The graph

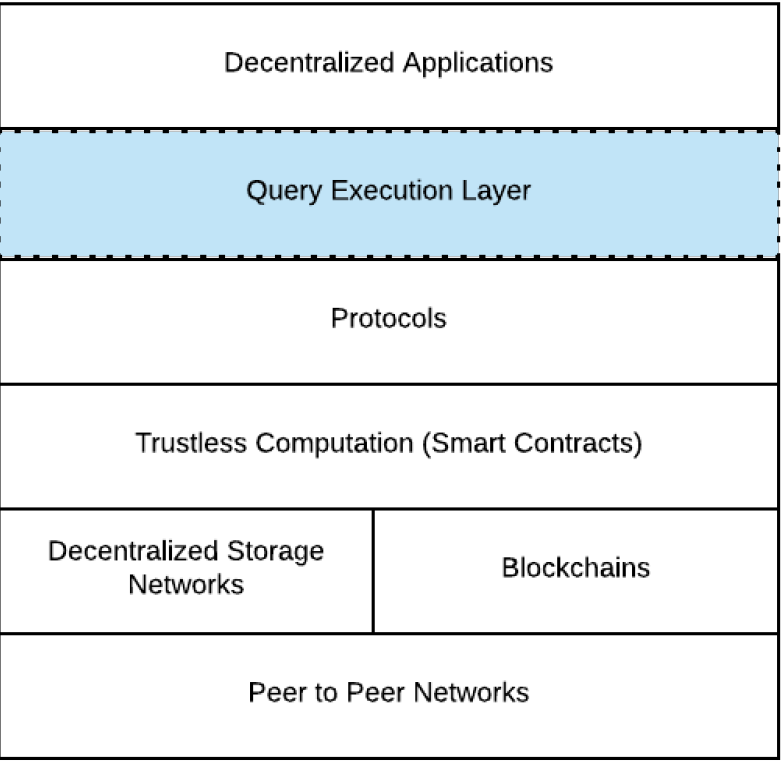
FIRST DRAFT TOTALLY TO REVIEW…..

Many ideas and text for appendix and thesis.

LINK for bibliography to the whitepaper https://github.com/graphprotocol/research/blob/master/papers/whitepaper/the-graph-whitepaper.pdf

The Graph plays a crucial role for this paper as Blockchain data are not structured for fast querying and access by processing the chain itself and ad hoc indexing is required. The Graph position itself as the google of the blockchains as it allows to efficiently access blockchain data, with the additional advantage of not relying on centralized services, offering faster and scalable access to information on-chain. Technically The Graph is a decentralized query protocol used for indexing and caching data from blockchains like Ethereum and storage networks. The vision behind The Graph is that Decentralised Applications (dApps) put users in control of their data and to create a wide-scale economic opportunity is needed an interoperability layer between web apps where applications are provided with a common way to query data. The Graph aims to provide a decentralised Query execution Layer for web 3.

FROM THE WHITEPAPER



In the whitepaper is described how The Graph achieves its goals of being a decentralised query execution layer addressing end users, app developers, the graph node operators, data source creators and the graph network validators. Summarising The Graph address indexing by allowing developers to create custom subgraphs, specialized datasets that define what data to extract from the blockchain and how it should be indexed. Once a subgraph is created, it processes new blocks, extracts relevant data, and makes it available for querying using GraphQL, an efficient query language. In appendix xxxx we provide a simple example of creating, deploying and querying a subgraph which allows to understand how the graph as a whole works. In this paper we are NOT providing our own subgraph, equivalent of a custom Aave V2 data indexing, in The Graph and we don’t take any active role in the network, differently from what we do for the ethereum itself. There is actually no need of setting up an ad hoc node or curate the indexing of Aave V2. Our goal is to access data for a specific protocol based on an existing indexing provided by a reliable curator, in our case Messari, using the exposed query language, graphql, which constitutes the external layer in front of the data sets which are exposed as REST endpoints. The protocol structure is therefore transparent to us and we focus on the graphql usage to provide us with meaningful data which facilitate finding interesting flashloans, building statistics by querying a reliable data source. Owning an ethereum node allows to use the extracted data and countercheck their validity. (See the section: process)

Usage of The Graph

In this paper we poll The Graph exposed endpoints for indexed Aave V2 data. The approach is to send different requests to filter data which facilitate further investigation. In presenting possible usage of flashloans, without a criterion to select potential arbitrages, or protocol attacks (literally hacking of protocol financed through flashloans) or collateral swapping would be almost impossible to find significant transitions without processing all the flashloans calls and examining them punctually. This is a colossal challenge as a flashloan is often wrapping plenty of transactions, even hundreds. An heuristic approach like selecting the largest 20 flashloans issued since inception assuming they will be used for an arbitrage or for hacking a protocol increases the probability of finding such transactions. A blockchain explorer helps in their analysis afterwards. This means that in this paper a recurrent pattern of work will be using in the following order The Graph, a blockchain scanner, otterscan in our case, direct queries on the node, erigon api calls in our case. The graph queries with less restrictive filters provide bulk exports of data which allow to analyse the evolution in time of flashloans calls. This approach is used in the second part of the paper where we proceed with a statistical analysis of flashloans usage on time, account type and size.

The graph runs on Arbitrum now as indexing service, an L2 chain chosen for low gas cost solution. The indexed data are selected from the proper chain, ethereum in our case, and stored through mappers in a db and exposed as already explained. Arbitrum is just a convenient solution for the protocol management.

FOR THE APPENDIX

1 HOW the graph works

To understand how the graph works under the hood is useful an example of creating, deploying and indexing a subgraph. After this is done it is possible to query the indexed data according to the defined interface. The steps consist in defining the subgraph.yaml, writing a mapper from blockchain data to indexed data telling how to transform and to save them in the db (at the moment a postgresql db), define the query language through GraphQL schema, deploying the subgraph, signalling to indexers to start to process it and finally having the data available to be queried by end users.

a.Technical prerequisites

Install Node.js and Yarn (package managers for JavaScript).

Install Graph CLI, a command-line tool to generate and manage subgraphs:

npm install -g @graphprotocol/graph-cli

b. define the Subgraph

The process begins by defining a subgraph that specifies which events, functions, or data you want to index from the blockchain. It is needed to provide a Subgraph Manifest as a YAML file (called `subgraph.yaml`) where you define your subgraph by including the network (Ethereum, Polygon, etc.), the contract addresses you're tracking and the specific smart contract events or functions to index.

Example subgraph.yaml:

`yaml

specVersion: 0.0.2

description: Example subgraph

schema:

file: ./schema.graphql

dataSources:

- kind: ethereum/contract

name: MyContract

network: mainnet

source:

address: "0x123...abc"

abi: MyContract

startBlock: 12345678

mapping:

kind: ethereum/events

apiVersion: 0.0.5

language: wasm/assemblyscript

entities:

- MyEntity

abis:

- name: MyContract

file: ./abis/MyContract.json

eventHandlers:

- event: Transfer(indexed address, indexed address, uint256)

handler: handleTransfer

file: ./src/mapping.ts

c. Write the Mapping Handlers

Mapping handlers are written in AssemblyScript and define how to process specific blockchain events. When the specified event occurs, the handler transforms it into a format that can be stored in the subgraph (saving it in a database).

Example handler for a Transfer event:

`typescript

import { Transfer } from '../generated/MyContract/MyContract'

import { MyEntity } from '../generated/schema'

export function handleTransfer(event: Transfer): void {

let entity = new MyEntity(event.transaction.hash.toHex())

entity.from = event.params.from

entity.to = event.params.to

entity.value = event.params.value

entity.save()

}

This code defines how to extract event parameters (`from`, `to`, `value`) and store them in the subgraph.

d. Define the GraphQL Schema

The schema defines how the data should be structured and queried. It is what the end user will call.

For example:

graphql

type MyEntity @entity {

id: ID!

from: Bytes!

to: Bytes!

value: BigInt!

}

This GraphQL schema defines the structure of the entity (e.g., transfers) you’re indexing from the blockchain.

d. Deploy the Subgraph

After defining the subgraph and handlers, deploy the subgraph to The Graph’s decentralized network or hosted service using:

graph deploy --node https://api.thegraph.com/deploy/ <your-subgraph-name>

e. Indexing

After the d. Step, the curator role is finished. The data are not yet available to be queried. Somebody has to index the data. This is the role of the Indexer which are incentivised to participate by the curators themselves. A curator stake some native token, GRT, and broadcasts a signal in the network. Indexers are compensated with these GRT for their work and the resources they make available.

f. querying

Once the subgraph is indexed, the end user can query it using GraphQL which allows to request specific data efficiently.

Example of querying token transfers:

`graphql

{

transfers(first: 5, orderBy: value, orderDirection: desc) {

from

to

value

}

}

This query retrieves the 5 largest transfers from the subgraph deployed by Messori that we are using in this paper, sorted by value.

2. QUERYING AN EXISTING SUBGRAPH PROGRAMMATICALLY

Javascript example

To poll data from a subgraph programmatically, a user can leverage GraphQL queries within his application to fetch real-time or periodic updates from the indexed subgraph. For example, using JavaScript with `fetch`:

`javascript

const query = `

{

transfers(first: 5, orderBy: value, orderDirection: desc) {

from

to

value

}

}`;

fetch('https://api.thegraph.com/subgraphs/name/<your-subgraph>', {

method: 'POST',

headers: {

'Content-Type': 'application/json',

},

body: JSON.stringify({ query }),

})

.then((res) => res.json())

.then((data) => console.log(data.data));

```

A more complete example in python using the `configparser` library i to read from a property file (INI-style format) that contains key-value pairs for a configuration, such as the directory name (to export responses in json format files), JSON requests, and other properties.

pip install configparser.

Create a `.ini` file (e.g., `config.properties`) with, for instance, the following structure:

`ini

directory = /path/to/your/directory

graphql\_query =

{

flashloans(first: 5, orderBy: amount, orderDirection: desc) {

id

amount

timestamp

amountUSD

blockNumber

hash

}

}

- The `directory` is the path where the JSON file will be saved.

- The `graphql\_query` is the GraphQL query that will be sent.

import os

import requests

import json

import configparser

# Function to query the subgraph and save the data to a file

def query\_subgraph\_and\_save(config):

# Read values from config

url = config.get("DEFAULT", "graphql\_endpoint")

query = config.get("DEFAULT", "graphql\_query").strip()

directory = config.get("DEFAULT", "directory")

# Define the headers

headers = {

"Content-Type": "application/json"

}

# Make the request

response = requests.post(url, json={"query": query}, headers=headers)

# Check the response

if response.status\_code == 200:

# Create the directory if it doesn't exist

if not os.path.exists(directory):

os.makedirs(directory)

# Define the file path

file\_path = os.path.join(directory, "flashloans\_response.json")

# Save the response JSON data to the file

with open(file\_path, "w") as file:

json.dump(response.json(), file, indent=4)

print(f"Data saved to {file\_path}")

else:

# If there's an error, print the status code and error message

print(f"Query failed with status code {response.status\_code}: {response.text}")

# Main function to read config and call the query function

def main():

# Create a ConfigParser object

config = configparser.ConfigParser()

# Read the config file

config.read("config.properties")

# Call the function to query and save the data

query\_subgraph\_and\_save(config)

if \_\_name\_\_ == "\_\_main\_\_":

main()

These are very simple examples to explain how the system works and simple bundle can be dumped.

Other appendix or bundle will include the code used.