Introduction

The purpose of this document is to provide a reasonably complete specification and introduction for anyone looking to understand the details of the sharding proposal, as well as to implement it. This document as written describes only "phase 1" of quadratic sharding; phases 2, 3 and 4 are at this point out of scope, and super-quadratic sharding ("Ethereum 3.0") is also out of scope.

Suppose that the variable c denotes the level of computational power available to one node. In a simple blockchain, the transaction capacity is bounded by O(c), as every node must process every transaction. The goal of quadratic sharding is to increase the capacity with a two-layer design. Stage 1 requires no hard forks; the main chain stays exactly as is. However, a contract is published to the main chain called the **validator manager contract** (VMC), which maintains the sharding system. There are O(c) **shards** (currently, 100), where each shard is like a separate "galaxy": it has its own account space, transactions need to specify which shard they are to be published inside, and communication between shards is very limited (in fact, in phase 1, it is nonexistent).

The shards are run on a simple longest-chain-rule proof of stake system, where the stake is on the main chain (specifically, inside the VMC). All shards share a common validator pool; this also means that anyone who signs up with the VMC as a validator could theoretically at any time be assigned the right to create a block on any shard. Each shard has a block size/gas limit of O(c), and so the total capacity of the system is O(c^2).

Most users of the sharding system will run both (i) either a full (O(c) resource requirements) or light (O(log(c)) resource requirements) node on the main chain, and (ii) a "shard client" which talks to the main chain node via RPC (this client is assumed to be trusted because it's also running on the user's computer) and which can also be used as a light client for any shard, as a full client for any specific shard (the user would have to specify that they are "watching" a specific shard) or as a validator node. In all cases, the storage and computation requirements for a shard client will also not exceed O(c) (unless the user chooses to specify that they are watching *every* shard; block explorers and large exchanges may want to do this).

In this document, the term Collation is used to differentiate from Block because (i) they are different RLP objects: transactions are level 0 objects, collations are level 1 objects that package transactions, and blocks are level 2 objects that package collation (headers); (ii) it's

clearer in context of sharding. Basically, Collation must consist of CollationHeader and TransactionList; Witness and the detailed format of Collation will be defined in **Stateless clients** section. Collator is the collation proposer sampled by getEligibleProposer function of **Validator Manager Contract** in the main chain; the mechanism will be introduced in the following sections.

Main Chain	Shard Chain
Block	Collation
BlockHeader	CollationHeader
Block Proposer (or Miner in PoW chain)	Collator

Quadratic sharding

Constants

- LOOKAHEAD_PERIODS: 4
- PERIOD_LENGTH: 5
- COLLATION_GASLIMIT: 10,000,000 gas
- SHARD_COUNT: 100
- SIG_GASLIMIT: 40000 gas
- COLLATOR_REWARD: 0.001 ETH

Validator Manager Contract (VMC)

We assume that at address VALIDATOR_MANAGER_ADDRESS (on the existing "main shard") there exists the VMC, which supports the following functions:

- deposit() returns uint256: adds a validator to the validator set, with the validator's size being the msg.value (i.e., the amount of ETH deposited) in the function call. This function returns the validator index.
- withdraw(uint256 validator_index) returns bool: verifies that msg.sender == validators[validator_index].addr. if it is removes the validator from the validator set and refunds the deposited ETH.
- get_eligible_proposer(uint256 shard_id, uint256 period) returns address: uses a block hash as a seed to pseudorandomly select a signer from the validator set. The chance of being selected should be proportional to the validator's deposit. The function should be

able to return a value for the current period or any future up to LOOKAHEAD_PERIODS periods ahead.
- add_header(uint256 shard_id, uint256 expected_period_number, bytes32 period_start_prevhash, bytes32 parent_hash, bytes32 transaction_root, address coinbase, bytes32 state_root, bytes32 receipt_root, uint256 number) returns bool: attempts to process a collation header, returns True on success, reverts on failure.
- get_shard_head(uint256 shard_id) returns bytes32: returns the header hash that is the head of a given shard as perceived by the manager contract.

There is also one log type:

- CollationAdded(indexed uint256 shard_id, bytes collation_header_bytes, bool is_new_head, uint256 score)

where collation_header_bytes can be constructed in vyper by

```
    collation_header_bytes = concat(
        as_bytes32(shard_id),
        as_bytes32(expected_period_number),
        period_start_prevhash,
        parent_hash,
        transaction_root,
        as_bytes32(collation_coinbase),
        state_root,
        receipt_root,
        as_bytes32(collation_number),
    )
```

Note: coinbase and number are renamed to collation_coinbase and collation_number, due to the fact that they are reserved keywords in vyper.

Collation header

We first define a "collation header" as an RLP list with the following values:

```
[
    shard_id: uint256,
    expected_period_number: uint256,
    period_start_prevhash: bytes32,
    parent_hash: bytes32,
```

```
    transaction_root: bytes32,
    coinbase: address,
    state_root: bytes32,
    receipt_root: bytes32,
    number: uint256,
]
```

Where:

- shard_id is the shard ID of the shard;
- expected_period_number is the period number in which this collation expects to be included; this is calculated as period_number = floor(block.number / PERIOD_LENGTH);
- period_start_prevhash is the block hash of block PERIOD_LENGTH * expected_period_number - 1 (i.e., it is the hash of the last block before the expected period starts). Opcodes in the shard that refer to block data (e.g. NUMBER and DIFFICULTY) will refer to the data of this block, with the exception of COINBASE, which will refer to the shard coinbase;
- parent_hash is the hash of the parent collation;
- transaction_root is the root hash of the trie holding the transactions included in this collation;
- state_root is the new state root of the shard after this collation;
- receipt_root is the root hash of the receipt trie;
- number is the collation number, which is also the score for the fork choice rule now; and

A **collation header** is valid if calling add_header(shard_id, expected_period_number, period_start_prevhash, parent_hash, transaction_root, coinbase, state_root, receipt_root, number) returns true. The validator manager contract should do this if:

- the shard_id is at least 0, and less than SHARD_COUNT;
- the expected_period_number equals the actual current period number (i.e., floor(block.number / PERIOD_LENGTH))
- a collation with the hash parent_hash for the same shard has already been accepted;
- a collation for the same shard has not yet been submitted during the current period;
- the address of the sender of add_header is equal to the address returned by get_eligible_proposer(shard_id, expected_period_number).

A **collation** is valid if: (i) its collation header is valid; (ii) executing the collation on top of the parent_hash's state_root results in the given

state_root and receipt_root; and (iii) the total gas used is less than or equal to COLLATION_GASLIMIT.

Collation state transition function

The state transition process for executing a collation is as follows:

- execute each transaction in the tree pointed to by transaction_root in order; and
- assign a reward of COLLATOR_REWARD to the coinbase.

Details of getEligibleProposer

Here is one simple implementation in Viper:

```
def getEligibleProposer(shardId: num, period: num) -> address:
    assert period >= LOOKAHEAD_PERIODS
    assert (period - LOOKAHEAD_PERIODS) * PERIOD_LENGTH < block.number
    assert self.num_validators > 0

    h = as_num256(
        sha3(
            concat(
                blockhash((period - LOOKAHEAD_PERIODS) *
PERIOD_LENGTH),
                as_bytes32(shardId)
            )
        )
    )
    return self.validators[
        as_num128(
            num256_mod(
                h,
                as_num256(self.num_validators)
            )
        )
    ].addr
```

Stateless clients

A validator is only given a few minutes' notice (precisely, LOOKAHEAD_PERIODS * PERIOD_LENGTH blocks worth of notice) when they are

asked to create a block on a given shard. In Ethereum 1.0, creating a block requires having access to the entire state in order to validate transactions. Here, our goal is to avoid requiring validators to store the state of the entire system (as that would be an O(c^2) computational resource requirement). Instead, we allow validators to create collations knowing only the state root, pushing the responsibility onto transaction senders to provide "witness data" (i.e., Merkle branches), to prove the pre-state of the accounts that the transaction affects, and to provide enough information to calculate the post-state root after executing the transaction.

(Note that it's theoretically possible to implement sharding in a non-stateless paradigm; however, this requires: (i) storage rent to keep storage size bounded; and (ii) validators to be assigned to create blocks in a single shard for O(c) time. This scheme avoids the need for these sacrifices.)

Data format

We modify the format of a transaction so that the transaction must specify an **access list** enumerating the parts of the state that it can access (we describe this more precisely later; for now consider this informally as a list of addresses). Any attempt to read or write to any state outside of a transaction's specified access list during VM execution returns an error. This prevents attacks where someone sends a transaction that spends 5 million cycles of gas on random execution, then attempts to access a random account for which the transaction sender and the collator do not have a witness, preventing the collator from including the transaction and thereby wasting the collator's time.

Outside of the signed body of the transaction, but packaged along with the transaction, the transaction sender must specify a "witness", an RLP-encoded list of Merkle tree nodes that provides the portions of the state that the transaction specifies in its access list. This allows the collator to process the transaction with only the state root. When publishing the collation, the collator also sends a witness for the entire collation.

Transaction package format

```
    [
        [nonce, acct, data....],    # transaction body (see below for specification)
        [node1, node2, node3....]   # witness
```

```
    ]
```

Collation format

```
    [
        [shard_id, ... , sig],    # header
        [tx1, tx2 ...],           # transaction list
        [node1, node2, node3...] # witness
    ]
```

See also ethresearch thread on The Stateless Client Concept.

Stateless client state transition function

In general, we can describe a traditional "stateful" client as executing a state transition function stf(state, tx) -> state' (or stf(state, block) -> state'). In a stateless client model, nodes do not store the state. The functions apply_transaction and apply_block can be rewritten as follows:

apply_block(state_obj, witness, block) -> state_obj', reads, writes

Where state_obj is a tuple containing the state root and other O(1)-sized state data (gas used, receipts, bloom filter, etc); witness is a witness; and block is the rest of the block. The returned output is:

- a new state_obj containing the new state root and other variables;
- the set of objects from the witness that have been read (which is useful for block creation); and
- the set of new state objects that have been created to form the new state trie.

This allows the functions to be "pure", as well as only dealing with small-sized objects (as opposed to the state in existing Ethereum, which is currently hundreds of gigabytes), making them convenient to use for sharding.

Client logic

A client would have a config of the following form:

```
{
    validator_address: "0x..." OR null,
```

```
    watching: [list of shard IDs],
    ...
}
```

If a validator address is provided, then it checks (on the main chain) if the address is an active validator. If it is, then every time a new period on the main chain starts (i.e., when floor(block.number / PERIOD_LENGTH) changes), then it should call getEligibleProposer for all shards for period floor(block.number / PERIOD_LENGTH) + LOOKAHEAD_PERIODS. If it returns the validator's address for some shard i, then it runs the algorithm CREATE_COLLATION(i) (see below).

For every shard i in the watching list, every time a new collation header appears in the main chain, it downloads the full collation from the shard network, and verifies it. It locally keeps track of all valid headers (where validity is defined recursively, i.e., for a header to be valid its parent must also be valid), and accepts as the main shard chain the shard chain whose head has the highest score, and where all collations from the genesis collation to the head are valid and available. Note that this implies the reorgs of the main chain *and* reorgs of the shard chain may both influence the shard head.

Fetch candidate heads in reverse sorted order

To implement the algorithms for watching a shard, and for creating a collation, the first primitive that we need is the following algorithm for fetching candidate heads in highest-to-lowest order. First, suppose the existence of an (impure, stateful) method getNextLog(), which gets the most recent CollationAdded log in some given shard that has not yet been fetched. This would work by fetching all the logs in recent blocks backwards, starting from the head, and within each block looking in reverse order through the receipts. We define an impure method fetch_candidate_head as follows:

```
unchecked_logs = []
current_checking_score = None

def fetch_candidate_head():
    # Try to return a log that has the score that we are checking for,
    # checking in order of oldest to most recent.
    for i in range(len(unchecked_logs)-1, -1, -1):
        if unchecked_logs[i].score == current_checking_score:
            return unchecked_logs.pop(i)
    # If no further recorded but unchecked logs exist, go to the next
```

```
    # isNewHead = true log
    while 1:
        unchecked_logs.append(getNextLog())
        if unchecked_logs[-1].isNewHead is True:
            break
    o = unchecked_logs.pop()
    current_checking_score = o.score
    return o
```

To re-express in plain language, the idea is to scan backwards through CollationAdded logs (for the correct shard), and wait until you get to one where isNewHead = True. Return that log first, then return all more recent logs with a score equal to that log with isNewHead = False, in order of oldest to most recent. Then go to the previous log with isNewHead = True (this is guaranteed to have a score that is 1 lower than the previous NewHead), then go to all more recent collations after it with that score, and so forth.

The idea is that this algorithm is guaranteed to check potential head candidates in highest-to-lowest sorted order of score, with the second priority being oldest to most recent.

For example, suppose that CollationAdded logs have hashes and scores as follows:

```
... 10 11 12 11 13   14 15 11 12 13   14 12 13 14 15   16 17 18 19 16
```

Then, isNewHead would be assigned as:

```
... T  T  T  F  T    T  T  F  F  F    F  F  F  F  F    T  T  T  T  F
```

If we number the collations A1..A5, B1..B5, C1..C5 and D1..D5, the precise returning order is:

```
D4 D3 D2 D1 D5 B2 C5 B1 C1 C4 A5 B5 C3 A3 B4 C2 A2 A4 B3 A1
```

Watching a shard

If a client is watching a shard, it should attempt to download and verify any collations in that shard that it can (checking any given collation only if its parent has already been verified). To get the head at any time, keep calling fetch_candidate_head() until it returns a collation that has been verified; that collation is the head. This will in normal circumstances return a valid collation immediately or at most after a few

tries due to latency or a small-scale attack that creates a few invalid or unavailable collations. Only in the case of a true long-running 51% attack will this algorithm degrade to O(N) time.

CREATE_COLLATION

This process has three parts. The first part can be called GUESS_HEAD(shard_id), with pseudocode here:

```
# Download a single collation and check if it is valid or invalid (memoized)
validity_cache = {}
def memoized_fetch_and_verify_collation(c):
    if c.hash not in validity_cache:
        validity_cache[c.hash] = fetch_and_verify_collation(c)
    return validity_cache[c.hash]


def main(shard_id):
    head = None
    while 1:
        head = fetch_candidate_head(shard_id)
        c = head
        while 1:
            if not memoized_fetch_and_verify_collation(c):
                break
            c = get_parent(c)
```

fetch_and_verify_collation(c) involves fetching the full data of c (including witnesses) from the shard network, and verifying it. The above algorithm is equivalent to "pick the longest valid chain, check validity as far as possible, and if you find it's invalid then switch to the next-highest-scoring valid chain you know about". The algorithm should only stop when the validator runs out of time and it is time to create the collation. Every execution of fetch_and_verify_collation should also return a "write set" (see stateless client section above). Save all of these write sets, and combine them together; this is the recent_trie_nodes_db.

We can now define UPDATE_WITNESS(tx, recent_trie_nodes_db). While running GUESS_HEAD, a node will have received some transactions. When it comes time to (attempt to) include a transaction into a collation, this algorithm will need to be run on the transaction first. Suppose that the transaction has an access list [A1 ... An], and a witness W. For each Ai,

use the current state tree root and get the Merkle branch for Ai, using the union of recent_trie_nodes_db and W as a database. If the original W was correct, and the transaction was sent not before the time that the client checked back to, then getting this Merkle branch will always succeed. After including the transaction into a collation, the "write set" from the state change should then also be added into the recent_trie_nodes_db.

Next, we have CREATE_COLLATION. For illustration, here is full pseudocode for a possible transaction-gathering part of this method.

```
# Sort by descending order of gasprice
txpool = sorted(copy(available_transactions), key=-tx.gasprice)
collation = new Collation(...)
while len(txpool) > 0:
    # Remove txs that ask for too much gas
    i = 0
    while i < len(txpool):
        if txpool[i].startgas > GASLIMIT - collation.gasused:
            txpool.pop(i)
        else:
            i += 1
    tx = copy.deepcopy(txpool[0])
    tx.witness = UPDATE_WITNESS(tx.witness, recent_trie_nodes_db)
    # Try to add the transaction, discard if it fails
    success, reads, writes = add_transaction(collation, tx)
    recent_trie_nodes_db = union(recent_trie_nodes_db, writes)
    txpool.pop(0)
```

At the end, there is an additional step, finalizing the collation (to give the collator the reward, which is COLLATOR_REWARD ETH). This requires asking the network for a Merkle branch for the collator's account. When the network replies with this, the post-state root after applying the reward, as well as the fees, can be calculated. The collator can then package up the collation, of the form (header, txs, witness), where the witness is the union of the witnesses of all the transactions and the branch for the collator's account.

Protocol changes

Transaction format

The format of a transaction now becomes (note that this includes account abstraction and read/write lists):

```
[
    chain_id,       # 1 on mainnet
    shard_id,       # the shard the transaction goes onto
    target,         # account the tx goes to
    data,           # transaction data
    start_gas,      # starting gas
    gasprice,       # gasprice
    access_list,    # access list (see below for specification)
    code            # initcode of the target (for account creation)
]
```

The process for applying a transaction is now as follows:

- Verify that the chain_id and shard_id are correct
- Subtract start_gas * gasprice wei from the target account
- Check if the target account has code. If not, verify that sha3(code)[12:] == target
- If the target account is empty, execute a contract creation at the target with code as init code; otherwise skip this step
- Execute a message with the remaining gas as startgas, the target as the to address, 0xff...ff as the sender, 0 value, and the transaction data as data
- If either of the two executions fail, and <= 200000 gas has been consumed (i.e., start_gas - remaining_gas <= 200000), the transaction is invalid
- Otherwise remaining_gas * gasprice is refunded, and the fee paid is added to a fee counter (note: fees are NOT immediately added to the coinbase balance; instead, fees are added all at once during block finalization)

Two-layer trie redesign

The existing account model is replaced with one where there is a single-layer trie, and all account balances, code and storage are incorporated into the trie. Specifically, the mapping is:

- Balance of account X: sha3(X) ++ 0x00
- Code of account X: sha3(X) ++ 0x01
- Storage key K of account X: sha3(X) ++ 0x02 ++ K

See also ethresearch thread on A two-layer account trie inside a single-layer trie

Additionally, the trie is now a new binary trie design: https://github.com/ethereum/research/tree/master/trie_research

Access list

The access list for an account looks as follows:

[[address, prefix1, prefix2...], [address, prefix1, prefix2...], ...]

This basically means "the transaction can access the balance and code for the given accounts, as well as any storage key provided that at least one of the prefixes listed with the account is a prefix of the storage key". One can translate it into "prefix list form", which essentially is a list of prefixes of the underlying storage trie (see above section):

```
def to_prefix_list_form(access_list):
    o = []
    for obj in access_list:
        addr, storage_prefixes = obj[0], obj[1:]
        o.append(sha3(addr) + b'\x00')
        o.append(sha3(addr) + b'\x01')
        for prefix in storage_prefixes:
            o.append(sha3(addr) + b'\x02' + prefix)
    return o
```

One can compute the witness for a transaction by taking the transaction's access list, converting it into prefix list form, then running the algorithm get_witness_for_prefix for each item in the prefix list form, and taking the union of these results.

get_witness_for_prefix returns a minimal set of trie nodes that are sufficient to access any key which starts with the given prefix. See implementation here: https://github.com/ethereum/research/blob/b0de8d352f6236c9fa2244fed871546fabb016d1/trie_research/new_bintrie.py#L250

In the EVM, any attempt to access (either by calling or SLOAD'ing or via an opcode such as BALANCE or EXTCODECOPY) an account that is outside the access list will lead to the EVM instance that made the access attempt immediately throwing an exception.

See also ethresearch thread on Account read/write lists.

Gas costs

To be finalized.

Subsequent phases

This allows for a quick and dirty form of medium-security proof of stake sharding in a way that achieves quadratic scaling through separation of concerns between block proposers and collators, and thereby increases throughput by ~100x without too many changes to the protocol or software architecture. This is intended to serve as the first phase in a multi-phase plan to fully roll out quadratic sharding, the latter phases of which are described below.

- **Phase 2 (two-way pegging)**: see section on USED_RECEIPT_STORE, still to be written
- **Phase 3, option a**: require collation headers to be added in as uncles instead of as transactions
- **Phase 3, option b**: require collation headers to be added in an array, where item i in the array must be either a collation header of shard i or the empty string, and where the extra data must be the hash of this array (soft fork)
- **Phase 4 (tight coupling)**: blocks are no longer valid if they point to invalid or unavailable collations. Add data availability proofs.