Ethereum state rent - rough proposal

This is a non-formalised, pre-EIP description of one of the ways the state rent could be introduced on top of the existing Ethereum protocol. It has not been approved, ratified, voted for by any individual, organisation, or group. It is just a proposal. It distills some ideas expressed in the past, and adds some novel data and insights.

The goal of this is to inform production of other proposals, Proof of Concept implementations, and formal descriptions (EIPs), when it is deemed appropriate.

Copyright

Warnings and disclaimers

Data used for the charts and table has not been cross-verified and may contain errors. It should only be used as a guidance for your own research.

Not all contracts have been analysed and classified into tokens, NFTs, DEXs etc. More research on such classification is welcome.

Different charts have been produced at different state of the blockchain.

Working group is not unanimous on any of the parts of this proposal.

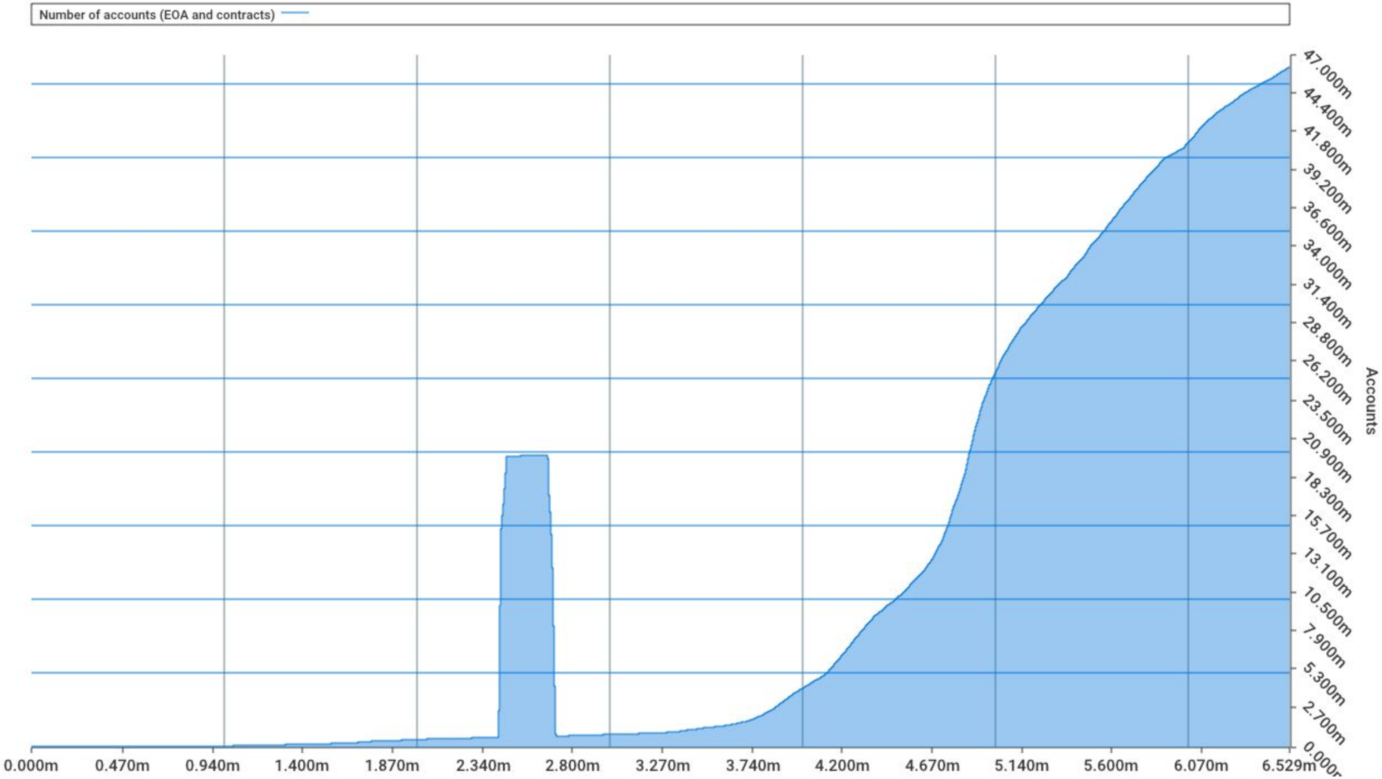
There was no Proof Of Concept, so some or all parts may contain fatal flaws.

Immediate next steps

After publication of this document, the Proof Of Concept needs to be prepared at the same time as public discussions are taking place. This will help identify unseen difficulties and/or unimplementable ideas. Only following that, a series of EIPs can be prepared with sufficient level of formalism and concreteness. Proof Of Concept will also be useful for generating test cases for the EIPs.

Proof Of Concept does not have to be written by the same working group that worked on the proposal, but likely there will be a big overlap.

Chart 1 - growth of accounts (contracts and EOAs)



Comments to Chart 1

Out of 47.64m accounts in total, 40.01m are Externally Owned (non-contracts).

The bump around blocks 2.4m is 2016 DOS attack and subsequent clearing after Spurious Dragon protocol change.

Chart 2 - growth of contract storage

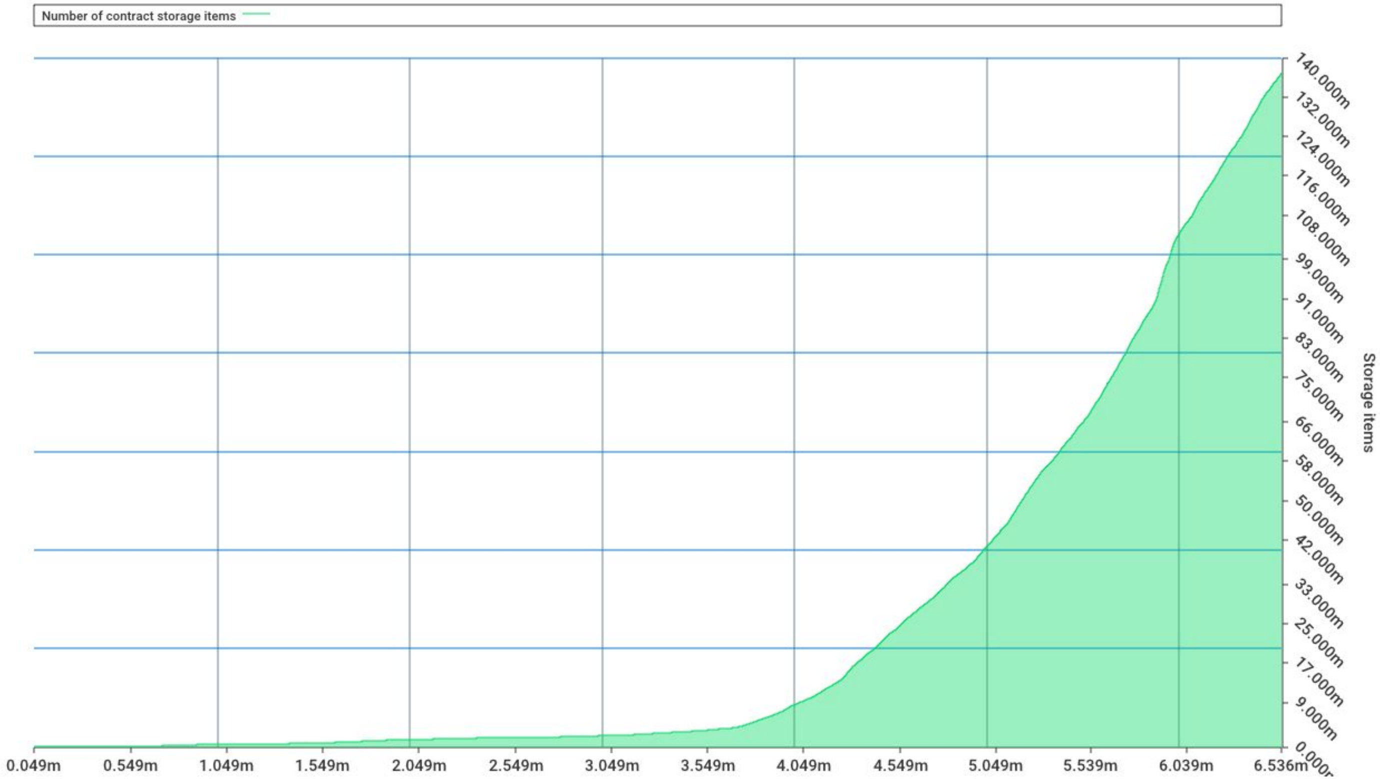
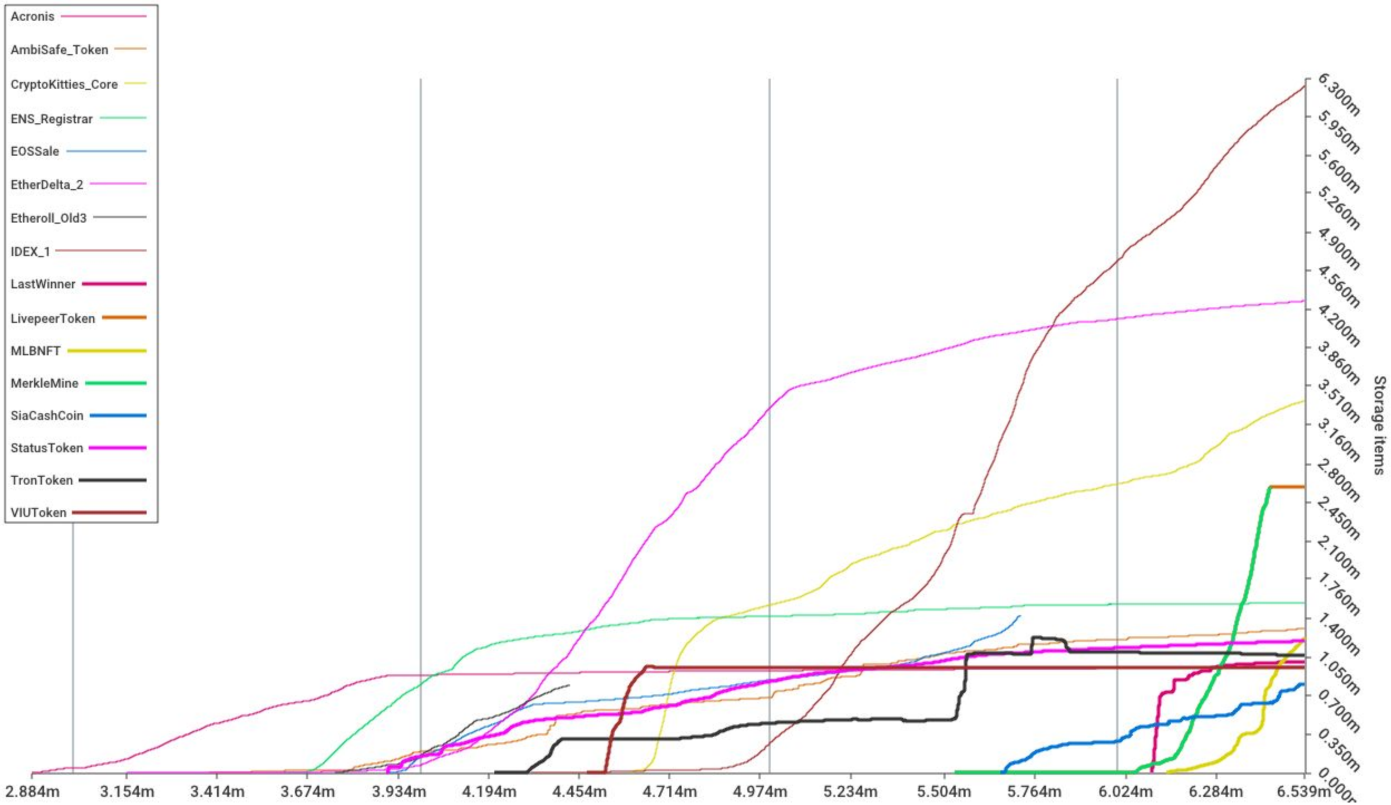


Chart 3 - top 16 contracts by storage items



Comments to Chart 3

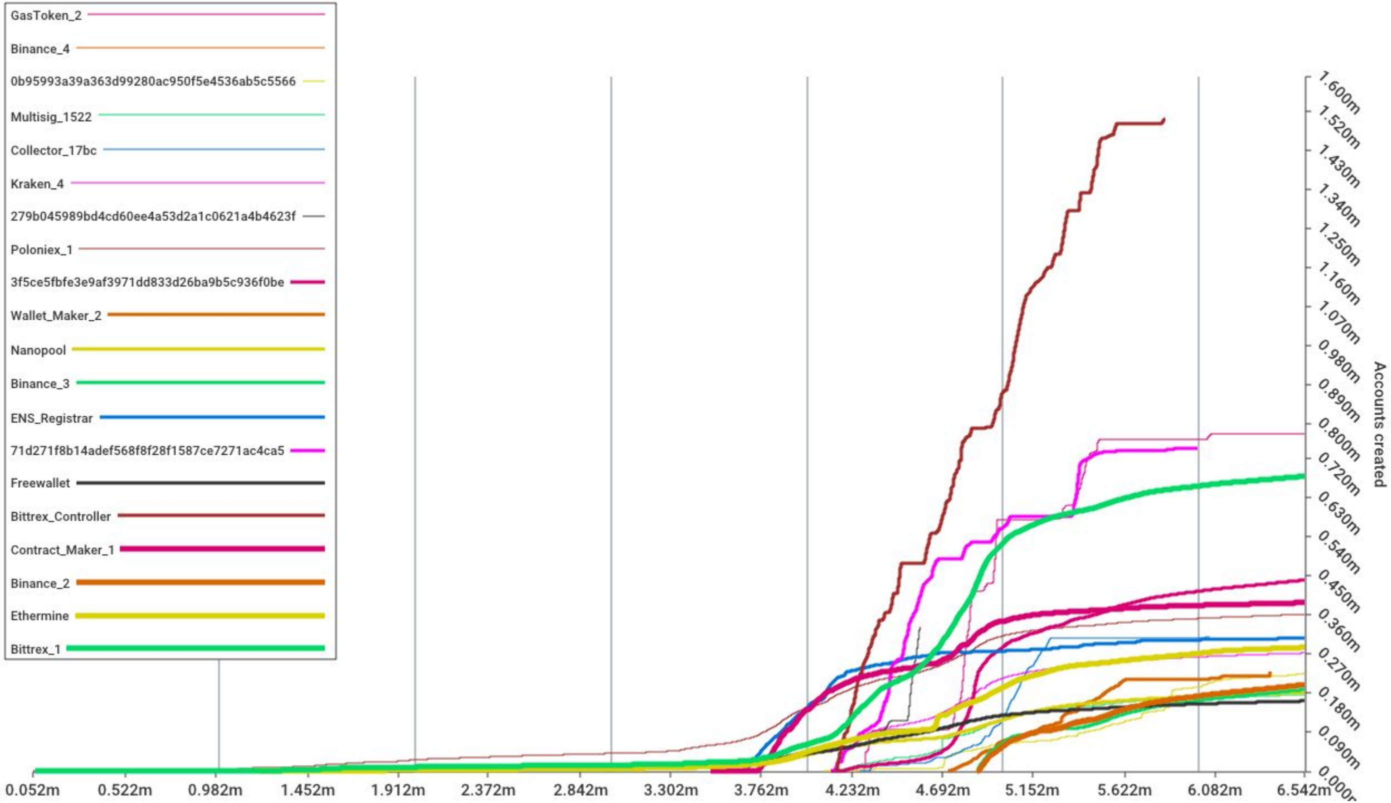
Lines stopping at a block before reaching the right side of the chart means that the contracts were not used since that block

Three largest active contracts are IDEX_1, EtherDelta_2, and CryptoKitties. They occupy 10% of the storage.

Top 216 occupy 50% of the storage, top 421 - 60%, 806 - 70%, 1866 - 80%, 16k - 90%. Total number of contracts in the state is around 7m

Apart from storing tokens balances, contract storage is often used as write-only sets

Chart 4 - top 20 creators of contracts and EOAs



Comments to the Chart 4

GasToken-2 (working by creating self-destructing contracts and not by reserving storage as GasToken-1) is the largest active creator of accounts at the moment.

The biggest creator (Bittrex_Controller) seem to have stopped, leaving all created contracts in the state. Current Bittrex creator is the second largest active creator.

This chart does not show deposit non-contract accounts belonging to exchanges, because their identification requires a bit more advanced heuristics

Chart 5 - number of EOAs (Y) with <= X wei

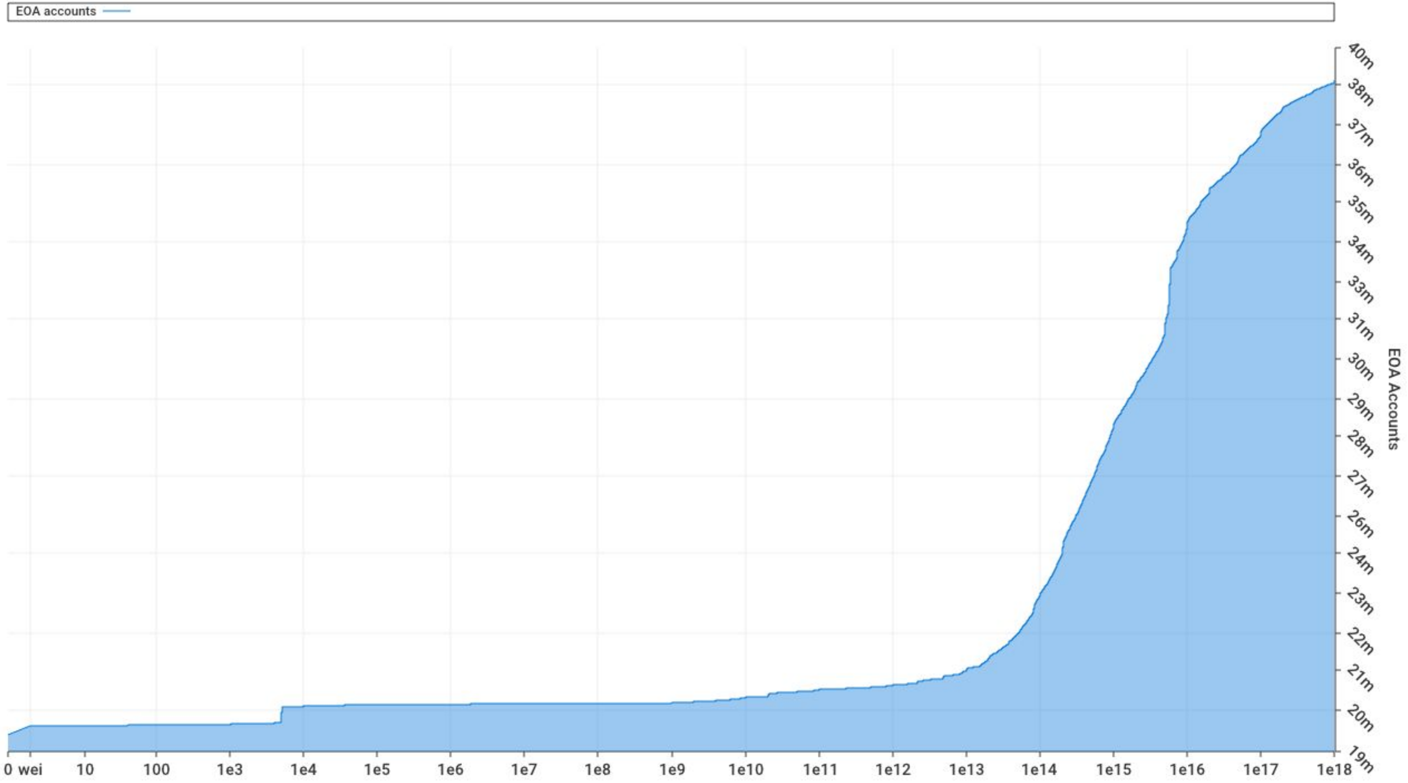


Table1 - top 100+ contracts by storage

	Contract name	Storage items	%, cumulative
0	IDEX_1	6687388	4.633
1	EtherDelta_2	4320148	7.625
2	CryptoKitties_Core	3585370	10.109
3	LivePeer_Token	2598735	11.909
4	MerkleMine	2598081	13.709
5	ENS_Registrar	1570676	14.797
6	MLBNFT	1495249	15.833
7	EOS_TokenSale	1428756	16.822
8	Ambisafe_EToken2	1325533	17.741
9	StatusToken	1213111	18.581
10	TronToken	1067753	19.321
11	LastWinner	1030760	20.035
12	Acronis_Contract	976385	20.711
13	VIUToken	956662	21.374
14	SiaCashCoin	806085	21.932
15	Etheroll_Old3	796685	22.484
16	XENON_Token	748143	23.002
17	OmiseGo_Token	673952	23.469
18	Dex.top_Dex2	609900	23.892
19	Etheroll	593935	24.303
20	BTCRelay	586440	24.709
21	NePay_Token	576136	25.108
22	Ethermon_CastleBattle	558198	25.495
23	CryptoKitties_SalesAuction	556393	25.88
24	Etheroll_Contract	551113	26.262
25	Etheroll_Old4	526288	26.627
26	Player_Tokens	517965	26.986
27	IG_Token	508063	27.338
28	0xProtocol_Exchange1	503683	27.686
29	VinToken	478918	28.018
30	DodgersNFT	478862	28.35
31	AirDropToken	472082	28.677

Table 1 - top 100+ contracts by storage

32	District0xNetworkToken	451617	28.99
33	Streamr_Token	435625	29.292
34	Mysterious_Minter?	432693	29.591
35	XID_Registrar?	430191	29.889
36	GasToken_Fake?	397153	30.164
37	BitcoinEOS_Token	391044	30.435
38	CarLiveChain_IOVToken	382081	30.7
39	BeautyChain_Token	364849	30.953
40	Aragon_Token	355470	31.199
41	OasisDEX_MatchingMarket	352353	31.443
42	EOS_DSToken	346414	31.683
43	GodsUnchained_CardMigration	338556	31.918
44	flamingostar_Token	337699	32.152
45	Ethermon_Data	331396	32.381
46	AICToken	322719	32.605
47	UCashToken	318859	32.826
48	GSENetwork_Token	304999	33.037
49	BinanceToken	300855	33.245
50	IONChain_Token	292288	33.448
51	GSG_Coin	285341	33.645
52	ENS-EthNameService	280371	33.84
53	NYBCoin	261406	34.021
54	VGAMES_Token	259765	34.201
55	Sachio_Token	254853	34.377
56	MST_Token	244438	34.546
57	IdleEth_Game	241961	34.714
58	Dice2Win_2	241144	34.881
59	Cindicator_Token	239766	35.047
60	FunKoin	236755	35.211
61	Enumivo_Token	234164	35.373
62	SoPay_Token	230085	35.533
63	QMQ_Token	225352	35.689

Table 1 - top 100+ contracts by storage

64	TokenStore	224492	35.844
65	Wei_Reciever_Jan2016?	222009	35.998
66	BitClave_Token	221158	36.151
67	OCoin	219945	36.304
68	Lottery?	217965	36.455
69	StorjToken	208745	36.599
70	Edgeless_Blackjack	200598	36.738
71	BOBsRepairToken	195328	36.874
72	PundiXToken	193914	37.008
73	DecenturionToken	193222	37.142
74	BeautyChain_Token?	192946	37.275
75	FoMo3Dlong	191060	37.408
76	Envion_Token	187417	37.538
77	DataToken	184898	37.666
78	PentaNetwork_Token	178788	37.79
79	IDAG_Token	175925	37.911

80	79048730d53691268249fc0275f70af9046c3134	175556	38.033
81	EMO_Token	173649	38.153
82	TimeNewBank_Token	173486	38.274
83	TrekChain_Token	172978	38.393
84	AVINOCToken	171073	38.512
85	DRC_Token	170356	38.63
86	EthLend_Token	166042	38.745
87	06a6a7af298129e3a2ab396c9c06f91d3c54aba8	164158	38.859
88	BITDINERO_Token	162927	38.971
89	BroFistCoin	162900	39.084
90	Rebellious_Token	162781	39.197
91	Dice2Win_1	159506	39.308
92	MCPSale_Token	158369	39.417
93	IFoods_Token	155392	39.525
94	CryptoKitties_SiringAuction	154183	39.632
95	OrmeusCoin	153266	39.738

Table 1 - top 100+ contracts by storage

96	ForAgricultureCoin	152892	39.844
97	GenaroX_Token	152616	39.95
98	f87e31492faf9a91b02ee0deaad50d51d56d5d4d	152479	40.055
99	Academicon	152453	40.161
100	SuperEdge_Token	151973	40.266
101	EtherBots?	150447	40.37
102	BrahmaOS_Token	146748	40.472
103	KickCoin_CSToken	146328	40.573
104	StorJ_Issuer	144444	40.673
105	HashPowerToken	142949	40.772
106	OMTM_Token	142849	40.871
107	5371a8d8d8a86c76de935821ad1a3e9b908cfced	142803	40.97
108	FTI_Token	141886	41.069
109	EC_Token	141617	41.167
110	ThreeDBToken	141595	41.265
111	e694010c4f1fcd35ebc04ceb60f847caaf2cd6f2	141591	41.363

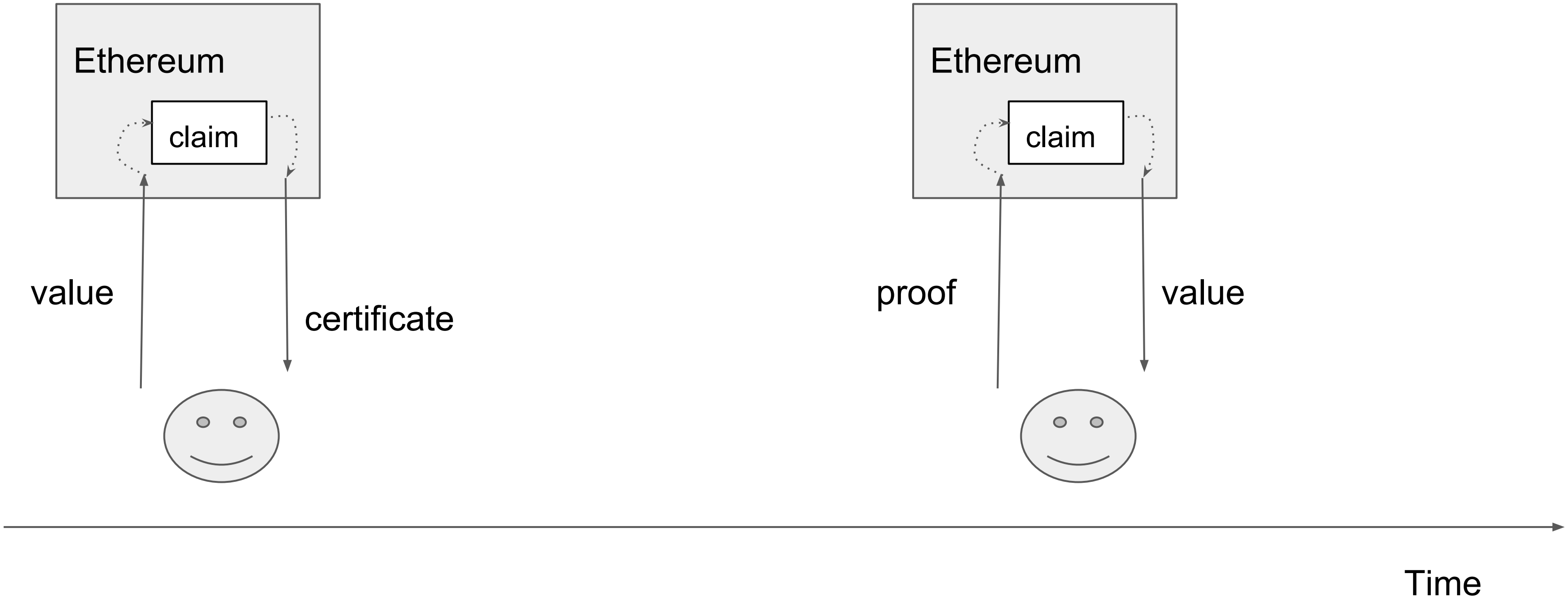
112	DACC_Token	139631	41.46
113	eddbit	139409	41.556
114	SilentNotary_Token	139408	41.653
115	XMAX_Token	139289	41.749
116	CryptoSpinners	137952	41.845
117	HadesCoin	137592	41.94
118	Etheroll	135270	42.034
119	007ac2f589eb9d4fe1cea9f46b5f4f52dab73dd4	133907	42.127
120	ExTradeCash_Token	133521	42.219

Comments to Table 1

Some notable contracts, like Status and Aragon tokens use much more storage than 1 word per holder. That is because they are based on MiniMe token, which stores the entire history of transfers in the contract state. It needs to research to see how the state would shrink if these were to become non-MiniMe tokens.

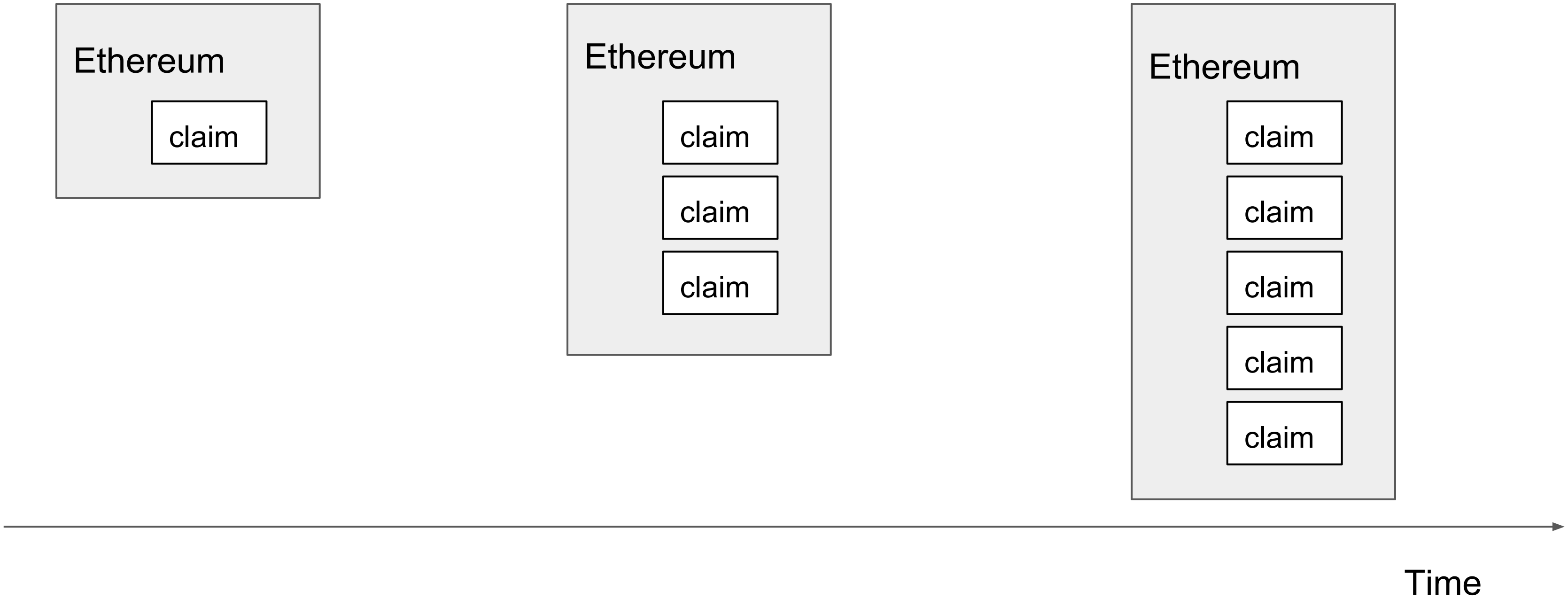
Problem statement - prevalent use case

Following the data analysis shown before, we will focus on the seemingly prevalent use case of Ethereum as a claim storage system.



Problem statement - no claim maintenance

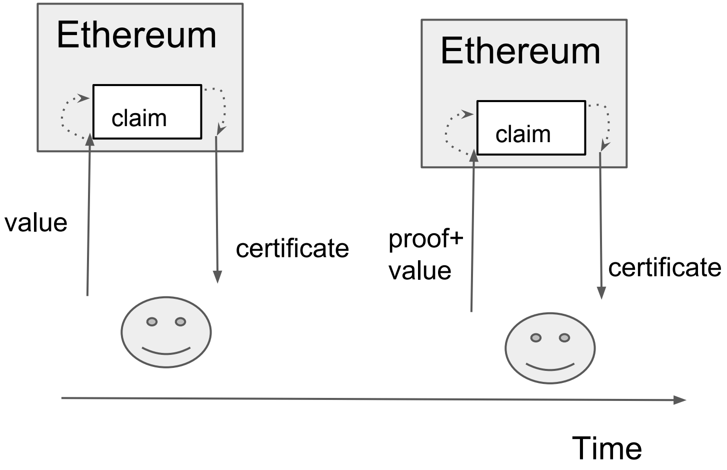
Claims do not need to be maintained if Ethereum state is allowed to grow without bounds:



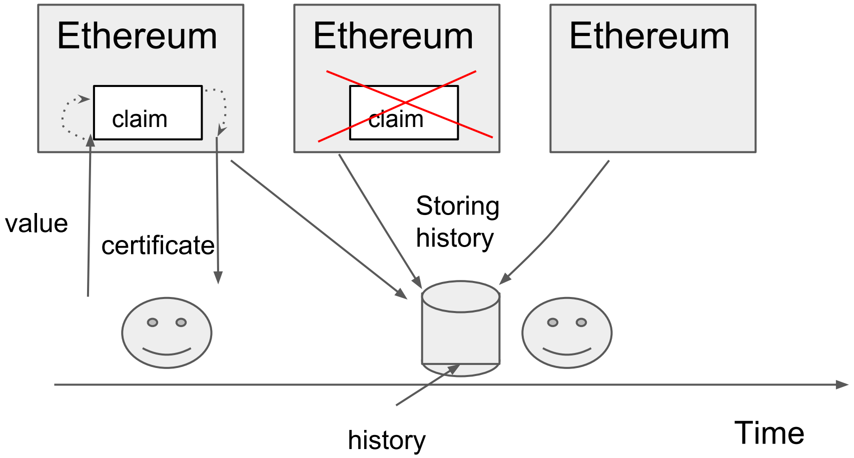
Problem statement - bounded state

If Ethereum state becomes bounded, claims will require maintenance (hypothesis, intuition). With active maintenance, users need to regularly interact with Ethereum and add value to renew their claims. With reactive maintenance, users need to keep storing the history of Ethereum state to keep their proof of their claims verifiable.

Active maintenance



Reactive maintenance



Problem statement - solutions

Active and reactive maintenance approaches are not mutually exclusive, and they most probably need to coexist.

Reactive maintenance solutions could be described as “witness-based” techniques, “stateless” contracts, and they are most probably implementable today without any modifications to the Ethereum protocol.

Active maintenance solutions are described here in form of linear cross-contract storage and alternative ideas, which are less radical, but also less general.

Main position

Note that this part of the position is not unanimous, and currently there are alternative points of view within the working group.

Given that the biggest class of storage users are currently token contracts and NFTs, the existing model of storage is incompatible with rent for two main reasons:

Reason 1: Ownership of storage (currently with token contracts) is not aligned with its utility (currently with the token holders). That creates free-riding problem (holders are not incentivised to contribute to the contract's rent), and it would very challenging for contracts to collect rent from the holders. Alternative point of view: It is possible to get token holders to contribute to the rent by offering tokens in exchange for the contributions.

Main position

Reason 2

Token dust griefing attack. Any token holder with access to transfer function can increase storage rent for the contract forever, paying only once for the attack. Alternative point of view: Tokens could require a signed ACK (consent for the receipt of tokens) from recipients. Those are to be provided during transfer. In combination with minimum transfer and minimum holding limits, this could prevent token dust griefing attack. There will have to a special handling for contracts that would like to be token holders, perhaps an ABI-based standard.

Main position

It is expected that after introduction of the full rent, most of the contracts that exist today will be non-viable and vulnerable to the token-dust griefing attacks.

If the notion of rent is introduced without a “safe place” to migrate to, the only recourse of current contracts would be to use reactive maintenance approach, in the form of stateless contracts, which might too big of a leap in terms of usability.

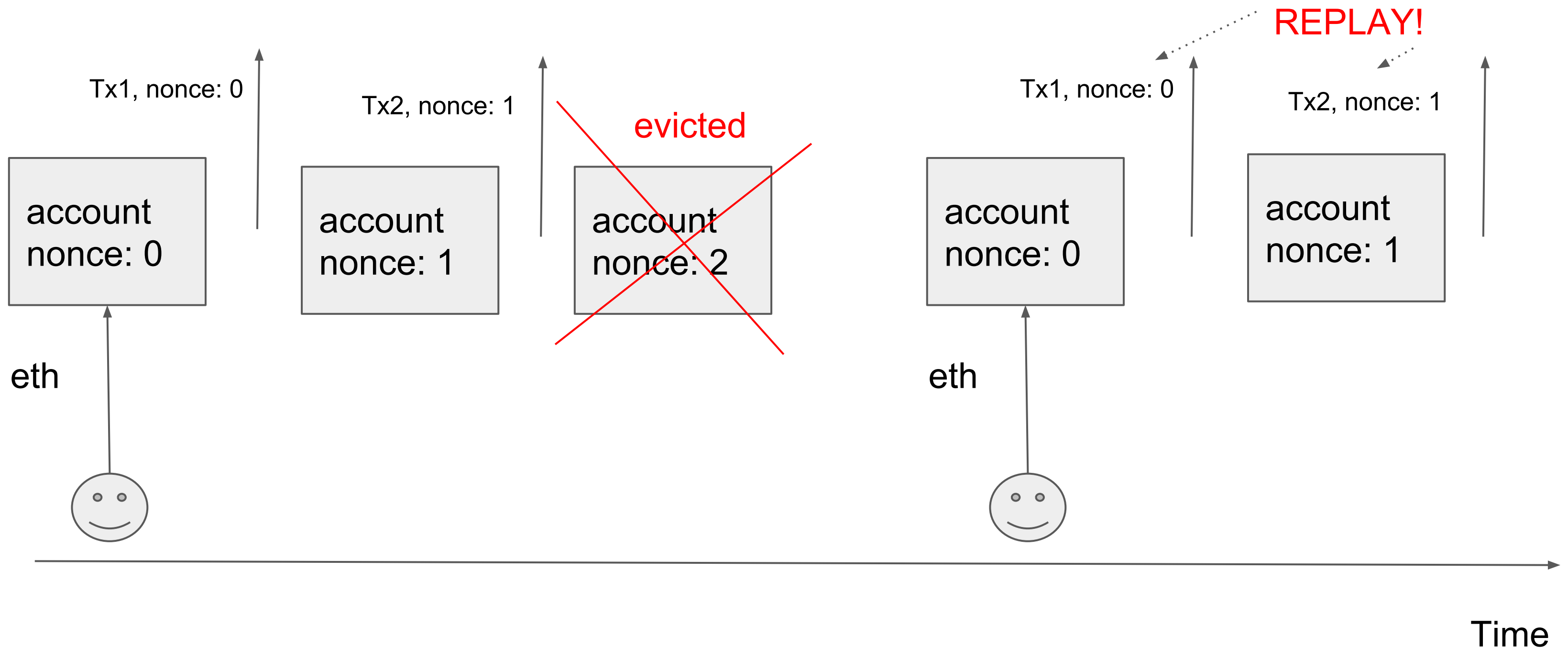
In Step 3, such a “safe place” is described, in a form of a new storage model (following active maintenance approach). This (or an alternative “safe place” solution, if found) needs to be introduced before storage rent for the entire storage (currently at Steps 4 and 5).

Main position

Alternative point of view: fungible token contracts can be modified to be viable, as described earlier. Case of Non-Fungible assets needs to be researched.

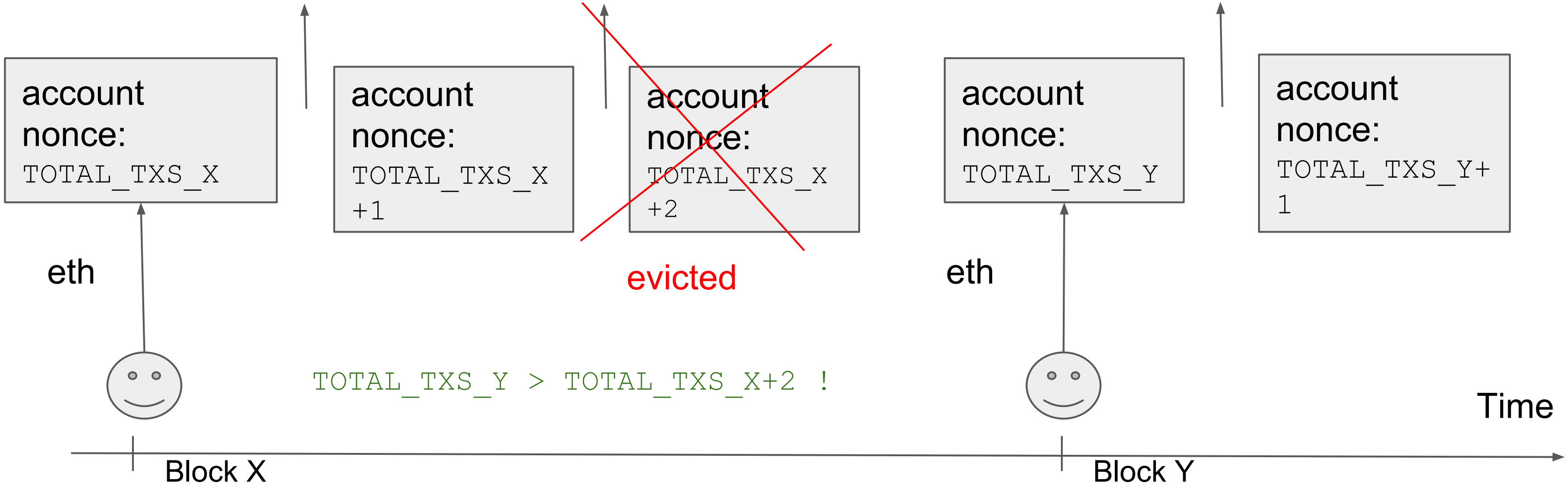
Step 1 - replay protection for new accounts

Eviction creates a replay problem when account is recreated



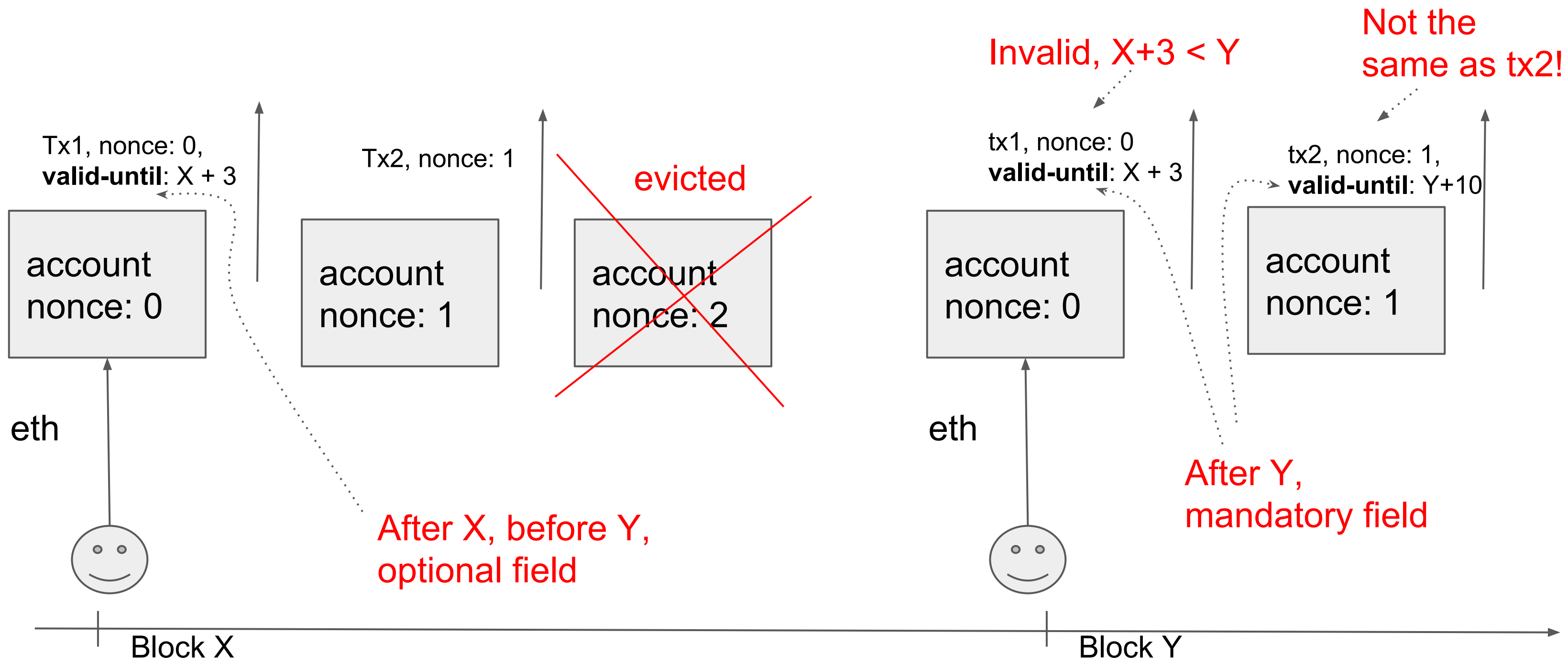
Step 1, Variant 1 - nonce based

Newly created accounts now have a nonce equal to `TOTAL_TXS` instead of zero. Value `TOTAL_TXS` becomes part of the state. All client implementations calculate the correct value of `TOTAL_TXS` upon the upgrade, using blocks bodies (for full sync) or receipts (for fast sync), with some checkpoint values hard-coded.


account nonce: TOTAL_TXS_X
account nonce: TOTAL_TXS_X +1
account nonce: TOTAL_TXS_X +2
evicted
account nonce: TOTAL_TXS_Y
account nonce: TOTAL_TXS_Y+ 1
eth
eth
TOTAL_TXS_Y > TOTAL_TXS_X+2 !
Time
Block X
Block Y

Step 1, Variant 2, temporal protection

EIP draft: https://gist.github.com/holiman/5300039af83375e1698117619554acf7



Step 2 - Fixed small rent on Externally Owned Accts

For example, 2 Gwei per account per block. That means, in a year, the rent would be 0.001 ETH per account. ETH paid as rent gets burnt.

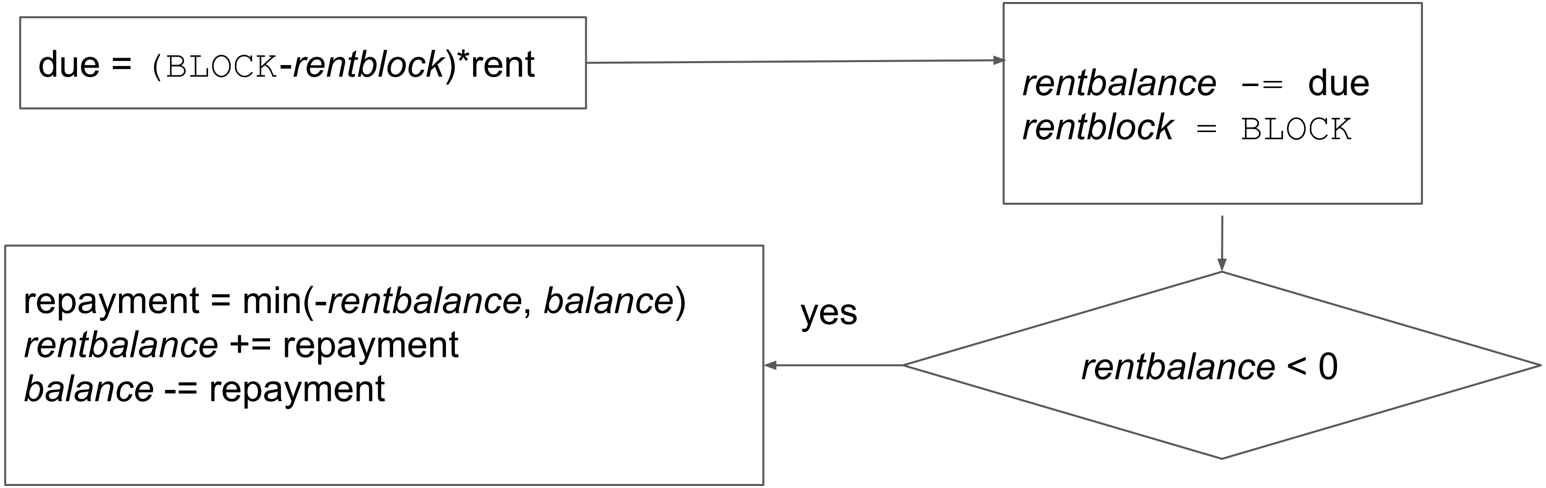
When rent is not paid, account is removed from the state, so Step 1 is for replay protection.

If no one paid that rent, in a year, that would eliminate around 28 million (Chart 5) of dust accounts, roughly 15% of the current state (Charts 1 and 2).

Every non-contract account now has 2 extra field: *rentblock* (last time rent was calculated), with default = STEP2_HARD_FORK_BLOCK, and *rentbalance*, which can be negative

Step 2 - calculation of dues

Calculation of dues happens whenever account is modified by EVM. Both *rentbalnce* and *balance* fields of an account may be updated as a result.



due = (BLOCK-*rentblock*)*rent

rentbalance -= due
rentblock = BLOCK

rentbalance < 0

yes

repayment = min(-*rentbalance*, *balance*)
rentbalance += repayment
balance -= repayment

Step 2 - Priority queue (eviction)

During the initial sync, when client software receives current state from other nodes in the network, it calculates this value (as a rational number, for determinism) for each account:

$$\text{advance} = \frac{\mathit{balance} + \mathit{rentbalance}}{\mathit{storagesize}} - [\text{cumulative rent since } \mathit{rentblock}]$$

For non-contract accounts, *storagesize* is fixed. Cumulative rent is calculated since *rentblock* until the block of the initial sync.

Step 2 - Priority queue

The value of “advance” determines how much rent has to accumulate from the block of initial sync to some future block, for the account to go into arrears.

Accounts are placed into a priority queue ordered by “advance” with some tie-breaking condition, for example, using ordering of address hashes.

Now, if an account is modified, its fields are updated by the rent recalculation, so the account is removed from the priority queue, its “advance” is recalculated, and the account is placed back into the priority queue. Note that *rentblock* will now be larger than block of initial sync, therefore, [cumulative rent since *rentblock*] will be negative.

Step 2 - Priority queue

Each block, the top of priority queue is checked, and a limited number of accounts is removed, if they are in arrears. Condition of being in arrears is calculated as:

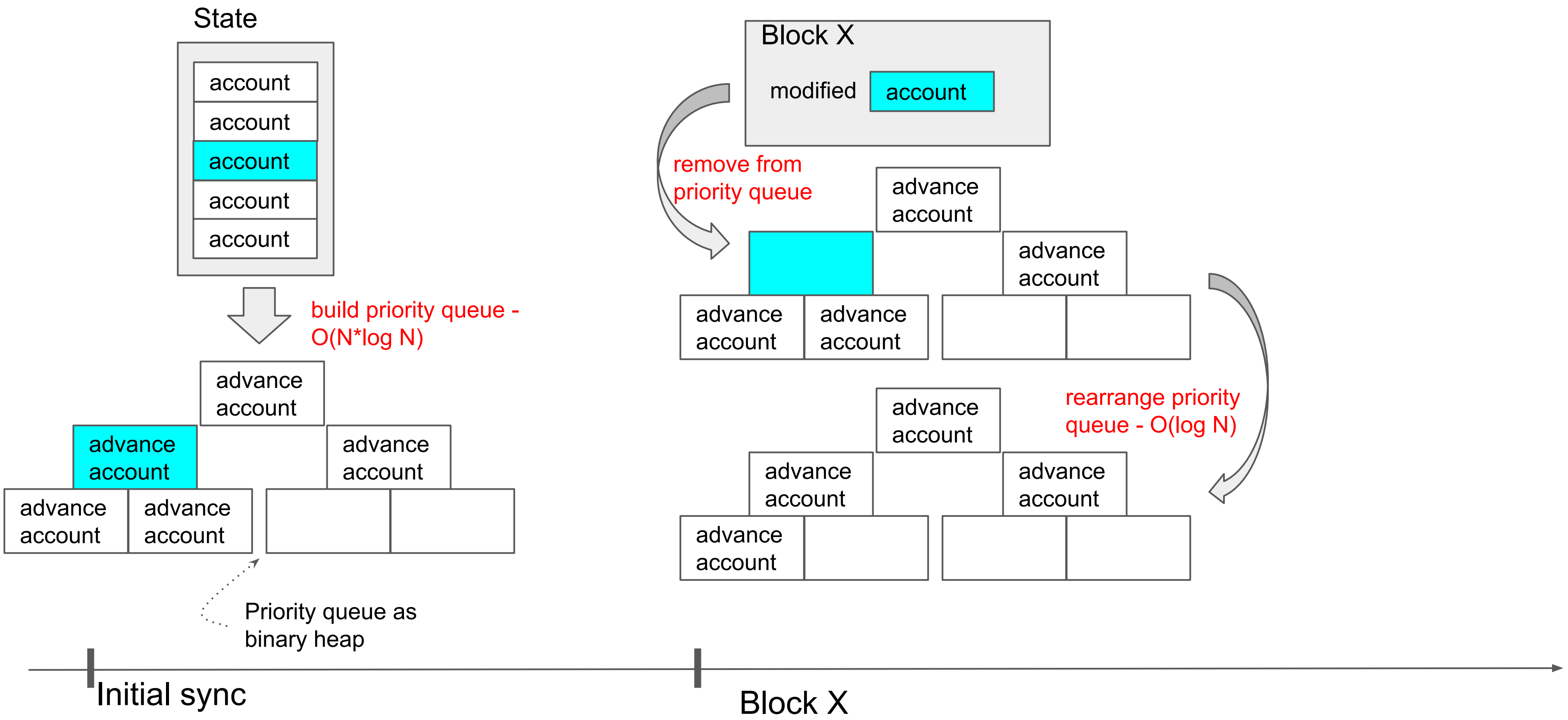
[cumulative rent since init sync block] > advance.

Hypothesis - even though different client instances will have different values of init sync block, and assign different “advance” values to accounts, the order in which the accounts will be removed from the top of the priority queue, will be the same across all the instances, and predictable prior to the next block (so that miners cannot use that as an attack vector against other miners).

Step 2 - Priority queue

For each account popped from the priority queue, a transaction is added to the block, in which eviction is happening. At Step 3, this transaction will also make a callback to the `<writer>` contract to notify of the eviction. The eviction transactions do not need to be transmitted over the network, but generated by full clients by the rules of the protocol. Eviction receipts might need to be served to light clients, though.

Step 2 - Priority queue: init and modifications



State

account

account

account

account

account

build priority queue - O(N*log N)

advance account

advance account

advance account

advance account

advance account

Priority queue as binary heap

Block X

modified account

remove from priority queue

advance account

advance account

advance account

advance account

rearrange priority queue - O(log N)

advance account

advance account

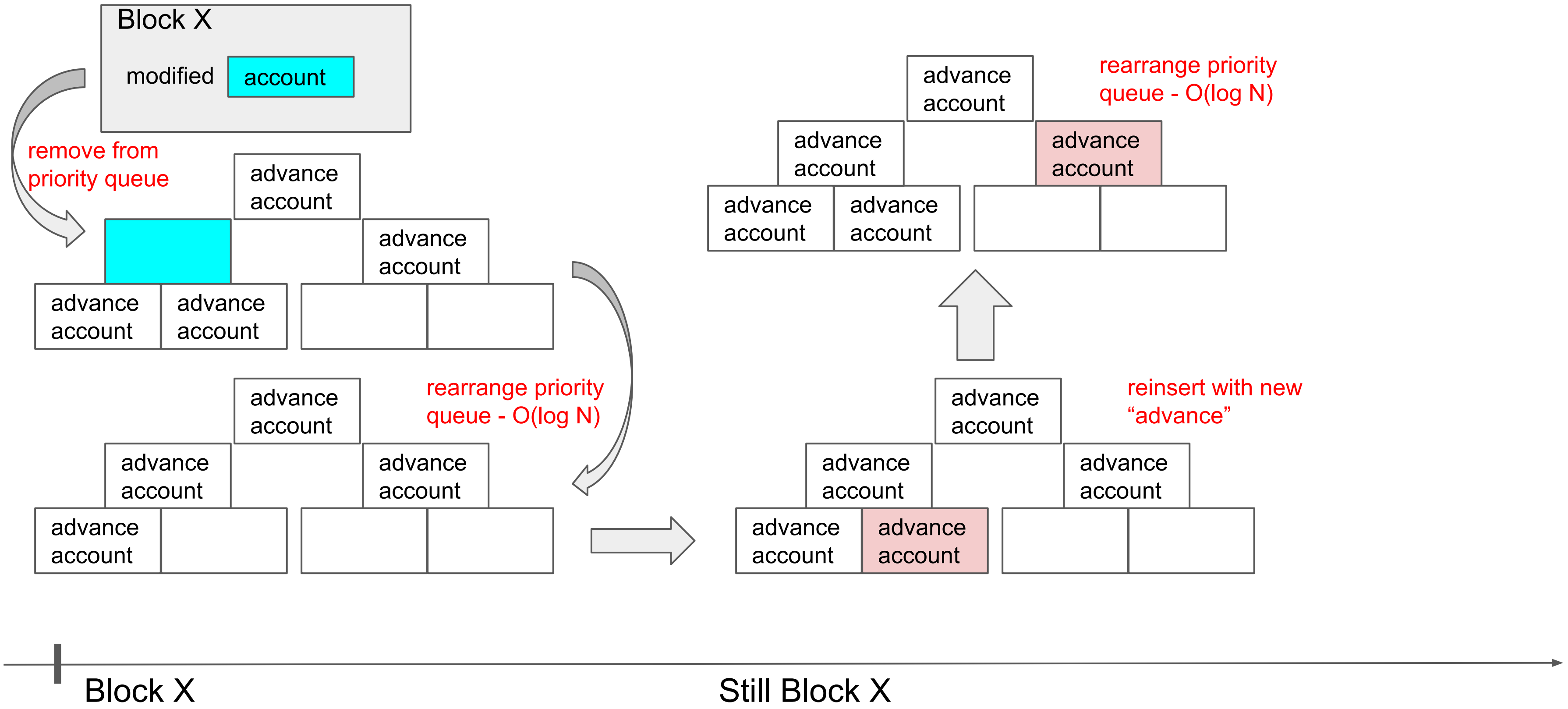
advance account

advance account

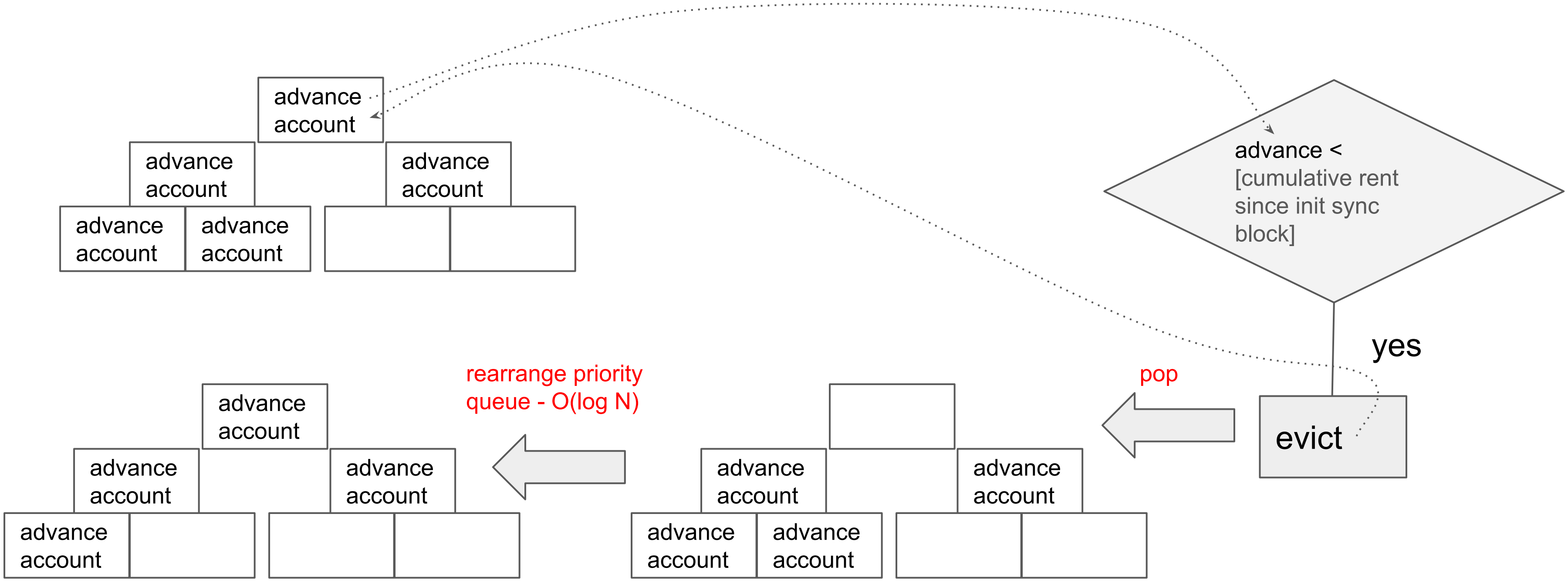
Initial sync

Block X

Step 2 - priority queue: modifications



Step 2 - priority queue - evictions



Step 2 - priority queue: efficiency

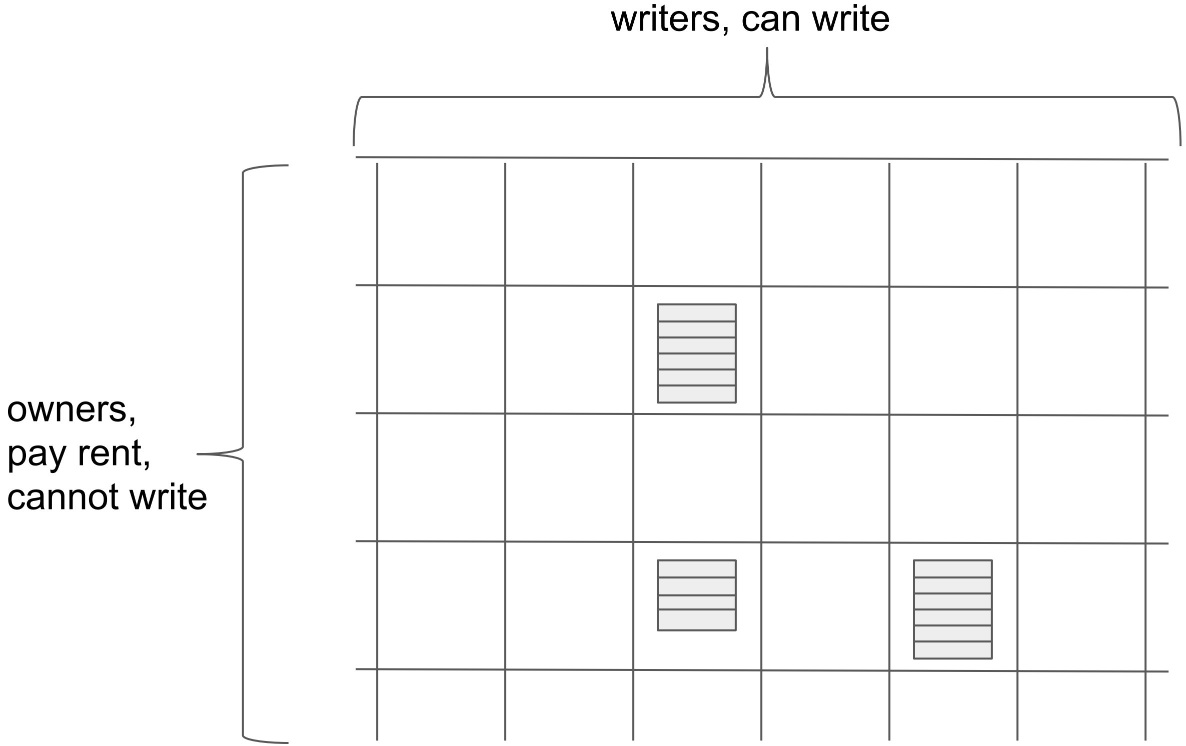
Rent recalculation only happens when accounts are modified (as opposed to when they are “touched”).

Priority queue requires O(N*log N) time to build up during the initial sync, where N - number of accounts in the state.

After that, every modification of an account, or an eviction requires O(log N) operation on the priority queue.

Therefore, it is believed that this priority queue design does not bring significant overhead, both in computation and in number of transactions.

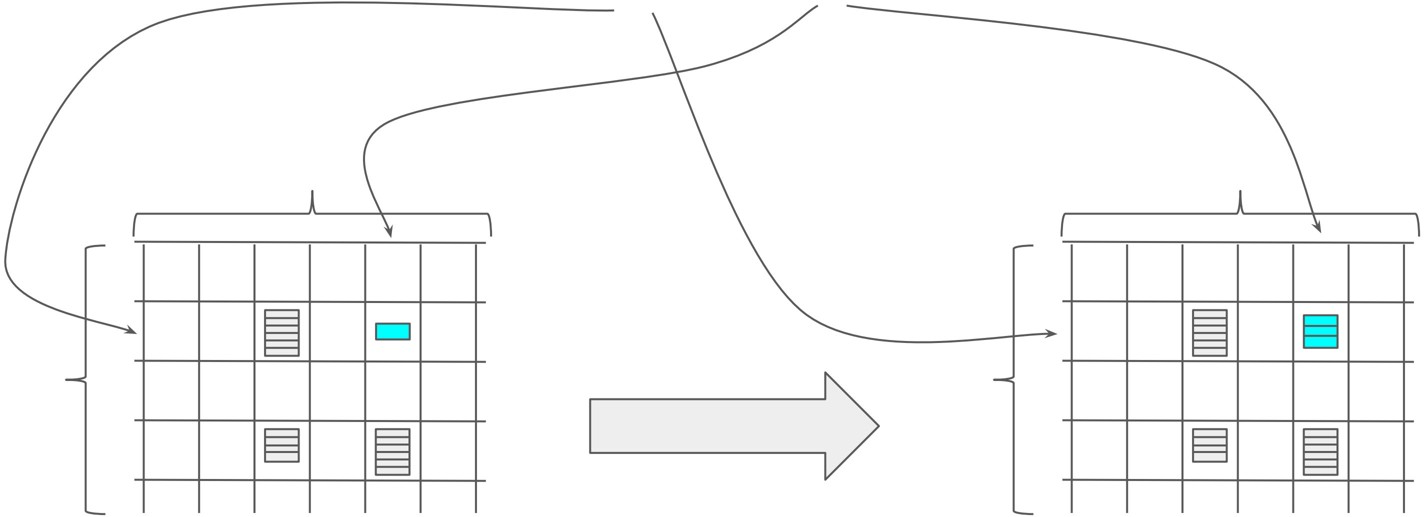
Step 3 - Linear cross-contract storage



Only contracts can be owners and writers in this model!

Step 3 - Linear cross-contract storage - opcodes

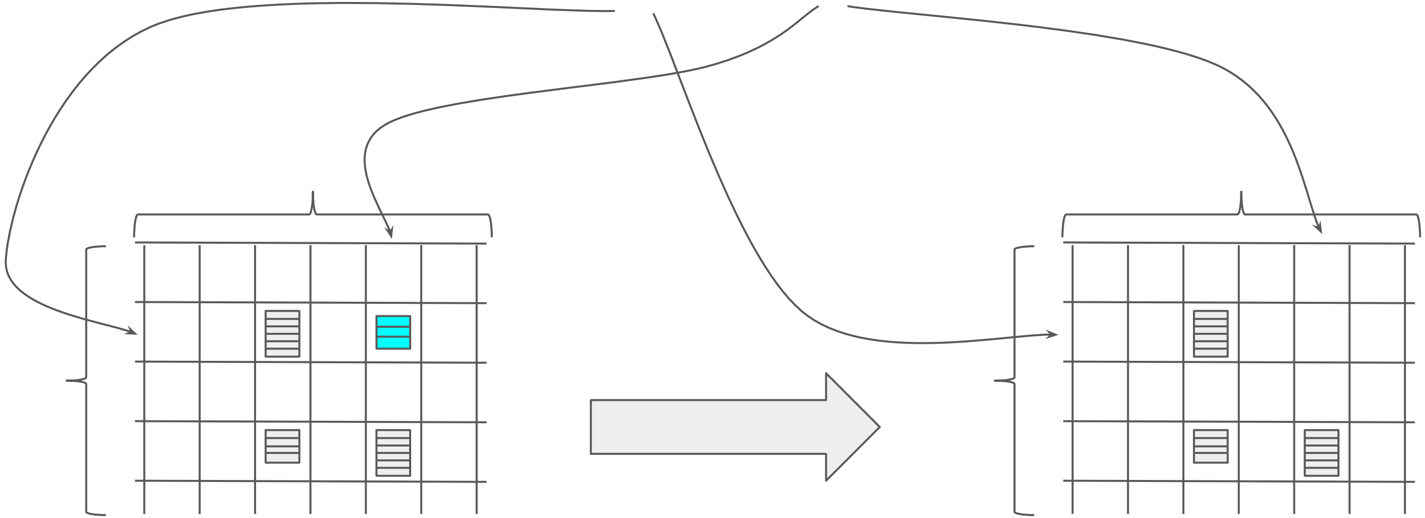
```
XGROW <owner> <writer> <growth>
```



If executed by the **owner** of the cell, and within limits, succeeds and returns the size of the cell after resize. Otherwise, fails, and returns 0

Step 3 - Linear cross-contract storage - opcodes

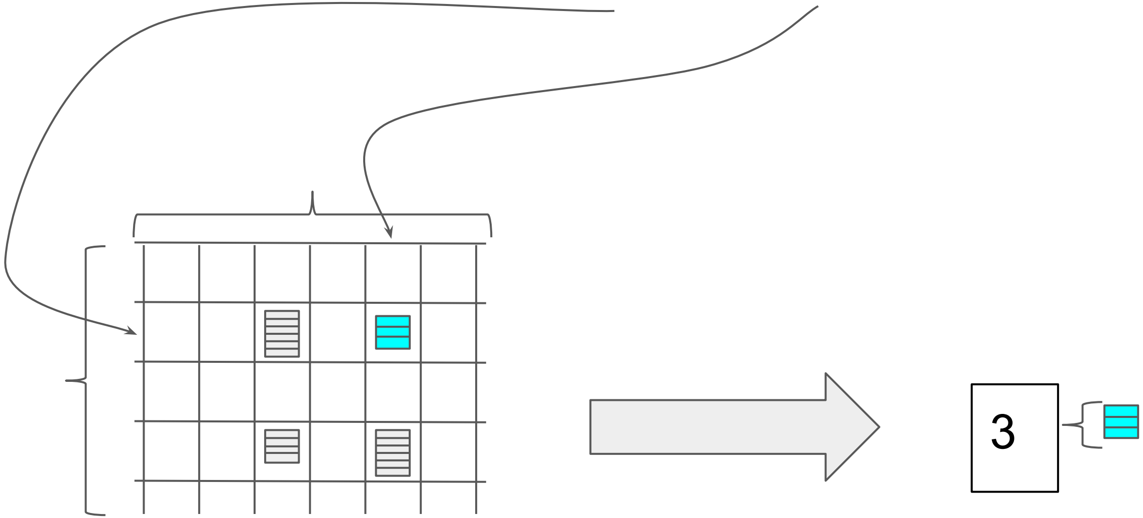
`XCLEAR <owner> <writer>`



If executed by the **owner** of the cell, clears the cell. Otherwise, silently fails

Step 3 - Linear cross-contract storage - opcodes

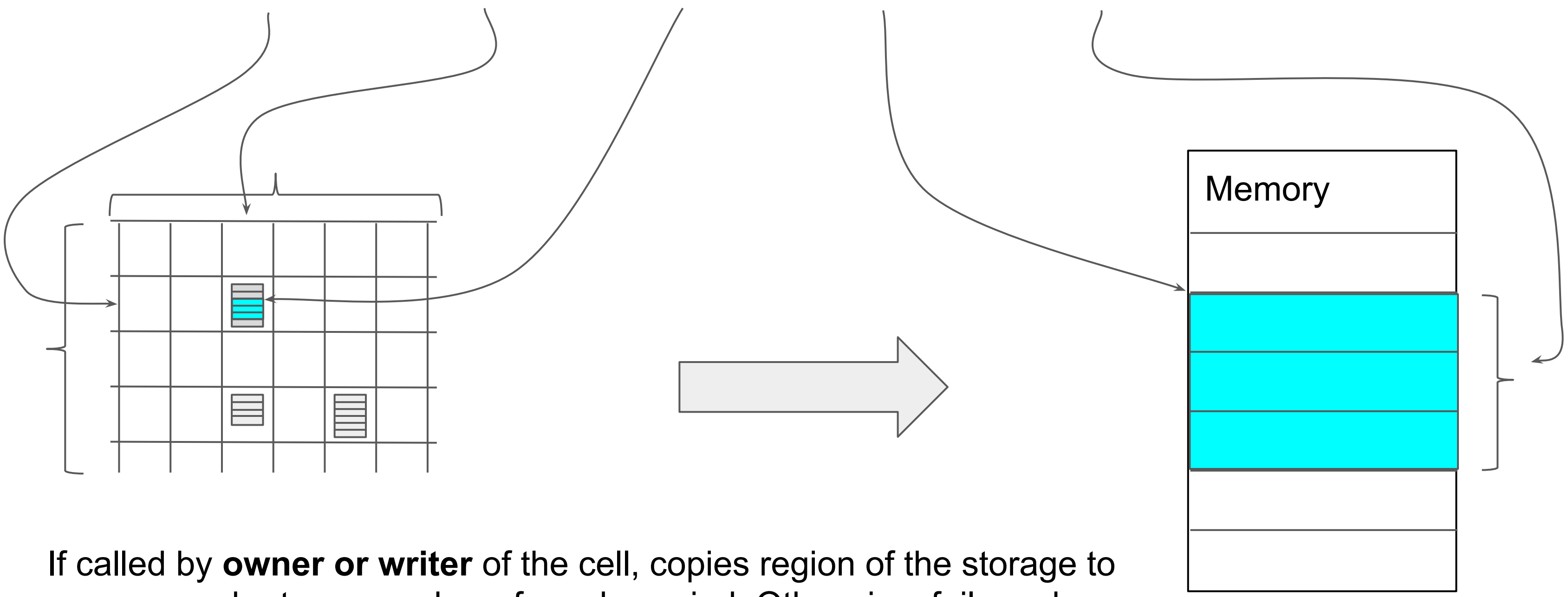
```
XSIZE <owner> <writer>
```



Succeeds if called by **owner or writer** of the cell, and returns size of the cell.
Otherwise, fails and returns 0

Step 3 - Linear cross-contract storage - opcodes

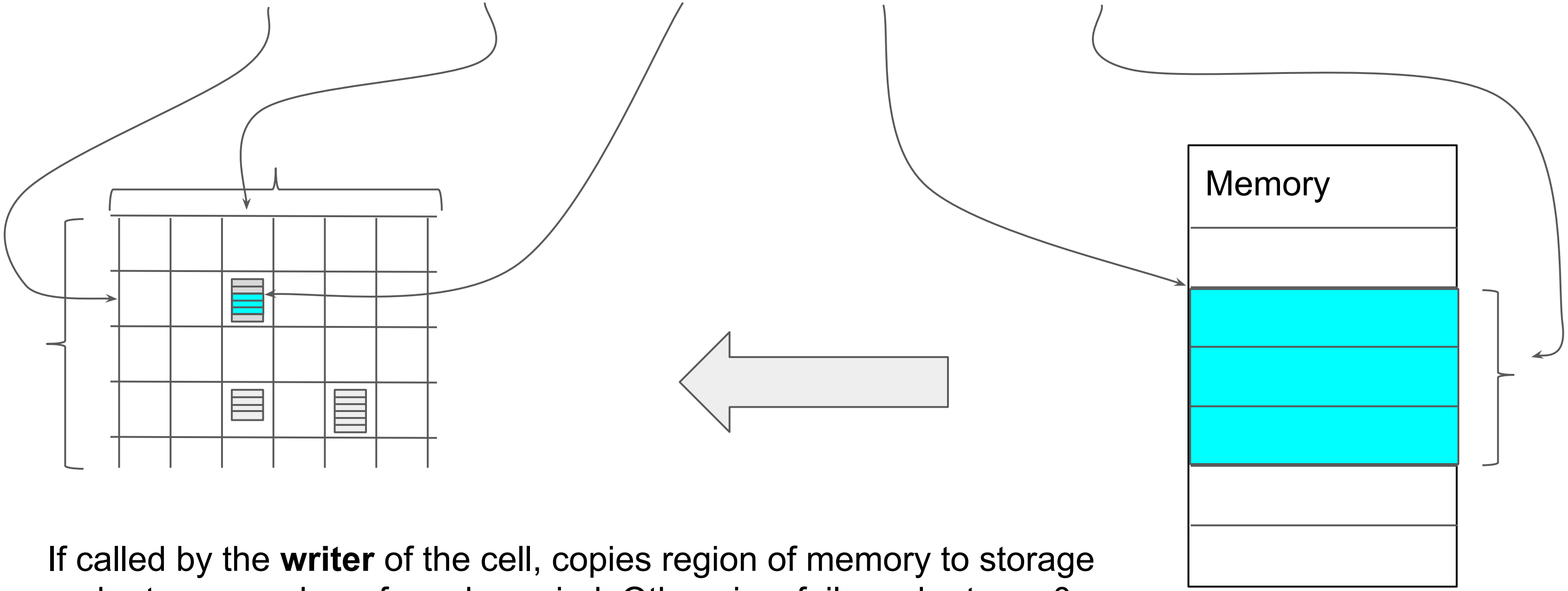
```
XREAD <owner> <writer> <strg_pos> <mem_pos> <size>
```



If called by **owner or writer** of the cell, copies region of the storage to memory and returns number of words copied. Otherwise, fails and returns 0

Step 3 - Linear cross-contract storage - opcodes

```
XWRITE <owner> <writer> <strg_pos> <mem_pos> <size>
```



If called by the **writer** of the cell, copies region of memory to storage and returns number of words copied. Otherwise, fails and returns 0

Step 3 - new account fields

Contract accounts get 3 additional fields: (*rentblock*, *rentbalance*, *storagesize*).

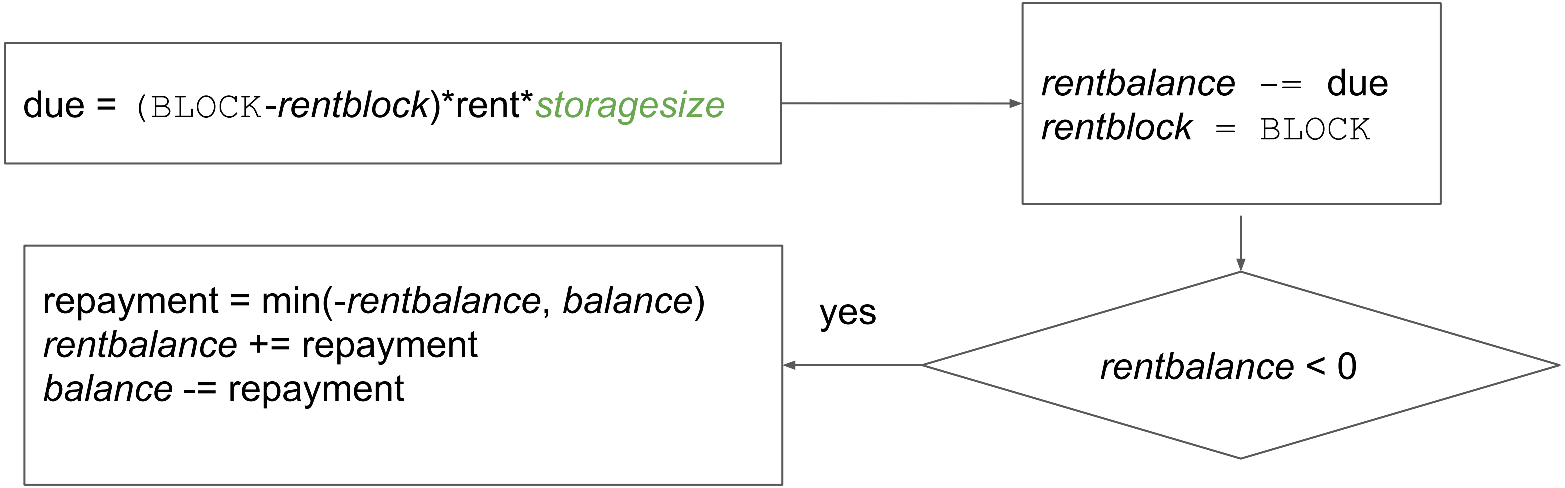
storagesize field of a contract now includes

Total size of linear cross-contract storage with `<owner>`==contract’s address

New opcode `SSIZE` can read *storagesize* field. Default for *rentblock* for contracts is STEP3_HARDFORK_BLOCK

Step 3 - calculation of dues

Rent is now calculated not per account, but per unit of *storagesize* field. Price per unit is still fixed (for example, 1Gwei per block per item).



due = (BLOCK-*rentblock*)*rent**storagesize*

rentbalance -= due
rentblock = BLOCK

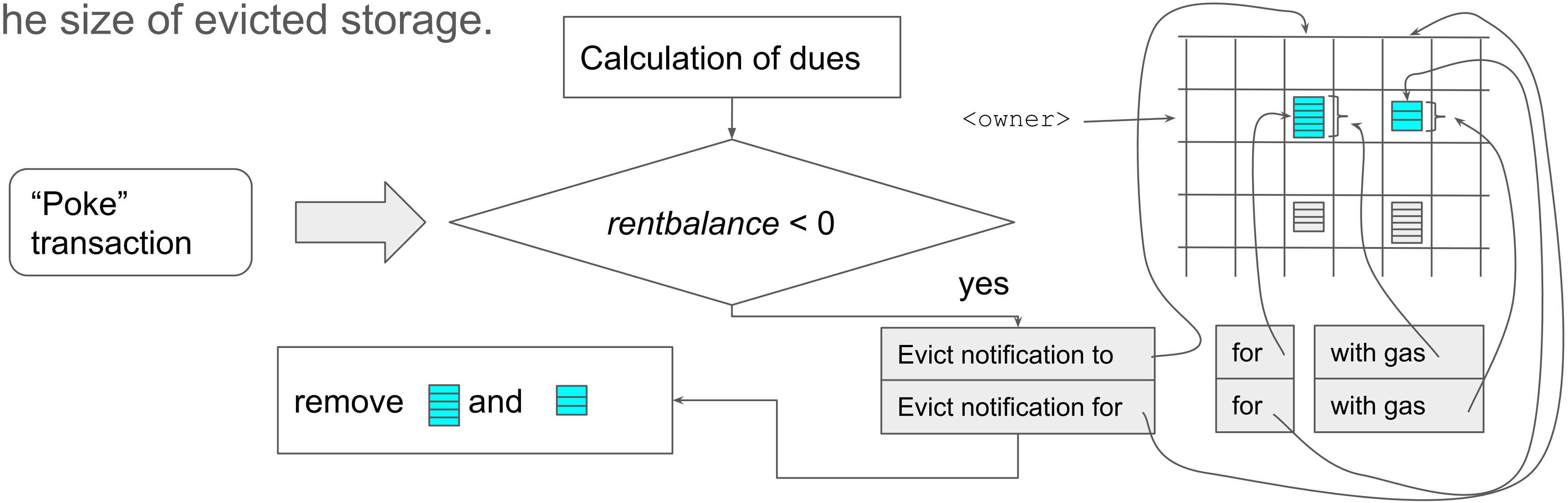
rentbalance < 0

yes

repayment = min(-*rentbalance*, *balance*)
rentbalance += repayment
balance -= repayment

Step 3 - eviction

When rent is not paid, contract loses linear cross-contract storage it owns for all writers, and that storage cannot be resurrected (only until Step 4). In an eviction transaction, `SYSTEM` account makes a call to the `<writer>` contract with the call data containing the address of the `<owner>` contract, with a stipend proportional to the size of evicted storage.



"Poke" transaction

Calculation of dues

rentbalance < 0

yes

Evict notification to

Evict notification for

for

with gas

for

with gas

remove ▤ and ▤

`<owner>`

Step 3 - gathering rent for the code

Since contracts need to pay rent for maintenance of their account, and their code, the free-riders problem does not completely go away. In order keep totally unmanned contracts viable and also promote code reuse, contracts have an additional parameter: *callfee*. When set (most probably during deployment), each invocation of the contract is charged extra fee equal to *callfee*. This is added to the contract's *rentbalance*, and cannot be turned back to ETH.

Setting *callfee* is done via a new opcode `CALLFEE`

There can be a way for waive the calling fee if the *rentbalance* is above some threshold, but this needs to be researched.

Step 3 - motivational example

In the model of linear cross-contract storage, a token contract would be the `<writer>`, whereas token holders would be `<owner>`s. Owners consent to be given tokens by executing `XGROW` with the `<writer>` corresponding to the token contract. They can destroy tokens any time by calling `XCLEAR`.

When a storage cell gets evicted, or removed by `XCLEAR`, token contract gets a notification call by `SYSTEM` account with a gas stipend enough to update token supply counter, and other summary information.

Critique 1 - active measures to retain tokens

This model requires active measures from users to retain their tokens, which creates ‘ooops I lost million dollars while in coma’ problem.

The problem appears to come from the proposed irrecoverability of linear storage (it becomes recoverable after step 4), and also from separating the fates of tokens for individual token holders.

This problem is reflecting the property of the real world, where every asset requires some sort of looking after. Ability to keep an asset forever, without doing anything for it, and just with storing limited (even though large) amount of data is unattainable unless we allow unbounded state growth, as conjectured in section “Problem statement - bounded state”.

Critique 2 - extra transaction churn

Extra transaction churn to maintain the status quo can come from two sources:

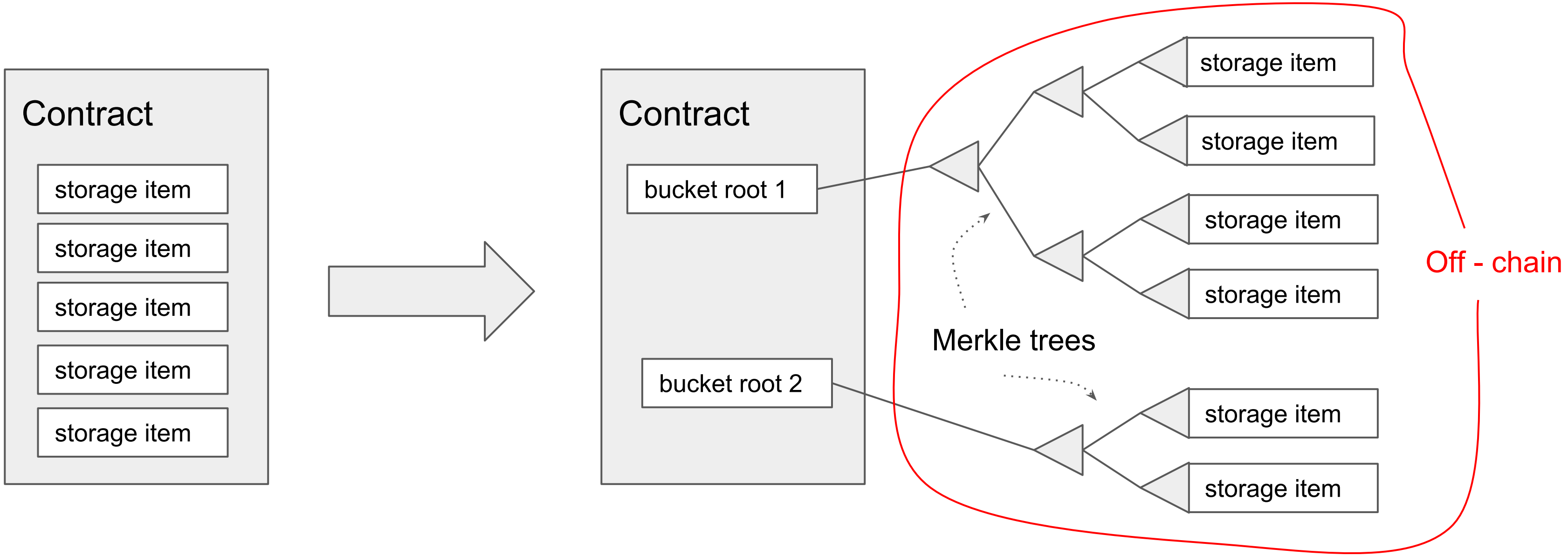
1. Users topping up their balanced to keep their storage alive
2. Eviction mechanism re-computing *rentbalance* and modifying *rentbalance*, *balance*, *rentblock*, even though actual eviction does not happen

For churn coming from the source (1), larger top ups could be solution. For churn coming from the source (2), hopefully priority queue design at Step 2 is a solution.

It has also been described as “rent payments are micro-transactions” in https://media.rsk.co/rsk-research-news-storage-rent/

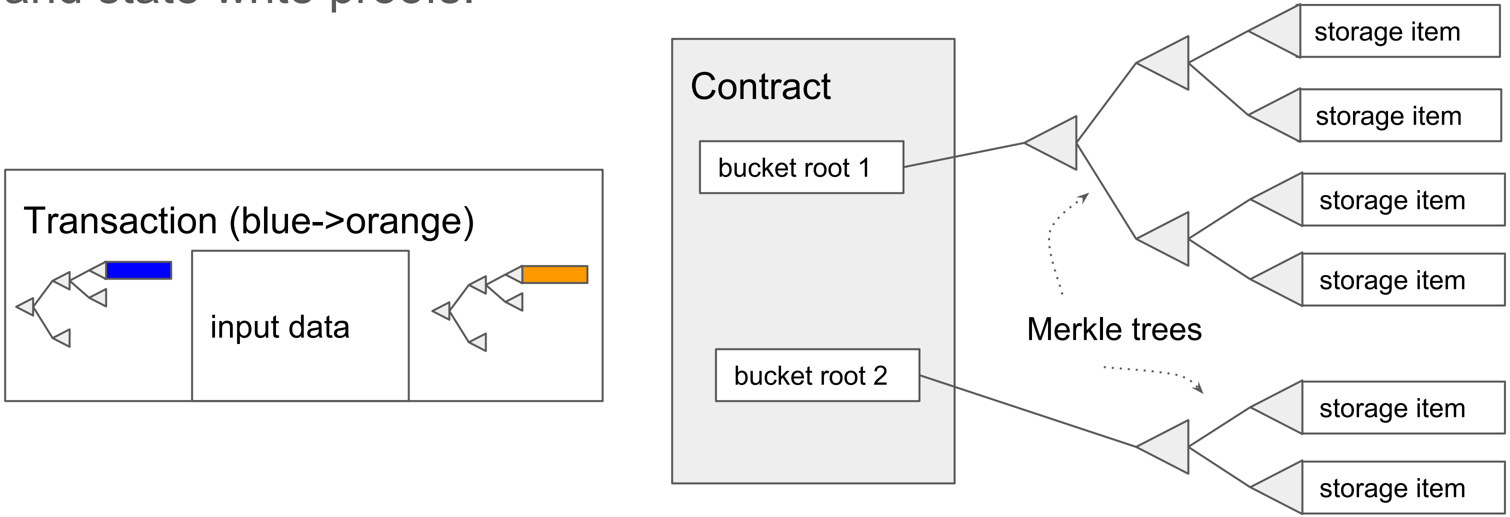
Alternative to Step 3 - Stateless contracts

If Step 3 is not implemented, but Step 4 is implemented directly after Step 2, contracts that cannot afford paying rent for all their users' assets, and do not have user-owned storage to use, might resort to the technique of stateless contracts.



Alternative to Step 3 - Stateless contracts

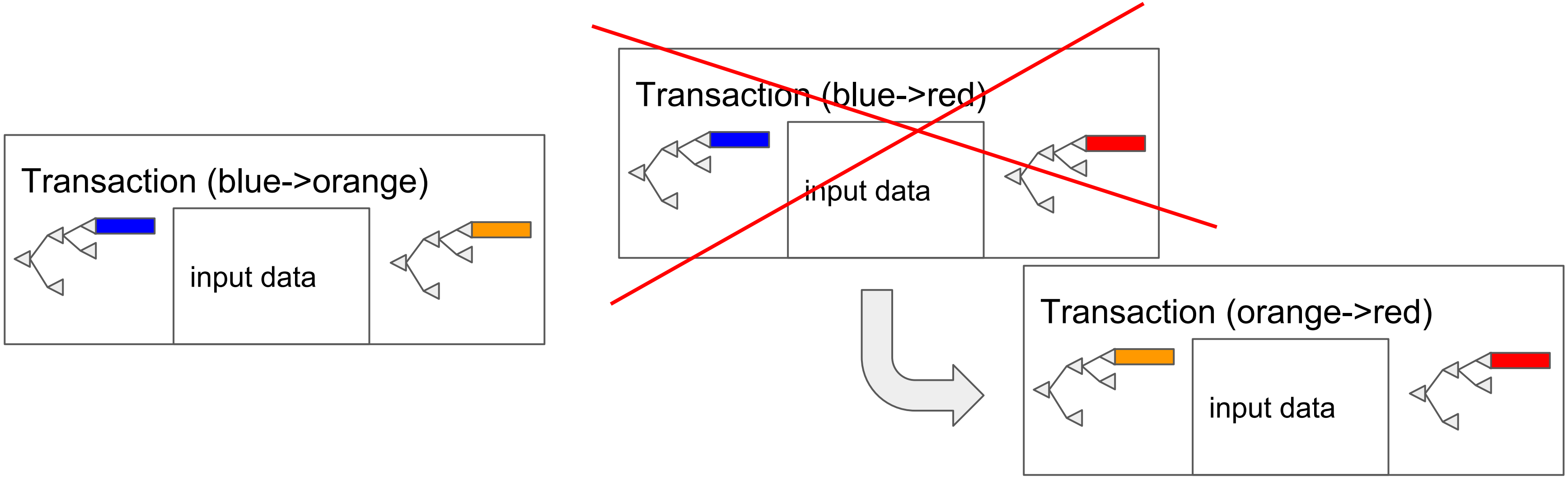
Most of the contract's data is moved off-chain, and only the bucket roots are stored on-chain. Each transaction that interacts with such contract's data will need to have 2 new components: state read proofs (together with state read values), and state write proofs.



Alternative to Step 3 - Stateless contracts

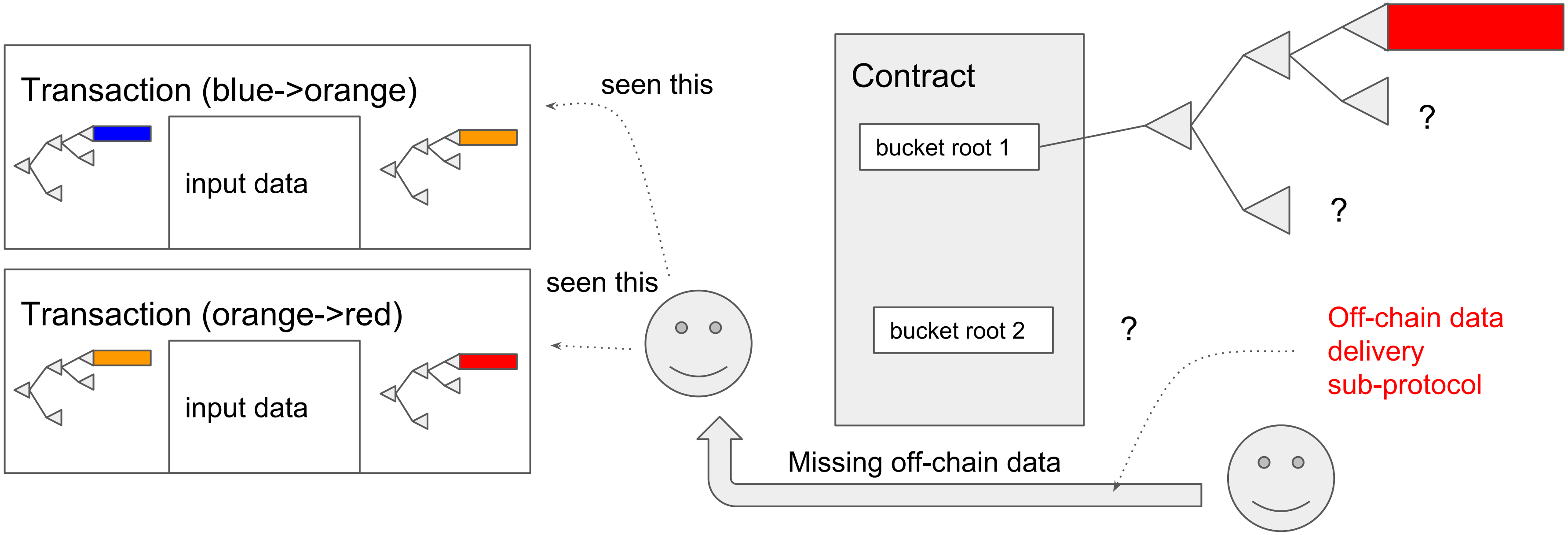
Transactions that are trying to modify items in the same bucket are now mutually exclusive, and the “losing” transactions will have to be recreated and re-submitted.

That is why contract’s state is split into buckets, to reduce contention



Alternative to Step 3 - Stateless contracts

A new user who has not watched the contract from the beginning, and knows the contract’s state only partially, will need to obtain a copy of the state by using some kind of sub-protocol



Step 4 - Top-up and recovery mechanism

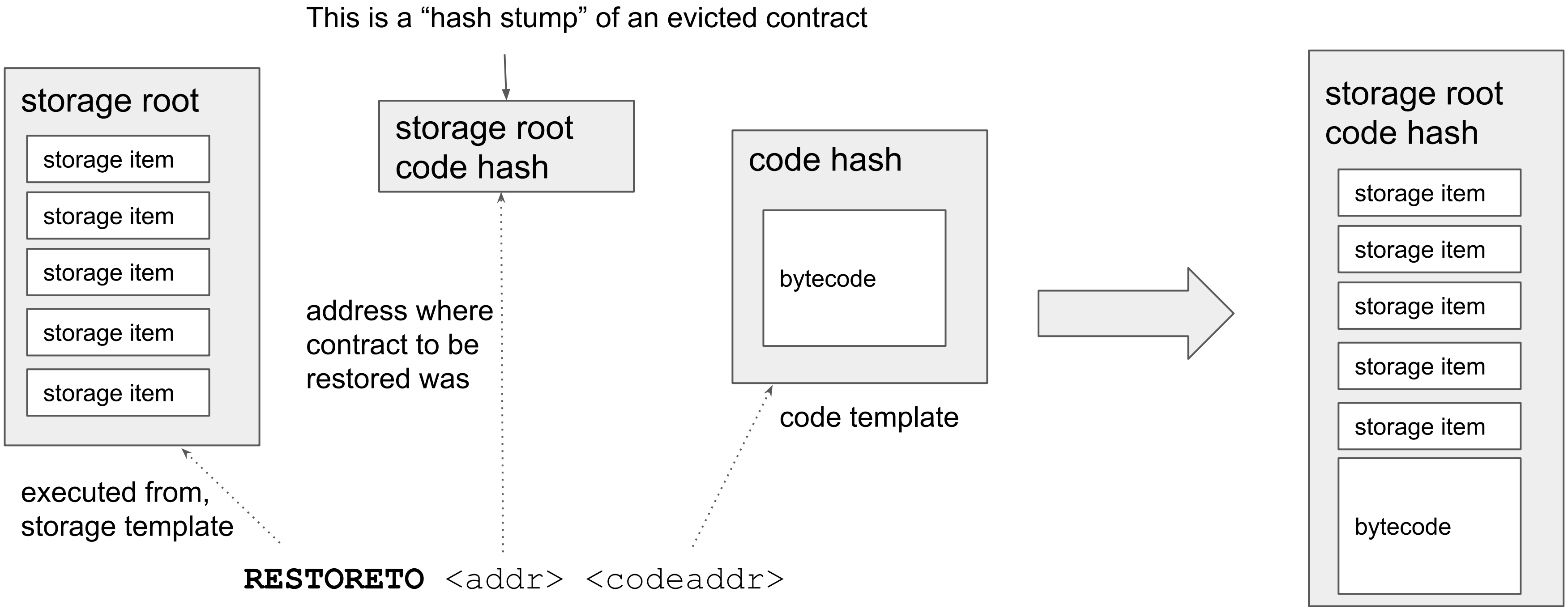
As described: **https://gist.github.com/fjl/b495aa2154944263811eb1a73c6498cd**

New opcode `PAYRENT` is introduced to top up *rentbalance* by spending ETH, and `RENTBALANCE` (to read *rentbalance*). This can help keep existing contracts alive until they are migrated.

When rent is not paid, contracts leave a “hash stump”, which can be used to restore the contract using opcode `RESTORETO`. This is different from semantics after Step 3, where linear cross-contract storage would be lost. At this step, linear cross-contract storage can also be recovered with `RESTORETO`.

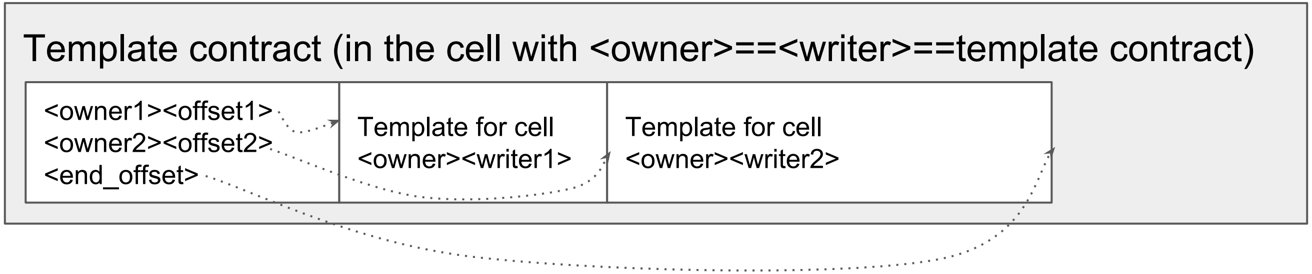
Step 4 - opcode `RESTORETO`

It solves the problem of contract restoration from a “hash stump” left in the state.



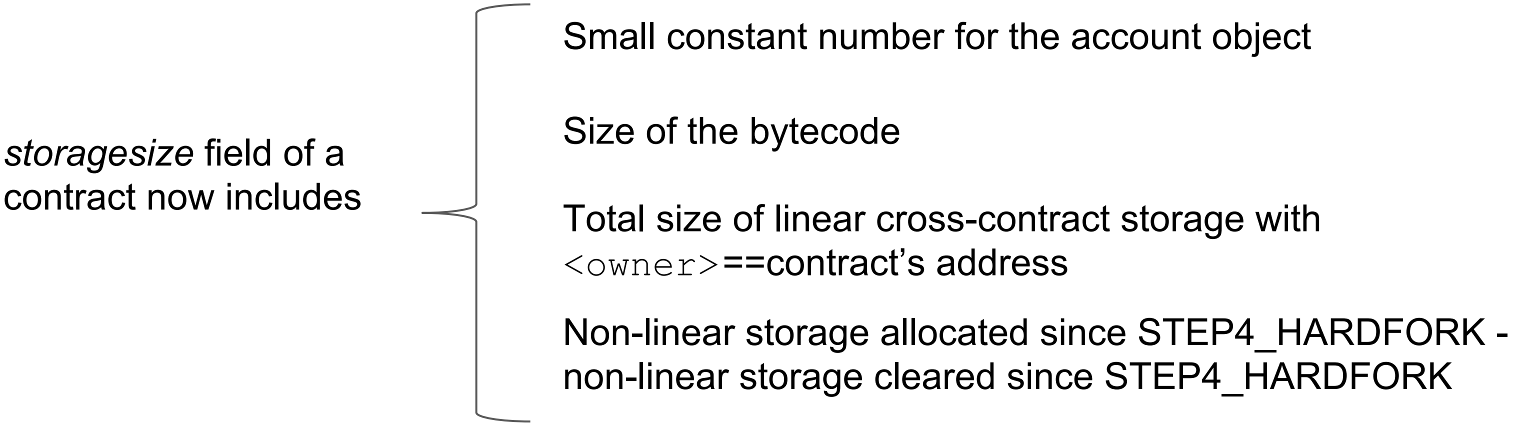
Step 4 - restoring linear cross-contract storage

Semantics of `RESTORETO` opcode needs to be modified for restoration of linear cross-contract storage, to designate cells for various writers. One idea is for the storage template to have everything in its own linear storage, with some descriptor in front:



Step 4 - storage size field

Field *storagesize* changes its composition to:



storagesize field of a contract now includes

Small constant number for the account object

Size of the bytecode

Total size of linear cross-contract storage with `<owner>`==contract's address

Non-linear storage allocated since STEP4_HARDFORK - non-linear storage cleared since STEP4_HARDFORK

Newly created (via `SSTORE`) non-linear storage (that exists now) now increases the *storagesize* field of a contract, and emptying storage items decreases the field. Refunds for `SSTORE` are abandoned.

Step 5 - floating rent

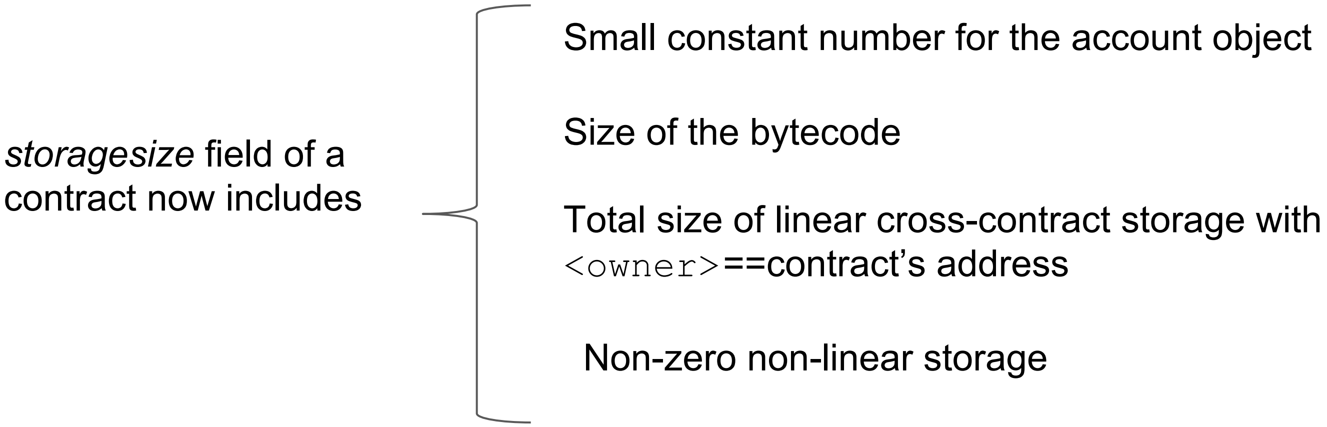
Subsequent protocol upgrade adds the sizes of non-linear storage that existed before Step 4 to the *storagesize*. This can be done efficiently because the calculation of the storage sizes at Step 4 block is done off-line and reconciled between all client implementations ahead of the upgrade.

Now all the storage is counted. This allows introduction of storage upper limit and floating rent price, depending on the “storage pressure”. Storage pressure is the measure of how closely the current size of all storage is to the upper limit.

Some more info on maintenance fees:
https://gist.github.com/zsfelfoldi/c40ff6637b9a6a095ddada87eb0d4891

Step 5 - storage size field

Field *storagesize* now includes all the storage.



storagesize field of a contract now includes

Small constant number for the account object

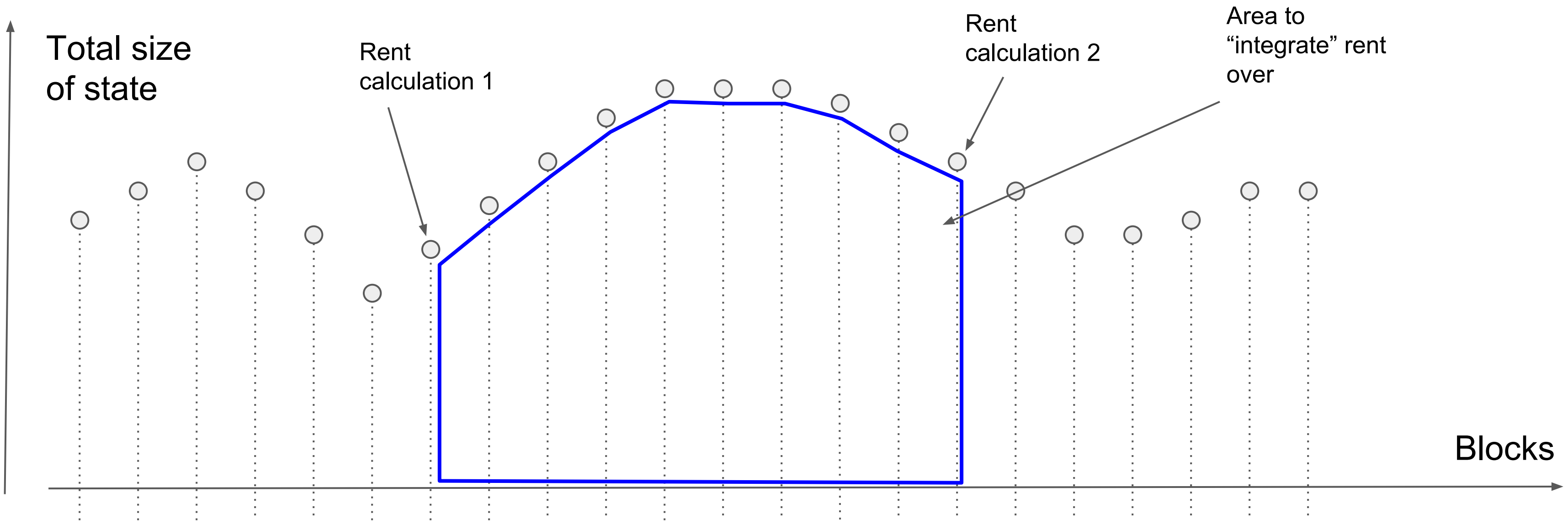
Size of the bytecode

Total size of linear cross-contract storage with `<owner>`==contract’s address

Non-zero non-linear storage

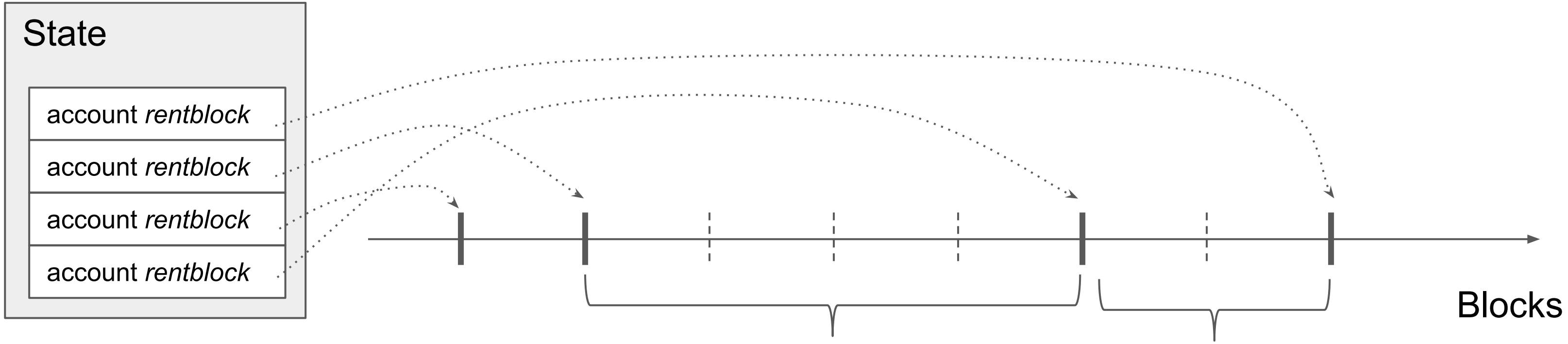
Step 5 - calculation of rent

Since rent is calculated at each modification of an account, the *storagesize* field does not change in between, so the only complexity is to aggregate the potentially changing rent over the calculation period.



Step 5 - calculation of rent

In order to integrate over the history of state size, the protocol needs to keep track of that history for `BLOCK` - min(*rentblock*) blocks, where “min” is taken over all the accounts that are presently in the state. Alternatively, it might be more space efficient to only keep track of intervals between various values of *rentblock*



only keep aggregated values for the intervals

Step 6 - removing/resurrecting hash stumps

2 options so far:

1) Vitalik’s suggested exclusion proofs, which imposes minimum live time on contracts, chapters 8-9 of https://github.com/ethereum/research/blob/master/papers/pricing/ethpricing.pdf
2) Graveyard tree - state includes a merkle root of the tree containing all removed contracts. To remove a contract, one needs to show the path in the graveyard tree where the contract’s hash will live (this requires knowledge of the history of all removals). To resurrect a contract, one needs to show the path to the contract in the graveyard tree. The root of the tree gets updated that the contract is not in the tree anymore (it is now alive).