How QueryAPI works

QueryApi is a streaming indexer implementation that executes custom indexing logic written by developers on the NEAR blockchain. QueryApi allows hosted execution of complex queries (ones that can't be answered by asimple RPC orEnhanced API call), data hosting, and its exposure via GraphQL endpoints.

Components involved

The QueryApi implementation integrates many different components in a single and streamlined solution. In a high-level overview, the main components are:

Components NEAR Protocol ->NEAR Lake ->Coordinator ->Runner ->Database ->API

Detailed overview

An in-depth, detailed overview of the QueryApi components:

Description

- Protocol:
- the underlying NEAR Layer-1 Blockchain, where dataBlocks
- andChunks
- are produced.
- NEAR Lake:
- an indexer which watches the Layer-1 network and stores all the events as JSON files on AWS S3. Changes are indexed as newBlocks
- arrive.
- Coordinator:
- the QueryApi coordinator indexer filters matching dataBlocks
- , runs historical processing threads, and queues developer's JS code to be indexed with these matched blocks.
- Runner:
- executes the user's indexer code, which outputs the data to the database.
- Database:
- a Postgres database where the developer's indexer data is stored, using a logical DB per user, and a logical schema per indexer function.
- API:
- a Hasura server running on Google Cloud Platform exposes a GraphQL endpoint so users can access their data from anywhere.

Historical backfill

When an indexer is created, two processes are triggered:

- real-time - starts from the block the indexer was registered (block_X
-), and will execute the indexer function on every matching block from there on.
- historical - starts from the configuredstart_from_block
- height, and will execute the indexer function for all matching blocks up untilblock_X
- .

The historical backfill process can be broken down in to two parts:indexed , andunindexed blocks.

Indexed blocks come from thenear-delta-lake bucket in S3. This bucket is populated via a DataBricks job which streams blocks from NEAR Lake, and for every account, stores the block heights that contain transactions made against them. This processed data allows QueryAPI to quickly fetch a list of block heights that match the contract ID defined on the Indexer, rather than filtering through all blocks.

NEAR Delta Lake is not updated in real time, so for the historical process to close the gap between it and the starting point of the real-time process, it must also manually process the remaining blocks. This is theunindexed portion of the backfill.

Provisioning

In summary, QueryAPI provisioning consists of the following steps:

1. Createdatabase
2. inPostgres
3. , named after account, if necessary

4. Adddatabase
5. to Hasura
6. Createschema
7. indatabase
8. Run DDL inschema
9. Track all tables inschema
10. inHasura
11. Track all foreign key relationships inschema
12. inHasura
13. Add all permissions to all tables inschema
14. for account

note This is the workflow for the initial provisioning. Nothing happens for the remaining provisions, as QueryAPI checks if it has been provisioned, and then skips these steps. info To check if an Indexer has been provisioned, QueryAPI checks if both thedatabase andschema exist. This check is what prevents the app from attempting to provision an already provisioned indexer.

Low level details

There are two main pieces to provisioning:Postgres andHasura .

Postgres

The dynamic piece in this process is the user-providedschema.sql file, which uses Data Definition Language (DDL) to describe the structure of the user's database. Theschema DDL will be executed within a new Postgres Schema named after the account + function name. For example, if you have anaccount.near andmy_function , the schema's name will beaccount_near_my_function . Each schema exists within a separate virtual database for that account, named after the account, i.e.account_near .

Hasura

After creating the new schema, Hasura is configured to expose this schema via a GraphQL endpoint. First, the database is added as a data source to Hasura. By default, all tables are hidden, so to expose them they must be "tracked".

Foreign keys in Postgres allow for nested queries via GraphQL, for example acustomer table may have a foreign relationship to anorders table, this enables the user to query the orders from a customer within a single query. These are not enabled by default, so they must be "tracked" too.

Finally, necessary permissions are added to the tables. These permissions control which GraphQL operations will be exposed. For example, theSELECT permission allows all queries, andDELETE will expose all delete mutations. A role, named after the account, is used to group these permissions. Specifying the role in the GraphQL request applies these permissions to that request, preventing access to other users data.

Who hosts QueryAPI

Pagoda Inc. runs and manages all the infrastructure of QueryAPI, including NEAR Lake nodes to store the data in JSON format on AWS S3.

- NEAR Lake indexing is hosted on AWS S3 buckets.
- Coordinator and Runners are hosted on GCP.
- Hasura GraphQL API server is hosted on GCP.

Pricing QueryAPI is currently free. Pagoda doesn't charge for storage of your indexer code and data as well as running the indexer, but usage pricing will be introduced once QueryApi is out of beta. Edit this page Last updatedonDec 6, 2023 byDamián Parrino Was this page helpful? Yes No

Previous Introduction Next Getting Started