Indexing best practices

In this article you can find suggested best practices when building blockchain indexers usingQueryAPI . If you're planning to design a production-ready indexer, please check the recommendations forindexing development anddatabase design .

Indexing development

This section presents a recommended workflow when building a new indexer using QueryAPI.

Design APIs for your UIs

If your application requires front-end User Interfaces (UIs), your first step should be to define the APIs that will be used by your front-end UI. The main objective here is to reduce the overall number of requests that each page makes to render all content. Once you define these APIs, you will have a an good overview of the data that you need to index from the blockchain.

Create a Database

Once you have a better idea of the indexed data, you can design the database to store the indexing results.

When defining your SQL database schema, consider these recommendations:

- Design forUPSERT
- s, so that indexed data can be replaced if needed
- Use foreign keys between entities for GraphQL linking
- Think of indexes (e.g. by accounts, by dates, etc.)
- Use views to generate more GraphQL queries – when youCREATE VIEW
- , QueryAPI generates a separate GraphQL query for it.

tip Check theDatabase design section to learn how to design optimal database schemas for your indexer.

Find blocks to test on

Using exploring tools such asNearBlocks , you can find a fewblock_height s with transactions to your smart contracts that you need to index. These example blocks will help you to test and debug while writing your indexer code.

Write JS code and debug

1. Start from a simpleindexingLogic.js
2. to get blockchain data dumped in a database, in a raw form. For example, start by getting theFunctionCall
3. 's arguments from the smart contract that you want to index. Then, use theGraphQL playground
4. to understand the raw dump and further analyze the data.

tip * Check theNEAR Lake Primitives * documentation * Usecontext.db * object to access your database tables * Write logs 1. Once you have figured out a good logic to process the raw data, test the processing logic byenabling debug mode 2. in theindexingLogic.js 3. editor, and set a list of block heights that contains different cases that your processing logic must handle. 4. Once your index logic extracts all data correctly as expected, you might find that you need to create new tables or change your schema to better organize the data. In that case, fork the indexer, change the SQL schema and update the indexer logic to process and store structured data. 5. If there were changes in smart contracts, e.g. changes in method and event definitions, you might need to implement conditional logic on the block height.

Deploy code and check logs

Make sure totry { } catch { } exceptions while processing each block. In thecatch section, log exceptional blocks and debug them by enabling debug mode. (set the blockheight of the problematic blocks, and run a local debug)

try

{ console . log ("Creating a Post Snapshot") ; const postData =

{ post_id : post_id , account_id : accountId , block_height : block_height , } ; await context . db . Posts . insert (postData) ; console . log (Post Snapshot with post_id { post_id } at block_height { block_height } has been added to the database) ; return

null ; }

catch

(e)

{ console . log (Error creating Post Snapshot with post_id { post_id } at block_height { block_height } : { e }) ; return e ; }

Fix bugs and redeploy

You may have to do several iterations to fix all bugs in your indexer to process all the blocks correctly. Currently QueryAPI does not allow to clean the database and change the schema, so you will need to fork your indexer, update the schema orindexingLogic , and try again. The new indexer can be namedYourIndexName_v2 ,YourIndexerName_v3 ,..._v4 , and so on. If you don't do that, your new indexing logic will re-run on old blocks, and if you don't handle re-indexing in yourindexingLogic.js , the same old data will be inserted again into the database, bringing further errors.

tip Remember to clean out old, unused indexers. If you getYourIndexerName_v8 to work, delete..._v7 ,..._v6 , so they can free resources taken from QueryAPI workers.

Generate GraphQL queries and export

When your indexer is deployed and ready, you can generate and export GraphQL queries that can be used in your front-end application, NEAR component, or any other integration.

To generate GraphQL queries:

- Use GraphiQL playground
- Click through and debug queries
- Use code exporter to NEAR components
- Changequery
- tosubscription
- for WebSockets

Database design

Designing an optimal database schema depends on the type of indexer that you want to build. Focusing on the two most common blockchain indexing use cases, you can consider:

- a database schema for an indexer doing blockchain analytics, reporting, business intelligence, and big-data queries.
- a database schema for an indexer built as a backend for a web3 dApp building interactive and responsive UIs, that tracks interactions over a specific smart contract.

info QueryAPI usesPostgreSQL 14.9 . You can find additional documentation about PostgresSQL data definition languagein this link .

Schema for Blockchain analytics

- Consider using summary tables for precomputed analytics. Example:

CREATE

TABLE summary_account_transactions_per_day (dim_signer_account_id TEXT

NOT

NULL , dim_transaction_date DATE

NOT

NULL , metric_total_transactions BIGINT

NOT

NULL , PRIMARY

KEY

(dim_signer_account_id , dim_transaction_date)) ;

INSERT

INTO summary_account_transactions_per_day (dim_signer_account_id , dim_transaction_date , metric_total_transactions) SELECT t . signer_account_id AS dim_signer_account_id , t . transaction_date AS dim_transaction_date , COUNT (*)

AS metric_total_transactions FROM transactions t WHERE t . transaction_date =

CURRENT_DATE GROUP

BY t . signer_account_id , t . transaction_date ON CONFLICT (dim_signer_account_id , dim_transaction_date) DO

UPDATE

SET metric_total_transactions = EXCLUDED . metric_total_transactions ; * If you want to do a SQLJOIN * query, use aVIEW * . For example:

CREATE

VIEW posts_with_latest_snapshot AS SELECT ps . post_id , p . parent_id , p . author_id , ps . block_height , ps . editor_id , ps . labels , ps . post_type , ps . description , ps . name , ps . sponsorship_token , ps . sponsorship_amount , ps . sponsorship_supervisor FROM posts p INNER

JOIN

(SELECT post_id , MAX (block_height)

AS max_block_height FROM post_snapshots GROUP

BY post_id) latest_snapshots ON p . id = latest_snapshots . post_id INNER

JOIN post_snapshots ps ON latest_snapshots . post_id = ps . post_id AND latest_snapshots . max_block_height = ps . block_height ;

Schema for interactive UIs

Indexing

Add indexes for efficient querying. Example:

CREATE

INDEX idx_transactions_signer_account_id ON

transactions (signer_account_id) ;

Partitioning

Utilize partitioning for large tables.

CREATE

TABLE transactions_partitioned_by_account_id (transaction_hash text

NOT

NULL , signer_account_id text

NOT

NULL , receipt_conversion_tokens_burnt numeric (45)

NULL , PRIMARY

KEY

(signer_account_id , transaction_hash)) PARTITION

BY LIST (signer_account_id) ;Edit this page Last updatedonJan 9, 2024 bygagdiez Was this page helpful? Yes No

Previous Getting Started Next Indexing Functions