map.remove(who);

}

}

}

**add\_mapping\_entry​**

The fee (balance that is reserved (moved from free user balance to contract reserved balance)) will be:

* Insert one item to storage. fee = 1 \* PricePerItem (0.0004XON)
* Value of the mapping is u32, 4 bytes. fee = 4 \* PricePerByte (0.00002XON) = 0.00008XON
* Total fee = 0.00408XON

**remove\_mapping\_entry​**

The balance repatriated (balance that is moved from the reserve of the contract account to the user account) will be:

* Remove one item from storage. fee = 1 \* PricePerItem (0.0004XON)
* Remove the value of the mapping u32, 4 bytes. fee = 4 \* PricePerByte (0.00002XON) = 0.00008XON
* Total reserve repatriated by caller = 0.00408XON

**remove\_entry\_account\_id​**

The caller will get the balance repatriated (and not the user that was first charged for, because it is transferred from account reserved balance to free balance of caller). the caller will get the 0.00408XON.

**flip\_bool & update\_32​**

It will not have rent fees because it will not store new data on-chain (only updating value).