---

title: "All you can cache" description: Learn how to create and use a caching contract for cheaper rollup transactions author: Ori Pomerantz tags: ["layer 2", "caching", "storage"] skill: intermediate published: 2022-09-15 lang: en

---

When using rollups the cost of a byte in the transaction is a lot more expensive than the cost of a storage slot. Therefore, it makes sense to cache as much information as possible on chain.

In this article you learn how to create and use a caching contract in such a way that any parameter value that is likely to be used multiple times will be cached and available for use (after the first time) with a much smaller number of bytes, and how to write off chain code that uses this cache.

If you want to skip the article and just see the source code,it is here. The development stack is Foundry.

Overall design {#overall-design}

For the sake of simplicity we'll assume all transaction parameters are`uint256`, 32 bytes long. When we receive a transaction, we'll parse each parameter like this:

1. If the first byte is `0xFF`, take the next 32 bytes as a parameter value and write it to the cache.
2. If the first byte is `0xFE`, take the next 32 bytes as a parameter value but do*not* write it to the cache.
3. For any other value, take the top four bits as the number of additional bytes, and the bottom four bits as the most significant bits of the cache key. Here are some examples:

| Bytes in calldata | Cache key | | :---------------- | --------: | | 0x0F | 0x0F | | 0x10,0x10 | 0x10 | | 0x12,0xAC | 0x02AC | | 0x2D,0xEA, 0xD6 | 0x0DEAD6 |

Cache manipulation {#cache-manipulation}

The cache is implemented in `Cache.sol`. Let's go over it line by line.

```solidity // SPDX-License-Identifier: UNLICENSED pragma solidity ^0.8.13;

contract Cache {

bytes1 public constant INTO_CACHE = 0xFF;
bytes1 public constant DONT_CACHE = 0xFE;

```

These constants are used to interpret the special cases where we provide all the information and either want it written into the cache or not. Writing into the cache requires two `SSTORE` operations into previously unused storage slots at a cost of 22100 gas each, so we make it optional.

```solidity

mapping(uint => uint) public val2key;

```

A mapping between the values and their keys. This information is necessary to encode values before you send out the transaction.

```
solidity // Location n has the value for key n+1, because we need to preserve // zero as "not in the cache". uint[]
public key2val;
```

We can use an array for the mapping from keys to values because we assign the keys, and for simplicity we do it sequentially.

```
solidity function cacheRead(uint _key) public view returns (uint) { require(_key <= key2val.length, "Reading
uninitialize cache entry"); return key2val[_key-1]; } // cacheRead
```

Read a value from the cache.

```
solidity // Write a value to the cache if it's not there already // Only public to enable the test to work function
cacheWrite(uint _value) public returns (uint) { // If the value is already in the cache, return the current key if
```

```
(val2key[_value] != 0) { return val2key[_value]; }
```

There is no point in putting the same value in the cache more than once. If the value is already there, just return the existing key.

```
solidity // Since 0xFE is a special case, the largest key the cache can // hold is 0x0D followed by 15 0xFF's. If the cache length is already that // large, fail. // 1 2 3 4 5 6 7 8 9 A B C D E F require(key2val.length+1 < 0x0DFFFFFFFFFFFFFFFFFFFFFFFFFFFF, "cache overflow");
```

I don't think we'll ever get a cache that big (approximately $1.8*10^{37}$ entries, which would require about 10^{27} TB to store). However, I'm old enough to remember "640kB would always be enough". This test is very cheap.

```
solidity // Write the value using the next key val2key[_value] = key2val.length+1;
```

Add the reverse lookup (from the value to the key).

```
solidity key2val.push(_value);
```

Add the forward lookup (from the key to the value). Because we assign values sequentially we can just add it after the last array value.

```
solidity return key2val.length; } // cacheWrite
```

Return the new length of `key2val`, which is the cell where the new value is stored.

```
solidity function _calldataVal(uint startByte, uint length) private pure returns (uint)
```

This function reads a value from the calldata of arbitrary length (up to 32 bytes, the word size).

```solidity { uint _retVal;

    require(length < 0x21,
        "_calldataVal length limit is 32 bytes");
    require(length + startByte <= msg.data.length,
        "_calldataVal trying to read beyond calldatasize");

```

This function is internal, so if the rest of the code is written correctly these tests are not required. However, they don't cost much so we might as well have them.

```
solidity assembly { _retVal := calldataload(startByte) }
```

This code is in Yul. It reads a 32 byte value from the calldata. This works even if the calldata stops before `startByte+32` because uninitialized space in EVM is considered to be zero.

```
solidity _retVal = _retVal >> (256-length*8);
```

We don't necessarily want a 32 byte value. This gets rid of the excess bytes.

```solidity return _retVal; } // _calldataVal

// Read a single parameter from the calldata, starting at _fromByte
function _readParam(uint _fromByte) internal
    returns (uint _nextByte, uint _parameterValue)
{

```

Read a single parameter from the calldata. Note that we need to return not just the value we read, but also the location of the next byte because parameters can range from 1 byte long to 33 bytes.

```solidity // The first byte tells us how to interpret the rest uint8 _firstByte;

    _firstByte = uint8(_calldataVal(_fromByte, 1));

```

Solidity tries to reduce the number of bugs by forbidding potentially dangerous implicit type conversions. A downgrade, for example from 256 bits to 8 bits, needs to be explicit.

```solidity
```

```
    // Read the value, but do not write it to the cache
    if (_firstByte == uint8(DONT_CACHE))
        return(_fromByte+33, _calldataVal(_fromByte+1, 32));

    // Read the value, and write it to the cache
    if (_firstByte == uint8(INTO_CACHE)) {
        uint _param = _calldataVal(_fromByte+1, 32);
        cacheWrite(_param);
        return(_fromByte+33, _param);
    }

    // If we got here it means that we need to read from the cache

    // Number of extra bytes to read
    uint8 _extraBytes = _firstByte / 16;

```

Take the lower nibble and combine it with the other bytes to read the value from the cache.

```solidity uint _key = (uint256(_firstByte & 0x0F) << (8*_extraBytes)) + _calldataVal(_fromByte+1, _extraBytes);

    return (_fromByte+_extraBytes+1, cacheRead(_key));

}  // _readParam


// Read n parameters (functions know how many parameters they expect)
function _readParams(uint _paramNum) internal returns (uint[] memory) {

```

We could get the number of parameters we have from the calldata itself, but the functions that call us know how many parameters they expect. It's easier to let them tell us.

```solidity // The parameters we read uint[] memory params = new uint;

    // Parameters start at byte 4, before that it's the function signature
    uint _atByte = 4;

    for(uint i=0; i<_paramNum; i++) {
        (_atByte, params[i]) = _readParam(_atByte);
    }

```

Read the parameters until you have the number you need. If we go past the end of the calldata, `_readParams` will revert the call.

```solidity

    return(params);
}   // readParams

// For testing _readParams, test reading four parameters
function fourParam() public
    returns (uint256,uint256,uint256,uint256)
{
    uint[] memory params;
    params = _readParams(4);
    return (params[0], params[1], params[2], params[3]);
}    // fourParam

```

One big advantage of Foundry is that it allows tests to be written in Solidity (see Testing the cache below). This makes unit tests a lot easier. This is a function that reads four parameters and returns them so the test can verify they were correct.

`solidity // Get a value, return bytes that will encode it (using the cache if possible) function encodeVal(uint _val) public view returns(bytes memory) {`

`encodeVal` is a function that off-chain code calls to help create calldata that uses the cache. It receives a single value and returns the bytes that encode it. This function is a `view`, so it does not require a transaction and when called externally does not cost any gas.

```solidity uint _key = val2key[_val];
```

```
    // The value isn't in the cache yet, add it
    if (_key == 0)
        return bytes.concat(INTO_CACHE, bytes32(_val));
```

In the EVM all uninitialized storage is assumed to be zeros. So if we look for the key for a value that isn't there, we get a zero. In that case the bytes that encode it are `INTO_CACHE` (so it will be cached next time), followed by the actual value.

```
solidity // If the key is <0x10, return it as a single byte if (_key < 0x10) return
bytes.concat(bytes1(uint8(_key)));
```

Single bytes are the easiest. We just use `bytes.concat` to turn a `bytes<n>` type into a byte array which can be any length. Despite the name, it works fine when provided with just one argument.

```
solidity // Two byte value, encoded as 0x1vvv if (_key < 0x1000) return bytes.concat(bytes2(uint16(_key) | 0x1000));
```

When we have a key that is less than 16^3, we can express it in two bytes. We first convert `_key`, which is a 256 bit value, to a 16 bit value and use logical or to add the number of extra bytes to the first byte. Then we just it into a `bytes2` value, which can be converted to `bytes`.

```solidity // There is probably a clever way to do the following lines as a loop, // but it's a view function so I'm optimizing for programmer time and // simplicity.

    if (_key < 16*256**2)
        return bytes.concat(bytes3(uint24(_key) | (0x2 * 16 * 256**2)));
    if (_key < 16*256**3)
        return bytes.concat(bytes4(uint32(_key) | (0x3 * 16 * 256**3)));
          .
          .
          .
    if (_key < 16*256**14)
        return bytes.concat(bytes15(uint120(_key) | (0xE * 16 * 256**14)));
    if (_key < 16*256**15)
        return bytes.concat(bytes16(uint128(_key) | (0xF * 16 * 256**15)));
```

The other values (3 bytes, 4 bytes, etc.) are handled the same way, just with different field sizes.

```
solidity // If we get here, something is wrong. revert("Error in encodeVal, should not happen");
```

If we get here it means we got a key that's not less than $16*256^{15}$. But `cacheWrite` limits the keys so we can't even get up to $14*256^{16}$ (which would have a first byte of 0xFE, so it would look like `DONT_CACHE`). But it doesn't cost us much to add a test in case a future programmer introduces a bug.

```solidity } // encodeVal

} // Cache ```

## Testing the cache {#testing-the-cache}

One of the advantages of Foundry is that it lets you write tests in Solidity, which makes it easier to write unit tests. The tests for the `Cache` class are here. Because the testing code is repetitive, as tests tend to be, this article only explains the interesting parts.

```solidity // SPDX-License-Identifier: UNLICENSED pragma solidity ^0.8.13;

import "forge-std/Test.sol";

// Need to run `forge test -vv` for the console. import "forge-std/console.sol"; ```

This is just boilerplate that is necessary to use the test package and `console.log`.

```
solidity import "src/Cache.sol";
```

We need to know the contract we are testing.

```solidity contract CacheTest is Test { Cache cache;

function setUp() public {
    cache = new Cache();
```

```
}
```

The `setUp` function is called before each test. In this case we just create a new cache, so that our tests won't affect each other.

```
solidity function testCaching() public {
```

Tests are functions whose names start with `test`. This function checks the basic cache functionality, writing values and reading them again.

```solidity for(uint i=1; i<5000; i++) { cache.cacheWrite(i*i); }

    for(uint i=1; i<5000; i++) {
        assertEq(cache.cacheRead(i), i*i);
```

This is how you do the actual testing, using `assert...` functions. In this case, we check that the value we wrote is the one we read. We can discard the result of `cache.cacheWrite` because we know that cache keys are assigned linearly.

```solidity } } // testCaching

// Cache the same value multiple times, ensure that the key stays
// the same
function testRepeatCaching() public {
    for(uint i=1; i<100; i++) {
        uint _key1 = cache.cacheWrite(i);
        uint _key2 = cache.cacheWrite(i);
        assertEq(_key1, _key2);
    }
```

First we write each value twice to the cache and make sure the keys are the same (meaning the second write didn't really happen).

```
solidity for(uint i=1; i<100; i+=3) { uint _key = cache.cacheWrite(i); assertEq(_key, i); } } // testRepeatCaching
```

In theory there could be a bug that doesn't affect consecutive cache writes. So here we do some writes that aren't consecutive and see the values are still not rewritten.

```
solidity // Read a uint from a memory buffer (to make sure we get back the parameters // we sent out) function
toUint256(bytes memory _bytes, uint256 _start) internal pure returns (uint256)
```

Read a 256 bit word from a `bytes memory` buffer. This utility function lets us verify that we receive the correct results when we run a function call that uses the cache.

```solidity { require(_bytes.length >= _start + 32, "toUint256_outOfBounds"); uint256 tempUint;

    assembly {
        tempUint := mload(add(add(_bytes, 0x20), _start))
    }
```

Yul does not support data structures beyond `uint256`, so when you refer to a more sophisticated data structure, such as the memory buffer `_bytes`, you get the address of that structure. Solidity stores `bytes memory` values as a 32 byte word that contains the length, followed by the actual bytes, so to get byte number `_start` we need to calculate `_bytes+32+_start`.

```solidity

    return tempUint;
}     // toUint256

// Function signature for fourParams(), courtesy of
// https://www.4byte.directory/signatures/?bytes4_signature=0x3edc1e6d
bytes4 constant FOUR_PARAMS = 0x3edc1e6d;

// Just some constant values to see we're getting the correct values back
uint256 constant VAL_A = 0xDEAD60A7;
uint256 constant VAL_B =     0xBEEF;
uint256 constant VAL_C =     0x600D;
```

```
uint256 constant VAL_D = 0x600D60A7;
```

Some constants we need for testing.

```
solidity function testReadParam() public {
```

Call `fourParams()`, a function that uses `readParams`, to test we can read parameters correctly.

```
solidity address _cacheAddr = address(cache); bool _success; bytes memory _callInput; bytes memory _callOutput;
```

We can't use the normal ABI mechanism to call a function using the cache, so we need to use the low level `<address>.call()` mechanism. That mechanism takes a `bytes memory` as input, and returns that (as well as a Boolean value) as output.

```
solidity // First call, the cache is empty _callInput = bytes.concat( FOUR_PARAMS,
```

It is useful for the same contract to support both cached functions (for calls directly from transactions) and non-cached functions (for calls from other smart contracts). To do that we need to continue to rely on the Solidity mechanism to call the correct function, instead of putting everything in a `fallback` function. Doing this makes composability a lot easier. A single byte would be enough to identify the function in most cases, so we are wasting three bytes (16*3=48 gas). However, as I'm writing this those 48 gas cost 0.07 cents, which is a reasonable cost of simpler, less bug prone, code.

```
solidity // First value, add it to the cache cache.INTO_CACHE(), bytes32(VAL_A),
```

The first value: A flag saying it's a full value that needs to be written to the cache, followed by the 32 bytes of the value. The other three values are similar, except that `VAL_B` isn't written to the cache and `VAL_C` is both the third parameter and the fourth one.

```
solidity . . . ); (_success, _callOutput) = _cacheAddr.call(_callInput);
```

This is where we actually call the `Cache` contract.

```
solidity assertEq(_success, true);
```

We expect the call to be successful.

```
solidity assertEq(cache.cacheRead(1), VAL_A); assertEq(cache.cacheRead(2), VAL_C);
```

We start with an empty cache and then add `VAL_A` followed by `VAL_C`. We'd expect the first one to have the key 1, and the second one to have 2.

```
assertEq(toUint256(_callOutput,0), VAL_A); assertEq(toUint256(_callOutput,32), VAL_B);
assertEq(toUint256(_callOutput,64), VAL_C); assertEq(toUint256(_callOutput,96), VAL_C);
```

The output is the four parameters. Here we verify it is correct.

```solidity // Second call, we can use the cache _callInput = bytes.concat( FOUR_PARAMS,

        // First value in the Cache
        bytes1(0x01),
```

Cache keys below 16 are just one byte.

```solidity // Second value, don't add it to the cache cache.DONT_CACHE(), bytes32(VAL_B),

        // Third and fourth values, same value
        bytes1(0x02),
        bytes1(0x02)
    );
    .
    .
    .
}   // testReadParam
```

The tests after the call are identical to those after the first call.

```
solidity function testEncodeVal() public {
```

This function is similar to `testReadParam`, except that instead of writing the parameters explicitly we use `encodeVal()`.

```
solidity . . . _callInput = bytes.concat( FOUR_PARAMS, cache.encodeVal(VAL_A), cache.encodeVal(VAL_B),
cache.encodeVal(VAL_C), cache.encodeVal(VAL_D) ); . . . assertEq(_callInput.length, 4+1*4); } // testEncodeVal
```

The only additional test in `testEncodeVal()` is to verify that the length of `_callInput` is correct. For the first call it is 4+33*4. For the second, where every value is already in the cache, it is 4+1*4.

```
solidity // Test encodeVal when the key is more than a single byte // Maximum three bytes because filling the cache
to four bytes takes // too long. function testEncodeValBig() public { // Put a number of values in the cache. // To
keep things simple, use key n for value n. for(uint i=1; i<0x1FFF; i++) { cache.cacheWrite(i); }
```

The `testEncodeVal` function above only writes four values into the cache, so[the part of the function that deals with multi-byte values] is not checked. But that code is complicated and error-prone.

The first part of this function is a loop that writes all the values from 1 to 0x1FFF to the cache in order, so we'll be able to encode those values and know where they are going.

```solidity . . .

    _callInput = bytes.concat(
        FOUR_PARAMS,
        cache.encodeVal(0x000F),   // One byte        0x0F
        cache.encodeVal(0x0010),   // Two bytes     0x1010
        cache.encodeVal(0x0100),   // Two bytes     0x1100
        cache.encodeVal(0x1000)    // Three bytes 0x201000
    );

```

Test one byte, two byte, and three byte values. We don't test beyond that because it would take too long to write enough stack entries (at least 0x10000000, approximately a quarter of a billion).

```solidity . . . . } // testEncodeValBig

```
// Test what with an excessively small buffer we get a revert
function testShortCalldata() public {

```

Test what happens in the abnormal case where there aren't enough parameters.

```
solidity . . . (_success, _callOutput) = _cacheAddr.call(_callInput); assertEq(_success, false); } //
testShortCalldata
```

Since it reverts, the result we should get is `false`.

``` // Call with cache keys that aren't there function testNoCacheKey() public { . . . _callInput = bytes.concat( FOUR_PARAMS,

        // First value, add it to the cache
        cache.INTO_CACHE(),
        bytes32(VAL_A),

        // Second value
        bytes1(0x0F),
        bytes2(0x1234),
        bytes11(0xA10102030405060708090A)
    );

```

This function gets four perfectly legitimate parameters, except that the cache is empty so there are no values there to read.

```solidity . . . // Test what with an excessively long buffer everything works file function testLongCalldata() public { address _cacheAddr = address(cache); bool _success; bytes memory _callInput; bytes memory _callOutput;

    // First call, the cache is empty
    _callInput = bytes.concat(
        FOUR_PARAMS,

        // First value, add it to the cache
        cache.INTO_CACHE(), bytes32(VAL_A),

        // Second value, add it to the cache
```
```

```
        cache.INTO_CACHE(), bytes32(VAL_B),

        // Third value, add it to the cache
        cache.INTO_CACHE(), bytes32(VAL_C),

        // Fourth value, add it to the cache
        cache.INTO_CACHE(), bytes32(VAL_D),

        // And another value for "good luck"
        bytes4(0x31112233)
    );

```

This function sends five values. We know that the fifth value is ignored because it is not a valid cache entry, which would have caused a revert had it not been included.

```solidity (_success, _callOutput) = _cacheAddr.call(_callInput); assertEq(_success, true); . . . } // testLongCalldata

} // CacheTest

```

A sample application {#a-sample-app}

Writing tests in Solidity is all very well, but at the end of the day a dapp needs to be able to process requests from outside the chain to be useful. This article demonstrates how to use caching in a dapp with `WORM`, which stands for "Write Once, Read Many". If a key is not yet written, you can write a value to it. If the key is written already, you get a revert.

The contract {#the-contract}

This is the contract. It mostly repeats what we have already done with`Cache` and `CacheTest`, so we only cover the parts that are interesting.

```solidity import "./Cache.sol";

contract WORM is Cache { ```

The easiest way to use `Cache` is to inherit it in our own contract.

```
solidity function writeEntryCached() external { uint[] memory params = _readParams(2); writeEntry(params[0], params[1]); } // writeEntryCached
```

This function is similar to `fourParam` in `CacheTest` above. Because we don't follow the ABI specifications, it is best not to declare any parameters into the function.

```
solidity // Make it easier to call us // Function signature for writeEntryCached(), courtesy of // https://www.4byte.directory/signatures/?bytes4_signature=0xe4e4f2d3 bytes4 constant public WRITE_ENTRY_CACHED = 0xe4e4f2d3;
```

The external code that calls `writeEntryCached` will need to manually build the calldata, instead of using`worm.writeEntryCached`, because we do not follow the ABI specifications. Having this constant value just makes it easier to write it.

Note that even though we define `WRITE_ENTRY_CACHED` as a state variable, to read it externally it is necessary to use the getter function for it, `worm.WRITE_ENTRY_CACHED()`.

```
solidity function readEntry(uint key) public view returns (uint _value, address _writtenBy, uint _writtenAtBlock)
```

The read function is a `view`, so it does not require a transaction and does not cost gas. As a result, there is no benefit to using the cache for the parameter. With view functions it is best to use the standard mechanism that is simpler.

The testing code {#the-testing-code}

This is the testing code for the contract. Again, let us look only at what is interesting.

```solidity function testWReadWrite() public { worm.writeEntry(0xDEAD, 0x60A7);

```
    vm.expectRevert(bytes("entry already written"));
    worm.writeEntry(0xDEAD, 0xBEEF);
```

```

This (vm.expectRevert) is how we specify in a Foundry test that the next call should fail, and the reported reason for a failure. This applies when we use the syntax <contract>.<function name>() rather than building the calldata and calling the contract using the low level interface (<contract>.call(), etc.).

solidity function testReadWriteCached() public { uint cacheGoat = worm.cacheWrite(0x60A7);

Here we use the fact that cacheWrite returns the cache key. This is not something we'd expect to use in production, because cacheWrite changes the state, and therefore can only be called during a transaction. Transactions don't have return values, if they have results those results are supposed to be emitted as events. So the cacheWrite return value is only accessible from on-chain code, and on-chain code does not need parameter caching.

solidity (_success,) = address(worm).call(_callInput);

This is how we tell Solidity that while<contract address>.call() has two return values, we only care about the first.

solidity (_success,) = address(worm).call(_callInput); assertEq(_success, false);

Since we use the low level<address>.call() function, we can't usevm.expectRevert() and have to look at the boolean success value we get from the call.

```solidity event EntryWritten(uint indexed key, uint indexed value);

    .
    .
    .

    _callInput = bytes.concat(
        worm.WRITE_ENTRY_CACHED(), worm.encodeVal(a), worm.encodeVal(b));
    vm.expectEmit(true, true, false, false);
    emit EntryWritten(a, b);
    (_success,) = address(worm).call(_callInput);

```

This is the way we verify that codeemits an event correctly in Foundry.

The client {#the-client}

One thing you don't get with Solidity tests is JavaScript code you can cut and paste into your own application. To write that code I deployed WORM to Optimism Goerli, Optimism's new testnet. It is at address0xd34335b1d818cee54e3323d3246bd31d94e6a78a.

You can see JavaScript code for the client here. To use it:

1. Clone the git repository:

sh git clone https://github.com/qbzzt/20220915-all-you-can-cache.git

1. Install the necessary packages:

sh cd javascript yarn

1. Copy the configuration file:

sh cp .env.example .env

1. Edit .env for your configuration:

| Parameter | Value | | ---------- | ---------- | | MNEMONIC | The mnemonic for an account that has enough ETH to pay for a transaction. You can get free ETH for the Optimism Goerli network here. | | OPTIMISM_GOERLI_URL | URL to Optimism Goerli. The public endpoint, https://goerli.optimism.io, is rate limited but sufficient to what we need here |

1. Run index.js.

sh node index.js

This sample application first writes an entry to WORM, displaying the calldata and a link to the transaction on Etherscan. Then it
```

reads back that entry, and displays the key it uses and the values in the entry (value, block number, and author).

Most of the client is normal Dapp JavaScript. So again we'll only go over the interesting parts.

```javascript . . . const main = async () => { const func = await worm.WRITE_ENTRY_CACHED()

// Need a new key every time
const key = await worm.encodeVal(Number(new Date()))

```

A given slot can only be written into once, so we use the timestamp to make sure we don't reuse slots.

```javascript const val = await worm.encodeVal("0x600D")

// Write an entry const calldata = func + key.slice(2) + val.slice(2) ```

Ethers expects the call data to be a hex string,`0x` followed by an even number of hexadecimal digits. As`key` and `val` both start with `0x`, we need to remove those headers.

```javascript const tx = await worm.populateTransaction.writeEntryCached() tx.data = calldata

sentTx = await wallet.sendTransaction(tx) ```

As with the Solidity testing code, we cannot call a cached function normally. Instead, we need to use a lower level mechanism.

```
javascript . . . // Read the entry just written const realKey = '0x' + key.slice(4) // remove the FF flag const
entryRead = await worm.readEntry(realKey) . . .
```

For reading entries we can use the normal mechanism. There's no need to use parameter caching with`view` functions.

Conclusion {#conclusion}

The code in this article is a proof of concept, the purpose is to make the idea easy to understand. For a production-ready system you might want to implement some additional functionality:

- Handle values that aren't`uint256`. For example, strings.
- Instead of a global cache, maybe have a mapping between users and caches. Different users use different values.
- Values used for addresses are distinct from those used for other purposes. It might make sense to have a separate cache just for addresses.
- Currently, the cache keys are on a "first come, smallest key" algorithm. The first sixteen values can be sent as a single byte. The next 4080 values can be sent as two bytes. The next approximately million values are three bytes, etc. A production system should keep usage counters on cache entries and reorganize them so that the sixteen *most common* values are one byte, the next 4080 most common values two bytes, etc.

However, that is a potentially dangerous operation. Imagine the following sequence of events:

1. Noam Naive calls `encodeVal` to encode the address to which he wants to send tokens. That address is one of the first used on the application, so the encoded value is 0x06. This is a `view` function, not a transaction, so it's between Noam and the node he uses, and nobody else knows about it

2. Owen Owner runs the cache reordering operation. Very few people actually use that address, so it is now encoded as 0x201122. A different value, 10^{18}, is assigned 0x06.

3. Noam Naive sends his tokens to 0x06. They go to the address`0x000000000000000000000000de0b6b3a7640000`, and since nobody knows the private key for that address, they are just stuck there. Noam is *not happy*.

There are ways to solve this problem, and the related problem of transactions that are in the mempool during the cache reorder, but you must be aware of it.

I demonstrated caching here with Optimism, because I'm an Optimism employee and this is the rollup I know best. But it should work with any rollup that charges a minimal cost for internal processing, so that in comparison writing the transaction data to L1 is the major expense.