Solana Validator Geyser Plugins

Overview

Validators under heavy RPC loads, such as when serving getProgramAccounts calls, can fall behind the network. To solve this problem, the validator has been enhanced to support a plugin mechanism, called a "Geyser" plugin, through which the information about accounts, slots, blocks, and transactions can be transmitted to external data stores such as relational databases, NoSQL databases or Kafka. RPC services then can be developed to consume data from these external data stores with the possibility of more flexible and targeted optimizations such as caching and indexing. This allows the validator to focus on processing transactions without being slowed down by busy RPC requests.

This document describes the interfaces of the plugin and the referential plugin implementation for the PostgreSQL database.

Important Crates:

- solana-geyser-plugin-interface
- —This crate defines the plugin
- interfaces.
- solana-accountsdb-plugin-postgres
- —The crate for the referential
- plugin implementation for the PostgreSQL database.

The Plugin Interface

The Plugin interface is declared insolana-geyser-plugin-interface . It is defined by the traitGeyserPlugin . The plugin should implement the trait and expose a "C" function_create_plugin to return the pointer to this trait. For example, in the referential implementation, the following code instantiates the PostgreSQL pluginGeyserPluginPostgres and returns its pointer.

[no_mangle]

[allow(improper_ctypes_definitions)]

/// # Safety /// /// This function returns the GeyserPluginPostgres pointer as trait GeyserPlugin. pub unsafe extern "C" fn _create_plugin() -> *mut dyn GeyserPlugin { let plugin = GeyserPluginPostgres::new(); let plugin: Box = Box::new(plugin); Box::into_raw(plugin) } A plugin implementation can implement theon_load method to initialize itself. This function is invoked after a plugin is dynamically loaded into the validator when it starts. The configuration of the plugin is controlled by a configuration file in JSON5 format. The JSON5 file must have a fieldlibpath that points to the full path name of the shared library implementing the plugin, and may have other configuration information, like connection parameters for the external database. The plugin configuration file is specified by the validator's CLI parameter--geyser-plugin-config and the file must be readable to the validator process.

Please see theconfig file for the referential PostgreSQL plugin below for an example.

The plugin can implement theon_unload method to do any cleanup before the plugin is unloaded when the validator is gracefully shutdown.

The plugin framework supports streaming either accounts, transactions or both. A plugin uses the following function to indicate if it is interested in receiving account data:

fn account_data_notifications_enabled(&self) -> bool And it uses the following function to indicate if it is interested in receiving transaction data:

fn transaction_notifications_enabled(&self) -> bool The following method is used for notifying on an account update:

fn update_account(&mut self, account: ReplicaAccountInfoVersions, slot: u64, is_startup: bool,) -> Result<()> TheReplicaAccountInfoVersions struct contains the metadata and data of the account streamed. Theslot points to the slot the account is being updated at. Whenis_startup is true, it indicates the account is loaded from snapshots when the validator starts up. Whenis_startup is false, the account is updated when processing a transaction.

The following method is called when all accounts have been notified when the validator restores the AccountsDb from snapshots at startup.

fn notify_end_of_startup(&mut self) -> Result<()> Whenupdate_account is called during processing transactions, the plugin should process the notification as fast as possible because any delay may cause the validator to fall behind the network.

Persistence to external data store is best to be done asynchronously.

The following method is used for notifying slot status changes:

fn update_slot_status(&mut self, slot: u64, parent: Option, status: SlotStatus,) -> Result<()> To ensure data consistency, the plugin implementation can choose to abort the validator in case of error persisting to external stores. When the validator restarts the account data will be re-transmitted.

The following method is used for notifying transactions:

fn notify_transaction(&mut self, transaction: ReplicaTransactionInfoVersions, slot: u64,) -> Result<()> TheReplicaTransactionInfoVersions struct contains the information about a streamed transaction. It wrapsReplicaTransactionInfo

pub struct ReplicaTransactionInfo<'a> { /// The first signature of the transaction, used for identifying the transaction. pub signature: &'a Signature,

/// Indicates if the transaction is a simple vote transaction. pub is_vote: bool,

/// The sanitized transaction. pub transaction: &'a SanitizedTransaction,

/// Metadata of the transaction status. pub transaction_status_meta: &'a TransactionStatusMeta, } For details ofSanitizedTransaction andTransactionStatusMeta , please refer tosolana-sdk andsolana-transaction-status

Theslot points to the slot the transaction is executed at. For more details, please refer to the Rust documentation insolana-geyser-plugin-interface .

Example PostgreSQL Plugin

Thesolana-accountsdb-plugin-postgres repository implements a plugin storing account data to a PostgreSQL database to illustrate how a plugin can be developed.

Configuration File Format The plugin is configured using the input configuration file. An example configuration file looks like the following:

{ "libpath": "/solana/target/release/libsolana_geyser_plugin_postgres.so", "host": "postgres-server", "user": "solana", "port": 5433, "threads": 20, "batch_size": 20, "panic_on_db_errors": true, "accounts_selector" : { "accounts" : ["*"] } } Thehost ,user , andport control the PostgreSQL configuration information. For more advanced connection options, please use theconnection_str field. Please seeRust postgres configuration.

To improve the throughput to the database, the plugin supports connection pooling using multiple threads, each maintaining a connection to the PostgreSQL database. The count of the threads is controlled by thethreads field. A higher thread count usually offers better performance.

To further improve performance when saving large numbers of accounts at startup, the plugin uses bulk inserts. The batch size is controlled by thebatch_size parameter. This can help reduce the round trips to the database.

Thepanic_on_db_errors can be used to panic the validator in case of database errors to ensure data consistency.

Account Selection

Theaccounts_selector can be used to filter the accounts that should be persisted.

For example, one can use the following to persist only the accounts with particular Base58-encoded Pubkeys,

"accounts_selector" : { "accounts" : ["pubkey-1", "pubkey-2", ..., "pubkey-n"], } Or use the following to select accounts with certain program owners:

"accounts_selector" : { "owners" : ["pubkey-owner-1", "pubkey-owner-2", ..., "pubkey-owner-m"], } To select all accounts, use the wildcard character (*):

"accounts_selector" : { "accounts" : ["*"], }

Transaction Selection

transaction_selector , controls if and what transactions to store. If this field is missing, none of the transactions are stored.

For example, one can use the following to select only the transactions referencing accounts with particular Base58-encoded Pubkeys,

"transaction_selector" : { "mentions" : ["pubkey-1", "pubkey-2", ..., "pubkey-n"], } Thementions field supports wildcards to

select all transaction or all 'vote' transactions. For example, to select all transactions:

"transaction_selector" : { "mentions" : ["*"], } To select all vote transactions:

"transaction_selector" : { "mentions" : ["all_votes"], }

Database Setup

Install PostgreSQL Server

Please followPostgreSQL Ubuntu Installation on instructions to install the PostgreSQL database server. For example, to install postgresql-14,

sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt (lsb_release -cs)-pgdg main" > /etc/apt/sources.list.d/pgdg.list' wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add - sudo apt-get update sudo apt-get -y install postgresql-14

Control the Database Access

Modify the pg_hba.conf as necessary to grant the plugin to access the database. For example, in /etc/postgresql/14/main/pg_hba.conf, the following entry allows nodes with IPs in the CIDR 10.138.0.0/24 to access all databases. The validator runs in a node with an ip in the specified range.

host all all 10.138.0.0/24 trust It is recommended to run the database server on a separate node from the validator for better performance.

Configure the Database Performance Parameters

Please refer to thePostgreSQL Server Configuration for configuration details. The referential implementation uses the following configurations for better database performance in the /etc/postgresql/14/main/postgresql.conf which are different from the default postgresql-14 installation.

max_connections = 200 # (change requires restart) shared_buffers = 1GB # min 128kB effective_io_concurrency = 1000 # 1-1000; 0 disables prefetching wal_level = minimal # minimal, replica, or logical fsync = off # flush data to disk for crash safety synchronous_commit = off # synchronization level; full_page_writes = off # recover from partial page writes max_wal_senders = 0 # max number of walsender processes The samplepostgresql.conf can be used for reference.

Create the Database Instance and the Role

Start the server:

sudo systemctl start postgresql@14-main Create the database. For example, the following creates a database named 'solana':

sudo -u postgres createdb solana -p 5433 Create the database user. For example, the following creates a regular user named 'solana':

sudo -u postgres createuser -p 5433 solana Verify the database is working using psql. For example, assuming the node running PostgreSQL has the ip 10.138.0.9, the following command will land in a shell where SQL commands can be entered:

psql -U solana -p 5433 -h 10.138.0.9 -w -d solana

Create the Schema Objects

Use thecreate_schema.sql to create the objects for storing accounts and slots.

Download the script from github:

wget https://raw.githubusercontent.com/solana-labs/solana/a70eb098f4ae9cd359c1e40bbb7752b3dd61de8d/accountsdb-plugin-postgres/scripts/create_schema.sql Then run the script:

psql -U solana -p 5433 -h 10.138.0.9 -w -d solana -f create_schema.sql After this, start the validator with the plugin by using the--geyser-plugin-config argument mentioned above.

Destroy the Schema Objects

To destroy the database objects, created bycreate_schema.sql , usedrop_schema.sql . For example,

psql -U solana -p 5433 -h 10.138.0.9 -w -d solana -f drop_schema.sql

Capture Historical Account Data

To capture account historical data, in the configuration file, turnstore_account_historical_data to true.

And ensure the database trigger is created to save data in theaudit_table when records inaccount are updated, as shown increate_schema.sql ,

CREATE FUNCTION audit_account_update() RETURNS trigger AS audit_account_update BEGIN INSERT INTO account_audit (pubkey, owner, lamports, slot, executable, rent_epoch, data, write_version, updated_on) VALUES (OLD.pubkey, OLD.owner, OLD.lamports, OLD.slot, OLD.executable, OLD.rent_epoch, OLD.data, OLD.write_version, OLD.updated_on); RETURN NEW; END;

audit_account_update LANGUAGE plpgsql;

CREATE TRIGGER account_update_trigger AFTER UPDATE OR DELETE ON account FOR EACH ROW EXECUTE PROCEDURE audit_account_update(); The trigger can be dropped to disable this feature, for example,

DROP TRIGGER account_update_trigger ON account; Over time, the account_audit can accumulate large amount of data. You may choose to limit that by deleting older historical data.

For example, the following SQL statement can be used to keep up to 1000 of the most recent records for an account:

delete from account_audit a2 where (pubkey, write_version) in (select pubkey, write_version from (select a.pubkey, a.updated_on, a.slot, a.write_version, a.lamports, rank() OVER (partition by pubkey order by write_version desc) as rnk from account_audit a) ranked where ranked.rnk > 1000)

Main Tables

The following are the tables in the Postgres database

Table Description account Account data slot Slot metadata transaction Transaction data account_audit Account historical data

Performance Considerations

When a validator lacks sufficient compute power, the overhead of saving the account data can cause it to fall behind the network especially when all accounts or a large number of accounts are selected. The node hosting the PostgreSQL database need to be powerful enough to handle the database loads as well. It has been found using GCP n2-standard-64 machine type for the validator and n2-highmem-32 for the PostgreSQL node is adequate for handling transmitting all accounts while keeping up with the network. In addition, it is best to keep the validator and the PostgreSQL in the same local network to reduce latency. You may need to size the validator and database nodes differently if serving other loads.
Previous Validator Runtime Next Operating a Validator