Querying Contract State

There are many cases where you want to view the state of a contract. Both as an external client (using the cli), but also while executing a contract. For example, we discussed resolving names like "Alice" or "Bob" in the last section, which would require a query to another contract. We will first cover the two types of queries - raw and custom - then look at the semantics of querying via anexternal client , as well as aninternal client (another contract). We will pay special attention not only to how it works practically but also to the design and security issues of executing queries from one contract to another.

Raw Queries

The simplest query to implement is just raw read access to the key-value store. If the caller (either an external client or another contract) passes in the raw binary key that is used in the contract's storage, we can easily return the raw binary value. The benefit of this approach is that it is very easy to implement and universal. The downside is that it links the caller to theimplementation of the storage and requires knowledge of the exact contract being executed.

This is implemented inside thewasmd runtime and circumvents the VM. As a consequence, it requires no support from the CosmWasm contract and all contract state is visible. Such aquery_raw function is exposed to all callers (external and internal).

Custom Queries

There are many cases where it is undesirable to couple tightly toimplementation , and we would rather depend on aninterface . For example, we will define a standard for "ERC20"ExecuteMsg for calling the contract and we would want to define such a standard for aQueryMsg . For example, query balance by address, query allowance via granter + grantee, query token info (ticker, decimals, etc). By defining a standardinterface , we allow many implementations, including complex contracts, where the "ERC20" interface is only a small subset of their functionality.

To enable custom queries, we allow each contract to expose aquery function, that can access its data store in read-only mode. It can load any data it wishes and even performs calculations on it. This method is exposed asquery_custom to all callers (external and internal). The data format (both query and response) is anything the contract desires and should be documented in the public schema, along withExecuteMsg andInitMsg .

Note that executing a contract may consume an unbounded amount of gas. Whereasquery_raw will read one key and has a small, mostly fixed cost, we need to enforce a gas limit on these queries. This is done differently for external and internal calls and is discussed below.

External Queries

External queries are the typical way all web and cli clients work with the blockchain. They call Tendermint RPC, which calls intoabci_query in the Cosmos SDK, which delegates down to the module to handle it. As far as I know, there is an infinite gas limit on queries, as they are only executed on one node, and cannot slow down the entire blockchain. This functionality is generally not exposed on validating nodes. The query functionality exposed in the current SDK is hard coded and has execution time limits designed by the developers. This limits abuse. But what about someone uploading a wasm contract with an infinite loop, and then using that to DoS any public RPC node that exposes querying?

To avoid such issues, we need to define some fixed gas limit for allquery_custom transactions called externally. This will not charge a fee but is used to limit abuse. However, it is difficult to define a standard value, for a free public node would prefer a small amount, but I may want to sync my archive node and perform complex queries. Thus, a gas limit for allquery_custom calls can be defined in an app-specific configuration file, which can be customized by each node operator, with a sensible default limit. This will allow public nodes to protect themselves from complex queries, while still allowing optional queries to perform a large aggregation over all contract data in specially-configured nodes.

Note that theabci_query call never reads the current "in-progress" state of the modules, but uses a read-only snapshot of the state after the last committed block.

Internal Queries

While many interactions between contracts can easily be modeled by sending messages, there are some cases where we would like to synchronously query other modules, without altering their state. For example, if I want to resolve a name to aAddress , or if I want to check the KYC status of some account (in another contract) before enabling an action. One could model this as a series of messages, but it is quite complex and makes such simple use-cases almost unusable in the system.

However, this design violates one of the basic principles of theactor model , namely that each contract has exclusive access to its own internal state. (Bothquery_raw andquery_custom fail in this regard). Far from just being a theoretical issue, this may lead to concurrency and reentrancy issues if not handled correctly. We do not want to push such safety-critical reasoning into the laps of the contract developers, but rather provide these security guarantees in the platform. However, providing old data also leads to many possible errors and bugs, especially since we use the sameQuerier interface to interact with the native SDK modules,including querying the contract's own balance .

As such, we provide theQuerier with read-only access to the state snapshotright before the execution of the current CosmWasm message . Since we take a snapshot and both the executing contract and the queried contract have read-only access to the databefore the contract execution , this is still safe with Rust's borrowing rules (as a placeholder for secure design). The current contract only writes to a cache, which is flushed afterward on success.

Another issue is to avoid reentrancy. Since these queries are called synchronously, they can call back into the calling contract and possibly cause issues. Since queries only have read-only access and cannot have side effects, this is not nearly as dangerous as executing a remote contract synchronously, but still may be an issue to consider. Notably, it will only have access to the state before the current execution. I cannot see a place where this could cause more errors than a query function intentionally returning false data, but it is a place to investigate more.

Since all queries are performed as part of a transaction, that already has a strongly enforced gas limit, we don't need extra work here. All storage reads and data processing performed as part of a query are deducted from the same gas meter as the rest of the transaction and thus limit processing time. We consider adding explicit guards against re-entrancy or max query depth, but have not enforced them yet inwasmd . As more work on cross-contract queries comes to fruition, this is another place to investigate. Previous Names and Addresses Next Serialization * Raw Queries * Custom Queries * External Queries * Internal Queries