Hype Indexer

info NEAR QueryAPI is currently under development. Users who want to test-drive this solution need to be added to the allowlist before creating or forking QueryAPI indexers.

You can request access throughthis link .

Overview

This indexer creates a new row in a pre-definedposts orcomments table created by the user in the GraphQL database for every new post or comment found on the blockchain that contains either "PEPE" or "DOGE" in the contents. This is a simple example that shows how to specify two tables, filter blockchain transaction data for a specific type of transaction and its contents, and save the data to the database.

tip This indexer can be found byfollowing this link .

Defining the Database Schema

The first step to creating an indexer is to define the database schema. This is done by editing theschema.sql file in the code editor. The schema for this indexer looks like this:

CREATE

TABLE "posts"

("id"

TEXT

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0)

NOT

NULL , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , "content"

TEXT

NOT

NULL , CONSTRAINT

"posts_pkey"

PRIMARY

KEY

("id")) ;

CREATE

TABLE "comments"

("id"

SERIAL

NOT

NULL , "post_id"

TEXT

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0)

NOT

NULL , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , "content"

TEXT

NOT

NULL , CONSTRAINT

"comments_pkey"

PRIMARY

KEY

("id")) ; This schema defines two tables:posts andcomments . Theposts table has columns:

- id
- : a unique identifier for each row in the table
- account_id
- : the account ID of the user who created the post
- block_height
- : the height of the block in which the post was created
- block_timestamp
- : the timestamp of the block in which the post was created
- receipt_id

- : the receipt ID of the transaction that created the post
- content
- : the content of the post

Thecomments table has columns:

- id
- : a unique identifier for each row in the table
- post_id
- : the ID of the post that the comment was made on
- account_id
- : the account ID of the user who created the comment
- block_height
- : the height of the block in which the comment was created
- block_timestamp
- : the timestamp of the block in which the comment was created
- receipt_id
- : the receipt ID of the transaction that created the comment
- content
- : the content of the comment

Defining the indexing logic

The next step is to define the indexing logic. This is done by editing theindexingLogic.js file in the code editor. The logic for this indexer can be divided into two parts:

1. Filtering blockchain transactions for a specific type of transaction
2. Saving the data from the filtered transactions to the database

Filtering Blockchain transactions

The first part of the logic is to filter blockchain transactions for a specific type of transaction. This is done by using thegetBlock function. This function takes in a block and a context and returns a promise. The block is a Near Protocol block, and the context is a set of helper methods to retrieve and commit state. ThegetBlock function is called for every block on the blockchain.

ThegetBlock function for this indexer looks like this:

import

{

Block

}

from

"@near-lake/primitives" ;

async

function

getBlock (block :

Block , context)

{ const

SOCIAL_DB

=

"social.near" ;

function

base64decode (encodedValue)

{ let buff =

Buffer . from (encodedValue ,

"base64") ; return

JSON . parse (buff . toString ("utf-8")) ; }

function

get_near_social_posts_comments (block_type = block , DB

=

SOCIAL_DB , decodeFunction = base64decode)

{ const nearSocialPostsComments = block_type . actions ( ) . filter ((action)

=> action . receiverId

===

DB) . flatMap ((action)

=> action . operations . map ((operation)

=> operation ["FunctionCall"]) . filter ((operation)

=> operation ?. methodName ===

"set") . map ((functionCallOperation)

=>

({ ... functionCallOperation , args :

decodeFunction (functionCallOperation . args) , receiptId : action . receiptId ,

// providing receiptId as we need it })) . filter ((functionCall)

=>

{ const accountId =

Object . keys (functionCall . args . data) [0] ; return

(Object . keys (functionCall . args . data [accountId]) . includes ("post")

|| Object . keys (functionCall . args . data [accountId]) . includes ("index")) ; })) ; return nearSocialPostsComments ; }

const nearSocialPostsComments =

get_near_social_posts_comments ( ) ;

...

// Further filtering for posts/comments that contain "PEPE" or "DOGE" in the contents and saving the data to the database is done in the next section } Again, like with theposts-indexer or thefeed-indexer , this filter selects transactions that are of typeFunctionCall to theset method on the contractsocial.near on the network. In addition, it searches forpost orindex string in the data for the call.

Saving the data to the Database

The second part of the logic is to save the data from the filtered transactions to the database. This section also performs the filtering of transactions for posts and comments that contain "PEPE" or "DOGE" in the contents.

The logic for this looks like:

...

// Logic for filtering blockchain transactions is above

if

(nearSocialPostsComments . length

0)

{ const blockHeight = block . blockHeight ; const blockTimestamp =

Number (block . header ( ) . timestampNanosec) ; await

Promise . all (nearSocialPostsComments . map (async

(postAction)

=>

{ const accountId =

Object . keys (postAction . args . data) [0] ; console . log (ACCOUNT_ID: { accountId }) ;

const isPost = postAction . args . data [accountId] . post

&& Object . keys (postAction . args . data [accountId] . post) . includes ("main") ; const isComment = postAction . args . data [accountId] . post

&& Object . keys (postAction . args . data [accountId] . post) . includes ("comment") ;

if

(isPost)

{ const isHypePost = postAction . args . data [accountId] . post . main . includes ("PEPE")

|| postAction . args . data [accountId] . post . main . includes ("DOGE") ; if

(! isHypePost)

{ return ; } console . log ("Creating a post...") ; const postId =

{ accountId } : { blockHeight } ; await

createPost (postId , accountId , blockHeight , blockTimestamp , postAction . receiptId , postAction . args . data [accountId] . post . main) ; } if

(isComment)

{ const commentString =

JSON . parse (postAction . args . data [accountId] . post . comment) ; const isHypeComment = commentString . includes ("PEPE")

|| commentString . includes ("DOGE") ; if

(! isHypeComment)

{ return ; } console . log ("Creating a comment...") ; const postBlockHeight = postAction . args . data [accountId] . post . blockHeight ; const postId =

{ accountId } : { postBlockHeight } ; await

createComment (accountId , postId , blockHeight , blockTimestamp , postAction . receiptId , commentString) ; } })) ; }

...

// Definitions for createPost and createComment are below

createPost

Creating a post is done by using the[context.db.Posts.insert()](context.db.Posts.insert()) function:

async

function

createPost (postId , accountId , blockHeight , blockTimestamp , receiptId , postContent)

{ try

{ const postObject =

{ id : postId , account_id : accountId , block_height : blockHeight , block_timestamp : blockTimestamp , receipt_id : receiptId , content : postContent , } ; await context . db . Posts . insert (postObject) ; console . log ("Post created!") ; }

catch

(error)

{ console . error (error) ; } }

createComment

Creating a comment is done by using thecontext.db.Comments.insert() function:

async

function

createComment (accountId , postId , blockHeight , blockTimestamp , receiptId , commentContent)

{ try

{ const commentObject =

{ account_id : accountId , post_id : postId , block_height : blockHeight , block_timestamp : blockTimestamp , receipt_id : receiptId , content : commentContent , } ; await context . db . Comments . insert (commentObject) ; console . log ("Comment created!") ; }

catch

(error)

{ console . error (error) ; } }

Querying data from the indexer

The final step is querying the indexer using the public GraphQL API. This can be done by writing a GraphQL query using the GraphiQL tab in the code editor.

For example, here's a query that fetchesposts andcomments from theHype Indexer , ordered byblock_height :

query

MyQuery

{ < user - name

_near_hype_indexer_posts (order_by :

{ block_height :

desc })

{ account_id block_height content } < user - name

_near_hype_indexer_comments (order_by :

{ block_height :

desc })

{ account_id block_height content } } Once you have defined your query, you can use the GraphiQL Code Exporter to auto-generate a JavaScript or NEAR Widget code snippet. The exporter will create a helper methodfetchGraphQL which will allow you to fetch data from the indexer's GraphQL API. It takes three parameters:

- operationsDoc
- : A string containing the queries you would like to execute.

- operationName
- : The specific query you want to run.
- variables
- : Any variables to pass in that your query supports, such asoffset
- andlimit
- for pagination.

Next, you can call thefetchGraphQL function with the appropriate parameters and process the results.

Here's the complete code snippet for a NEAR component using theHype Indexer :

```
const

QUERYAPI_ENDPOINT

=

https://near-queryapi.api.pagoda.co/v1/graphql/ ;

State . init ( { data :

[ ] } ) ;

const query =

query MyHypeQuery { <user-name>_near_hype_indexer_posts(order_by: {block_height: desc}) { account_id block_height content } <user-name>_near_hype_indexer_comments(order_by: {block_height: desc}) { account_id block_height content } }

function

fetchGraphQL ( operationsDoc , operationName , variables )

{ return

asyncFetch ( QUERYAPI_ENDPOINT , { method :

"POST" , headers :

{

"x-hasura-role" :

<user-name>_near

} , body :

JSON . stringify ( { query : operationsDoc , variables : variables , operationName : operationName , } ) , } ) ; }

fetchGraphQL ( query ,

"MyHypeQuery" ,

{ } ) . then ( ( result )

=>

{ if

( result . status

===

200 )

{ if

( result . body . data )

{ const data = result . body . data . < user - name

        _near_hype_indexer_posts ; State . update ( { data } ) console . log ( data ) ; } } } ) ;

const
```

renderData

=

(a)

=>

{ return

(< div key = { JSON . stringify (a) }

{ JSON . stringify (a) } < / div

) ; } ;

const renderedData = state . data . map (renderData) ; return

({ renderedData }) ; tip To view a more complex example, see this widget which fetches posts with proper pagination:Posts Widget powered By QueryAPI . Edit this page Last updatedonApr 10, 2024 bygagdiez Was this page helpful? Yes No Need some help?Chat with us or check ourDev Resources ! Twitter Telegram Discord Zulip

Previous Posts Indexer Next NFTs Indexer