Social Feed 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 .

Runningfeed-indexer

The indexerindexingLogic.js is comprised of functions that help handle, transform and record data. The main logic for handling transaction data as it occurs from the blockchain can be found underneath the comment marked:

// Add your code here A schema is also specified for the tables in which data from relevant transactions is to be persisted, this can be found in theschema.sql tab.

tip This indexer can be found byfollowing this link .

Schema Definition

note Note that database tables are named asroshaan_near_feed_indexer_posts which follows the format*near*_ .

Schema-Defined Table Names

CREATE

TABLE "posts"

("id"

SERIAL

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , "content"

TEXT

NOT

NULL , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , "accounts_liked" JSONB NOT

NULL

DEFAULT

'[]' , "last_comment_timestamp"

DECIMAL (20 ,

0) , CONSTRAINT

"posts_pkey"

PRIMARY

KEY

("id")) ;

CREATE

TABLE "comments"

("id"

SERIAL

NOT

NULL , "post_id"

SERIAL

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0)

NOT

NULL , "content"

TEXT

NOT

NULL , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , CONSTRAINT

"comments_pkey"

PRIMARY

KEY

("id")) ;

CREATE

TABLE "post_likes"

("post_id"

SERIAL

NOT

NULL , "account_id"

VARCHAR

NOT

NULL , "block_height"

DECIMAL (58 ,

0) , "block_timestamp"

DECIMAL (20 ,

0)

NOT

NULL , "receipt_id"

VARCHAR

NOT

NULL , CONSTRAINT

"post_likes_pkey"

PRIMARY

KEY

("post_id" ,

"account_id")) ;

CREATE

UNIQUE

INDEX

"posts_account_id_block_height_key"

ON

"posts"

("account_id"

ASC ,

"block_height"

ASC) ;

CREATE

UNIQUE

INDEX

"comments_post_id_account_id_block_height_key"

ON

"comments"

("post_id"

ASC , "account_id"

ASC , "block_height"

ASC) ;

CREATE

INDEX "posts_last_comment_timestamp_idx"

ON

"posts"

("last_comment_timestamp"

DESC) ;

ALTER

TABLE "comments" ADD CONSTRAINT

"comments_post_id_fkey"

FOREIGN

KEY

("post_id")

REFERENCES

"posts"

("id")

ON

DELETE

NO

ACTION

ON

UPDATE

NO

ACTION ;

ALTER

TABLE "post_likes" ADD CONSTRAINT

"post_likes_post_id_fkey"

FOREIGN

KEY

("post_id")

REFERENCES

"posts"

("id")

ON

DELETE

CASCADE

ON

UPDATE

NO

ACTION ; The tables declared in the schema definition are created when the indexer is deployed. In this schema definition, three tables are created:posts ,comments andpost_likes . Indexes are then defined for each and then foreign key dependencies.

Main Function

The main function can be explained in two parts. The first filters relevant transactional data for processing by the helper functions defined earlier in the file scope, the second part uses the helper functions to ultimately save the relevant data to for querying by applications.

Filtering for Relevant Data

const

SOCIAL_DB

=

"social.near" ;

const nearSocialPosts = block . actions ( ) . filter ((action)

=> action . receiverId

===

SOCIAL_DB) . flatMap ((action)

=> action . operations . map ((operation)

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

=> operation ?. methodName ===

"set") . map ((functionCallOperation)

=>

{ try

{ const decodedArgs =

base64decode (functionCallOperation . args) ; return

{ ... functionCallOperation , args : decodedArgs , receiptId : action . receiptId , } ; }

catch

(error)

{ console . log ("Failed to decode function call args" , functionCallOperation , error) ; } }) . filter ((functionCall)

=>

{ try

{ 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")) ; }

catch

(error)

{ console . log ("Failed to parse decoded function call" , functionCall , error) ; } })) ; We first designate the near account ID that is on the receiving end of the transactions picked up by the indexer, asSOCIAL_DB = "social.near" and later with the equality operator for this check. This way we only filter for transactions that are relevant to thesocial.near account ID for saving data on-chain.

The filtering logic then begins by callingblock.actions() whereblock is defined within the@near-lake/primtives package. The output from this filtering is saved in anearSocialPosts variable for later use by the helper functions. The.filter() line helps specify for transactions exclusively that have interacted with the SocialDB..flatMap() specifies the types of transaction and looks for attributes in the transaction data on which to base the filter.

Specifically,.flatMap() filters forFunctionCall call types, calling theset method of the SocialDB contract. In addition, we look for transactions that include areceiptId and include eitherpost orindex in the function call argument data.

Processing Filtered Data

if

(nearSocialPosts . length

0)

{ console . log ("Found Near Social Posts in Block...") ; const blockHeight = block . blockHeight ; const blockTimestamp = block . header ( ) . timestampNanosec ; await

Promise . all (nearSocialPosts . map (async

(postAction)

=>

{ const accountId =

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

// if creates a post if

(postAction . args . data [accountId] . post

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

{ console . log ("Creating a post...") ; await

handlePostCreation (...

// arguments required for handlePostCreation) ; }

else

if

(postAction . args . data [accountId] . post

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

{ // if creates a comment await

handleCommentCreation (...

// arguments required for handleCommentCreation) ; }

else

if

(Object . keys (postAction . args . data [accountId]) . includes ("index"))

{ // Probably like or unlike action is happening if

(Object . keys (postAction . args . data [accountId] . index) . includes ("like"))

{ console . log ("handling like") ; await

handleLike (...

// arguments required for handleLike) ; } } })) ; } This logic is only entered if there are anynearSocialPosts , in which case it first declares theblockHeight andblockTimestamp variables that will be relevant when handling (transforming and persisting) the data. Then the processing for every transaction (or function call) is chained as a promise for asynchronous execution.

Within every promise, theaccountId performing the call is extracted from the transaction data first. Then, depending on the attributes in the transaction data, there is logic for handling post creation, comment creation, or a like/unlike.

Helper Functions

base64decode

function

base64decode (encodedValue)

{ let buff =

Buffer . from (encodedValue ,

"base64") ; return

JSON . parse (buff . toString ("utf-8")) ; } This function decodes a string that has been encoded in Base64 format. It takes a single argument,encodedValue , which is the Base64-encoded string to be decoded. The function returns the decoded string as a JavaScript object. Specifically:

1. TheBuffer.from()
2. method is called with two arguments:encodedValue
3. and"base64"
4. . This creates a newBuffer
5. object from theencodedValue
6. string and specifies that the encoding format is Base64.
7. TheJSON.parse()
8. method is called with theBuffer
9. object returned by theBuffer.from()
10. method as its argument. This parses theBuffer
11. object as a JSON string and returns a JavaScript object.
12. ThetoString()
13. method is called on theBuffer
14. object with"utf-8"
15. as its argument. This converts theBuffer
16. object to a string in UTF-8 format.
17. The resulting string is returned as a JavaScript object.

handlePostCreation

async

function

handlePostCreation (accountId , blockHeight , blockTimestamp , receiptId , content)

{ try

{ const postData =

{ account_id : accountId , block_height : blockHeight , block_timestamp : blockTimestamp , content : content , receipt_id : receiptId , } ;

// Call GraphQL mutation to insert a new post await context . db . Posts . insert (postData) ;

console . log (Post by { accountId } has been added to the database) ; }

catch

(e)

{ console . log (Failed to store post by { accountId } to the database (perhaps it is already stored)) ; } } An object containing the relevant data to populate theposts table defined in the schema is created first to then be passed into the graphQLcreatePost() query that creates a new row in the table.

handleCommentCreation

async

function

handleCommentCreation (accountId , blockHeight , blockTimestamp , receiptId , commentString)

{ try

{ const comment =

JSON . parse (commentString) ; const postAuthor = comment . item . path . split ("/") [0] ; const postBlockHeight = comment . item . blockHeight ;

// find post to retrieve Id or print a warning that we don't have it try

{ // Call GraphQL query to fetch posts that match specified criteria const posts =

await context . db . Posts . select ({

account_id : postAuthor ,

block_height : postBlockHeight } , 1) ; console . log (posts: { JSON . stringify (posts) }) ; if

(posts . length

===

0)

{ return ; }

const post = posts [0] ;

try

{ delete comment ["item"] ; const commentData =

{ account_id : accountId , receipt_id : receiptId , block_height : blockHeight , block_timestamp : blockTimestamp , content :

JSON . stringify (comment) , post_id : post . id , } ; // Call GraphQL mutation to insert a new comment await context . db . Comments . insert (commentData) ;

// Update last comment timestamp in Post table const currentTimestamp =

Date . now ( ) ; await context . db . Posts . update ({

id : post . id

} , {

last_comment_timestamp : currentTimestamp }) ; console . log (Comment by { accountId } has been added to the database) ; }

catch

(e)

{ console . log (Failed to store comment to the post { postAuthor } / { postBlockHeight } by { accountId } perhaps it has already been stored. Error { e }) ; } }

catch

(e)

{ console . log (Failed to store comment to the post { postAuthor } / { postBlockHeight } as we don't have the post stored.) ; } }

catch

(error)

{ console . log ("Failed to parse comment content. Skipping..." , error) ; } } To save or create a comment the relevant post is fetched first. If no posts are found the comment will not be created. If there is a post created in the graphQL DB, themutationData object is constructed for thecreateComment() graphQL query that adds a row to thecomments table. Once this row has been added, the relevant row in theposts table is updated to this comment's timestamp.

handleLike

async

function

handleLike (accountId , blockHeight , blockTimestamp , receiptId , likeContent)

{ try

{ const like =

JSON . parse (likeContent) ; const likeAction = like . value . type ;

// like or unlike const

[itemAuthor , _ , itemType]

= like . key . path . split ("/" ,

3) ; const itemBlockHeight = like . key . blockHeight ; console . log ("handling like" , receiptId , accountId) ; switch

(itemType)

{ case

"main" : try

{ const posts =

await context . db . Posts . select ({

account_id : itemAuthor ,

block_height : itemBlockHeight } , 1) ; if

(posts . length

==

0)

{ return ; }

const post = posts [0] ; switch

(likeAction)

{ case

"like" : await

_handlePostLike (post . id , accountId , blockHeight , blockTimestamp , receiptId) ; break ; case

"unlike" : await

_handlePostUnlike (post . id , accountId) ; break ; } }

catch

(e)

{ console . log (Failed to store like to post { itemAuthor } / { itemBlockHeight } as we don't have it stored in the first place.) ; } break ; case

"comment" : // Comment console . log (Likes to comments are not supported yet. Skipping) ; break ; default : // something else console . log (Got unsupported like type " { itemType } ". Skipping...) ; break ; } }

catch

(error)

{ console . log ("Failed to parse like content. Skipping..." , error) ; } } As withhandleCommentCreation , first the relevant post is sought from the DB store. If the relevant post is found, the logic proceeds to handling the like being either a like or a dislike.

_handlePostLike

async

function

_handlePostLike (postId , likeAuthorAccountId , likeBlockHeight , blockTimestamp , receiptId)

{ try

{ const posts =

await context . db . Posts . select ({

id : postId }) ; if

(posts . length

==

0)

{ return ; } const post = posts [0] ; let accountsLiked = post . accounts_liked . length

===

0 ? post . accounts_liked :

JSON . parse (post . accounts_liked) ;

if

(accountsLiked . indexOf (likeAuthorAccountId)

===

- 1)

{ accountsLiked . push (likeAuthorAccountId) ; }

// Call GraphQL mutation to update a post's liked accounts list await context . db . Posts . update ({

id : postId } , {

accounts_liked :

JSON . stringify (accountsLiked)

}) ;

const postLikeData =

{ post_id : postId , account_id : likeAuthorAccountId , block_height : likeBlockHeight , block_timestamp : blockTimestamp , receipt_id : receiptId , } ; // Call GraphQL mutation to insert a new like for a post await context . db . PostLikes . insert (postLikeData) ; }

catch

(e)

{ console . log (Failed to store like to in the database: { e }) ; } } As withhandleLike , the relevantpost is first sought from the graphQL DB table defined inschema.sql . If a post is found, theaccountsLiked array is defined from the post's previous array plus the additional account that has performed the like account inaccountsLiked.push(likeAuthorAccountId) . The graphQL query then updates theposts table to include this information. Lastly, thepostLikeMutation object is created with the required data for adding a new row to thepost_likes table.

_handlePostUnlike

async

function

_handlePostUnlike (postId , likeAuthorAccountId)

{ try

{ const posts =

await context . db . Posts . select ({

id : postId }) ; if

(posts . length

==

0)

{ return ; } const post = posts [0] ; let accountsLiked = post . accounts_liked . length

===

0 ? post . accounts_liked :

JSON . parse (post . accounts_liked) ;

console . log (accountsLiked) ;

let indexOfLikeAuthorAccountIdInPost = accountsLiked . indexOf (likeAuthorAccountId) ; if

(indexOfLikeAuthorAccountIdInPost

- 1)

{ accountsLiked . splice (indexOfLikeAuthorAccountIdInPost ,

1) ; // Call GraphQL mutation to update a post's liked accounts list await context . db . Posts . update ({

id : postId } , {

accounts_liked :

JSON . stringify (accountsLiked)

}) ; } // Call GraphQL mutation to delete a like for a post await context . db . PostLikes . delete ({ account_id : likeAuthorAccountId , post_id : postId , }) ; }

catch

(e)

{ console . log (Failed to delete like from the database: { e }) ; } } Here we also search for an existing relevant post in theposts table and if one has been found, theaccountsLiked is defined as to update it removing the account ID of the account that has performed the like action. Then a graphQLdelete query is called to remove the like from thepost_likes table.

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 fetcheslikes from theFeed Indexer , ordered byblock_height :

query

MyQuery

{ < user - name

_near_feed_indexer_post_likes (order_by :

{ block_height :

desc })

{ account_id block_height post_id } } 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 theFeed Indexer :

```
const

QUERYAPI_ENDPOINT

=

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

State . init ( { data :

[ ] } ) ;

const query =

query MyFeedQuery { <user-name>_near_feed_indexer_post_likes(order_by: {block_height: desc}) { account_id block_height post_id } }

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 ,

"MyFeedQuery" ,

{ }) . then ((result)

=>

{ if

(result . status

===

200)

{ if

(result . body . data)

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

_near_feed_indexer_post_likes ; 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 NFTs Indexer