Subgraph for OpenSea Smart Contract

Jiarong Shi
UNI: js6132
js6132@columbia.edu

Yutao Zhou
UNI: yz4359
yz4359@columbia.edu

Xinyuan Fu
UNI: xf2247
xf2247@columbia.edu

Zhixuan Sun
UNI: zs2572
zs2572@columbia.edu

Qingcheng Yu
UNI: qy2281
qy2281@columbia.edu

Zerui Li
UNI: zl3194
zl3194@columbia.edu

I. Introduction

OpenSea is one of the largest and most popular decentralized marketplaces for buying, selling, and discovering non-fungible tokens (NFTs). As the popularity of NFTs continues to grow, so does the demand for accessing OpenSea's smart contract data. However, querying OpenSea's smart contract data directly can be slow and resource-intensive, particularly for complex queries or large datasets. Building a subgraph for OpenSea's smart contract data can help to mitigate these issues by providing a more efficient and user-friendly way to access the data. So in this project, we develop a subgraph where users can easily query NFT data or sales history from OpenSea in a standardized format, without needing to understand the underlying smart contract code or manage complex indexing and caching themselves. This platform can make it easier for developers to build NFT-related applications on top of OpenSea and can help to foster innovation and growth in the NFT ecosystem.

II. Methodology

A. *OpenSea smart contract*

Firstly we need to understand the structure of OpenSea smart and which functions we would use. A smart contract is a self-executing contract with the terms of the agreement between buyer and seller being directly written into lines of code. It can store and manage data and execute pre-programmed actions based on specific conditions. And the OpenSea smart contract consists of several key functions, including the ability to create, transfer, and update ownership of non-fungible tokens (NFTs), see Fig. 1. Also, the contract includes functions for managing bids and sales of NFTs, as well as allowing users to set their own prices for their assets. For example, the contract Exchange and ExchangeCore account for most of OpenSea contract contents. They are responsible for managing the exchange of NFTs between buyers and sellers. We can obtain the necessary historical data by listening to the events provided in these contracts.

Also, some interfaces are provided by the smart contract, which is called ABI (Application Binary Interface). The subgraph can use the ABI to listen to events emitted by the contract and extract relevant data, which can be used to create entities and update their fields. We could download the Opensea.json abi file through the website.

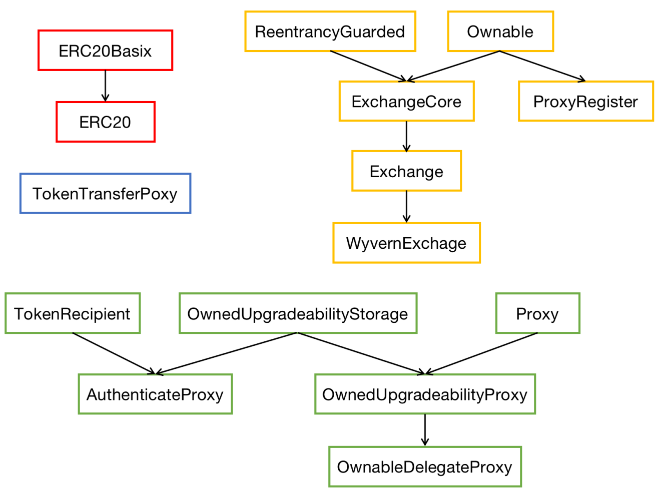


Fig. 1. The internal structure of OpenSea smart contract

B. *GraphQL*

GraphQL is a query language for APIs and a runtime for fulfilling those queries with your existing data. GraphQL allows clients to specify the exact data they need, and the server responds with only that data, reducing network overhead and improving performance. In our project, we define our GraphQL schema in schema.graphql, and write GraphQL code to do queries and testing.

C. *Subgraph*

A subgraph extracts data from a blockchain, processing it and storing it so that it can be easily queried via GraphQL. Using subgraphs, developers can define the data schema for a specific smart contract or group of contracts, and then use that schema to automatically index and store data in a database. This makes it easy to query and filter data from the blockchain and enables developers to build powerful decentralized applications that leverage blockchain data. Subgraph relies on the smart contract to provide the data that it indexes and queries.

A subgraph project consists of a few files. `subgraph.yaml` contains the subgraph manifest. `schema.graphql` is a GraphQL schema that defines what data is stored for your subgraph, and how to query it via GraphQL. Mapping.ts is AssemblyScript code that translates from the event data to the entities defined in your schema. These are the three most important files and we only need to modify them to complete the project.

III. Procedures of Work

To build a subgraph based on OpenSea smart contract, we can follow these general procedures. Firstly, we will define the scope and purpose of the subgraph, which could be to extract data about NFT transactions, owners, collections, or any other specific information related to OpenSea. Then we choose a subgraph development tool and set up the environment. What's more, we should define the GraphQL schema for the subgraph, which includes defining the types, fields, and relationships of the data that will be extracted from the smart contracts. Next, write the mapping code to extract the relevant data from the smart contracts and map it to the GraphQL schema. Last but not least we deploy and test the subgraph. In the following sections, we will deep into these steps and explain in detail the techniques we used.

A. Environment Setup and Subgraph Initialization

First and foremost, we should make sure that we have a MetaMask wallet and Node.js installed on our local machine. Then we can go to the Subgraph studio website to connect our wallet address and create a subgraph. You can see the status, subgraph slug, and deploy key in detail. Then we need to install the Graph CLI, which we will need to build and deploy our subgraph. After installation, we could create a new subgraph from OpenSea smart contract with `graph init --from-contract <contract>`. Make sure to choose the mainnet of Ethereum. Next, we could see a new directory with the original code. Then with the environment setup and project creation complete, we are ready to write the project code.

B. Schema Definition

The schema for our subgraph is in the file `schema.graphql`. All queries will be made against the data model defined in the subgraph schema and the entities indexed by the subgraph. We create six entities: User, NFT, Collection, Transaction, Auction, and Fee. They are all mutable as default, which means that mappings can load existing entities, modify them and store a new version of that entity. We also define an Event interface, which is implemented by Auction and Transaction entity. If we dig into one of the entities, for instance, the User entity shown in Listing. 1, we can see the entity structure and One-TO-Many relationships with other entities. The User entity has several required fields: id, salesNum (number of NFTs total sold), purchasesNum (number of NFTs total bought), spent (total spent money), and earned (total earned money). The nftOwneds, collectionOwneds, and transactions are collection fields derived from the relationship defined on the other entity. For example, NFT is derived from the owner field of the NFT entity. If the owner field in one NFT entity matches the certain User type, it will be added to this user. Other entity definitions are similar. Note that the selection of attributes within the entity is based on the data selection definition provided by the event in the smart contract.

Listing 1. User Entity Define

```
{
    type User @entity {
        id: ID!
        salesNum: BigInt!
        purchasesNum: BigInt!
        spent: BigDecimal!
        earned: BigDecimal!
        nftOwneds: [NFT!]! @derivedFrom(
            field: "owner")
        collectionOwneds: [Collection!]
            @derivedFrom(field: "owner")
        transactions: [Transcation!]!
            @derivedFrom(field: "buyer")
    }
}
```

C. Mapping Definition

The mappings take data from a particular source and transform it into entities that are defined within our schema. Because we create the subgraph based on OpenSea smart contract, in the manifest file `subgraph.yaml` we already have five event handlers: OrdersMatched, OrderApprovedPartOne, OrderApprovedPartTwo, OrderCancelled, OwnershipRenounced and OwnershipTransferred. We use three events, OrdersMatched, OrderApprovedPartOne, and OrderApprovedPartTwo from the Exchange contract of the OpenSea smart contract. In the `mapping.ts` file, create an exported function of the same name for each event handler. The main function of the subgraph is defined in the OrdersMatched event handler. We can get the transaction history when listening to that event. To ensure the event we get is a transfer event, we need to compare the event address and topic to the regular OpenSea transfer event defined in [2]. According to that information, we could create user and transaction schema. When creating a new entity, first use `entity.load(id)` to retrieve the existing data, if it is null, we create a new one. To get the NFT information, we can use the topics information in the Transfer event. And for collection, use the address information in the Transfer event. Similarly, the creation and update of the auction and fee schema are completed separately in handleOrderApprovedPartTwo and handleOrderApprovedPartOne.

D. Subgraph Deployment

Finally, we use the deploy key and code `graph auth studio` to deploy our subgraph to the subgraph studio. Then we can write the query and get the result in the project playground.

The query and test part is discussed in the next section, Testing Results.

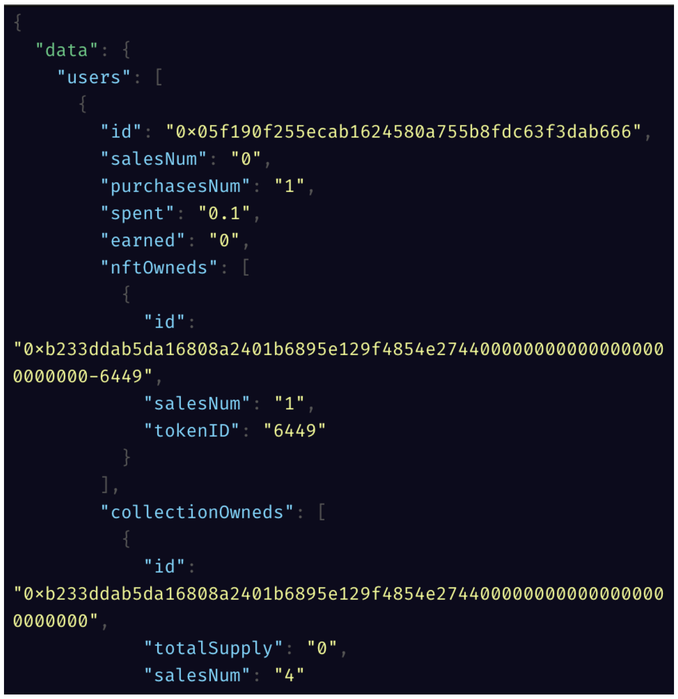
IV. Testing Results

A. *User account data query*

Firstly we can query the user schema. Users can check the number of NFTs they sold, the number of NFTs they bought, the total money they spent, and the total money they earned. Also, they can retrieve their own NFTs and collections from their wallet address. If using the ”users” field to do the query, the results will be all the users. If using the ”user” field with a specific ”id” field, the result will be a certain user with that id. The test code and querying result are shown in Listing. 2 and Fig. 2.

Listing 2. User Query Code

```
{
    users(first:5 skip:10) {
        id
        salesNum
        purchasesNum
        spent
        earned
        nftOwneds{
            id
            salesNum
            tokenID
        }
        collectionOwneds{
            id
            totalSupply
            salesNum
        }
    }
}
```



```
{
  "data": {
    "users": [
      {
        "id": "0×05f190f255ecab1624580a755b8fdc63f3dab666",
        "salesNum": "0",
        "purchasesNum": "1",
        "spent": "0.1",
        "earned": "0",
        "nftOwneds": [
          {
            "id":
"0×b233ddab5da16808a2401b6895e129f4854e274400000000000000000
0000000-6449",
            "salesNum": "1",
            "tokenID": "6449"
          }
        ],
        "collectionOwneds": [
          {
            "id":
"0×b233ddab5da16808a2401b6895e129f4854e274400000000000000000
0000000",
            "totalSupply": "0",
            "salesNum": "4"
```

Fig. 2. Result of users query

B. *NFT data query*

Then we can query the NFT schema. Users can get the id and token ID from the OpenSea smart contract. The token ID is retrieved from the ERC721 transfer event according to the buyer id and seller id. Also, users will know the collection of this NFT, the present owners, and the total sales number of this NFT. This schema guarantees that a user can retrieve a specific NFT from the OpenSea smart contract using its token ID. The test code and querying result are shown in Listing. 3 and Fig. 3.

Listing 3. NFT Query Code

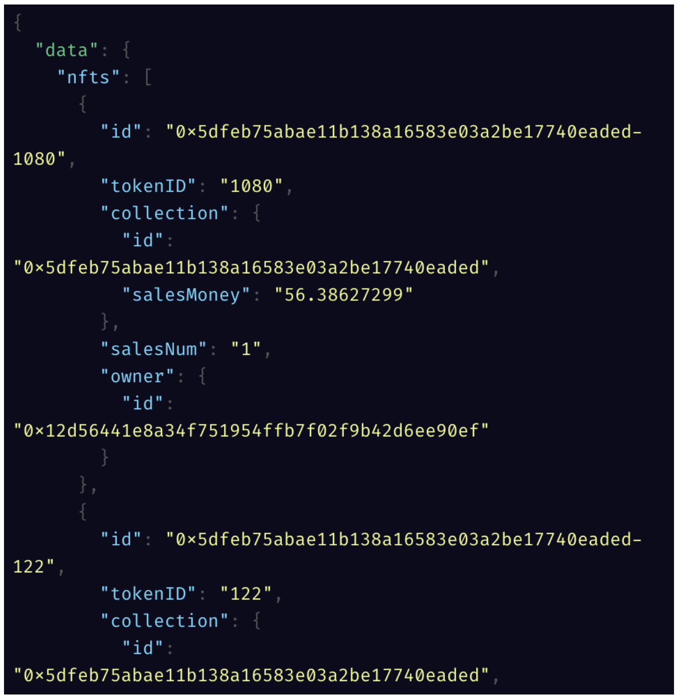
```
{
    nfts(first:5) {
        id
        tokenID
        collection {
            id
            salesMoney
        }
        salesNum
        owner {
            id
        }
    }
}
```

C. *Collection data query*

Thirdly we can query the collection schema. Users can get the id, total number of NFTs supplied, and total sales money of a certain collection from the OpenSea smart contract. Also, users will retrieve the NFTs detail information in each collection and transaction history about that collection. The test code and querying result are shown in Listing. 4 and Fig. 4.

Listing 4. Collection Query Code

```
{
    collections(first:5) {
        id
        nft {
            id
            tokenID
            salesNum
        }
        totalSupply
        salesMoney
        owner {
            id
```



```
{
  "data": {
    "nfts": [
      {
        "id": "0×5dfeb75abae11b138a16583e03a2be17740eaded-
1080",
        "tokenID": "1080",
        "collection": {
          "id":
"0×5dfeb75abae11b138a16583e03a2be17740eaded",
          "salesMoney": "56.38627299"
        },
        "salesNum": "1",
        "owner": {
          "id":
"0×12d56441e8a34f751954ffb7f02f9b42d6ee90ef"
        }
      },
      {
        "id": "0×5dfeb75abae11b138a16583e03a2be17740eaded-
122",
        "tokenID": "122",
        "collection": {
          "id":
"0×5dfeb75abae11b138a16583e03a2be17740eaded",
```

Fig. 3. Result of NFT query

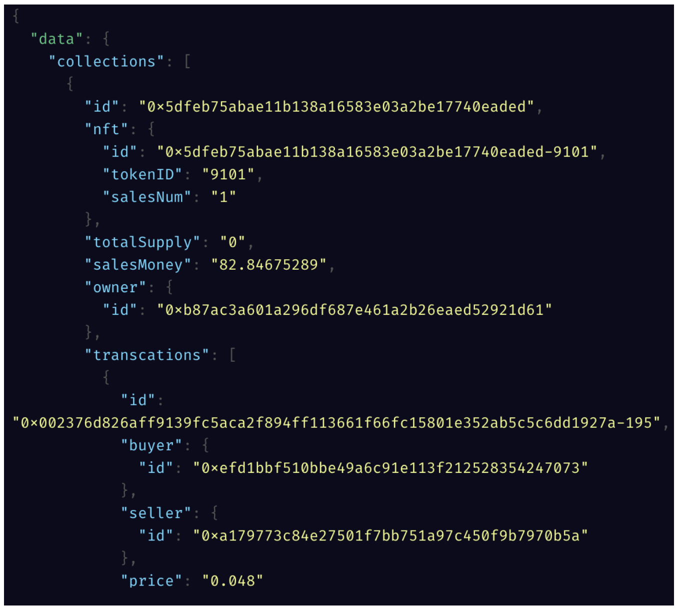
```
            }
            transcations {
                id
                buyer {id}
                seller {id}
                price
            }
        }
}
```

D. *Transaction data query*

In addition, we can query the transaction schema. Users can get the id, sellHash (ensure sell order validity and calculate hash if necessary), buyHash, buyer, seller, NFT involved, price, event type, the time of the transaction, and the block where the transaction occurred from the OpenSea smart contract. The test code and querying result are shown in Listing. 5 and Fig. 5.

Listing 5. Transaction Query Code

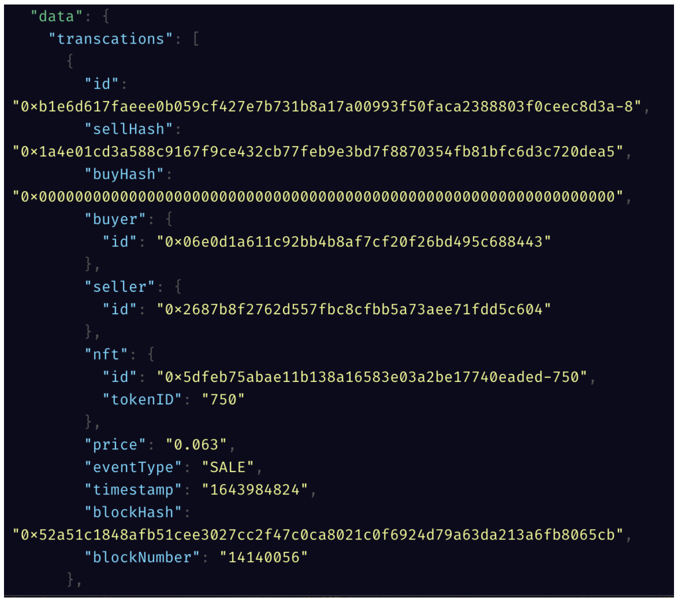
```
{
    transactions(first:5) {
        id
        sellHash
        buyHash
        buyer {id}
        seller {id}
        nft {
            id
            tokenID
            salesMoney
```



```
{
  "data": {
    "collections": [
      {
        "id": "0×5dfeb75abae11b138a16583e03a2be17740eaded",
        "nft": {
          "id": "0×5dfeb75abae11b138a16583e03a2be17740eaded-9101",
          "tokenID": "9101",
          "salesNum": "1"
        },
        "totalSupply": "0",
        "salesMoney": "82.84675289",
        "owner": {
          "id": "0×b87ac3a601a296df687e461a2b26eaed52921d61"
        },
        "transcations": [
          {
            "id":
"0×002376d826aff9139fc5aca2f894ff113661f66fc15801e352ab5c5c6dd1927a-195",
            "buyer": {
              "id": "0×efd1bbf510bbe49a6c91e113f212528354247073"
            },
            "seller": {
              "id": "0×a179773c84e27501f7bb751a97c450f9b7970b5a"
            },
            "price": "0.048"
```

Fig. 4. Result of collection query

```
            }
            price
            eventType
            timestamp
            blockHash
            blockNumber
    }
}
```



```
  "data": {
    "transcations": [
      {
        "id":
"0×b1e6d617faeee0b059cf427e7b731b8a17a00993f50faca2388803f0ceec8d3a-8",
        "sellHash":
"0×1a4e01cd3a588c9167f9ce432cb77feb9e3bd7f8870354fb81bfc6d3c720dea5",
        "buyHash":
"0×0000000000000000000000000000000000000000000000000000000000000000",
        "buyer": {
          "id": "0×06e0d1a611c92bb4b8af7cf20f26bd495c688443"
        },
        "seller": {
          "id": "0×2687b8f2762d557fbc8cfbb5a73aee71fdd5c604"
        },
        "nft": {
          "id": "0×5dfeb75abae11b138a16583e03a2be17740eaded-750",
          "tokenID": "750"
        },
        "price": "0.063",
        "eventType": "SALE",
        "timestamp": "1643984824",
        "blockHash":
"0×52a51c1848afb51cee3027cc2f47c0ca8021c0f6924d79a63da213a6fb8065cb",
        "blockNumber": "14140056"
      },
```

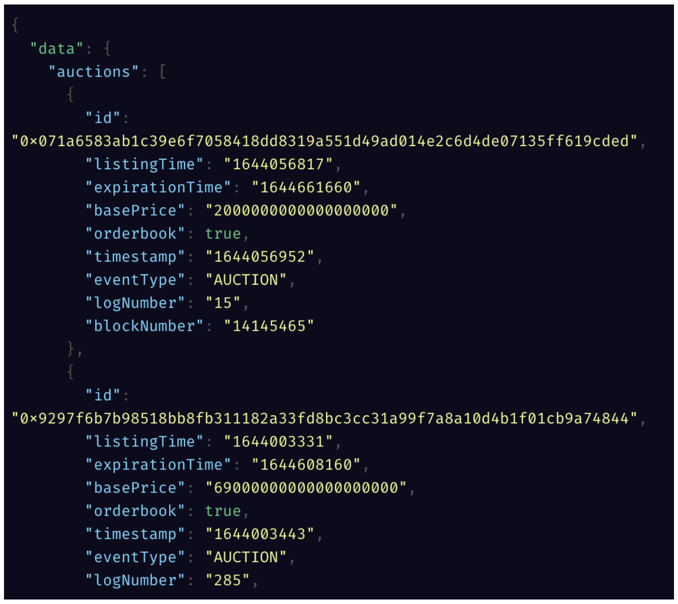
Fig. 5. Result of transaction query

E. Auction data query

What's more, we can query the auction schema. An auction in the NFT market is a method of selling a unique digital asset to the highest bidder. Users can get the id, listing time and expiration time of the auction, base price, orderbook (A record of all open orders to buy or sell a particular NFT on the platform), the time of auction released, event type, and block number from OpenSea smart contract. The test code and querying result are shown in Listing. 6 and Fig. 6.

Listing 6. Auction Query Code

```
{
    auctions(first:5) {
        id
        listingTime
        expirationTime
        basePrice
        orderbook
        timestamp
        eventType
        logNumber
        blockNumber
    }
}
```



```
{
  "data": {
    "auctions": [
      {
        "id":
"0×071a6583ab1c39e6f7058418dd8319a551d49ad014e2c6d4de07135ff619cded",
        "listingTime": "1644056817",
        "expirationTime": "1644661660",
        "basePrice": "2000000000000000000",
        "orderbook": true,
        "timestamp": "1644056952",
        "eventType": "AUCTION",
        "logNumber": "15",
        "blockNumber": "14145465"
      },
      {
        "id":
"0×9297f6b7b98518bb8fb311182a33fd8bc3cc31a99f7a8a10d4b1f01cb9a74844",
        "listingTime": "1644003331",
        "expirationTime": "1644608160",
        "basePrice": "69000000000000000000",
        "orderbook": true,
        "timestamp": "1644003443",
        "eventType": "AUCTION",
        "logNumber": "285",
```

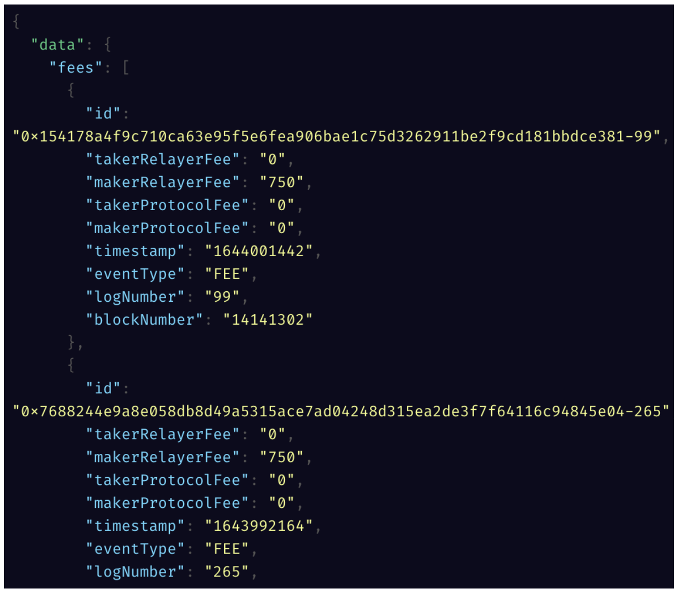
Fig. 6. Result of auction query

F. Fee data query

Last but not least, we can query the fee schema. Users can get the takerRelayerFee and makerRelayerFee, which represent the platform fee of the buyer and seller. Also, the takerProtocolFee and makerProtocolFee, they are the protocol fee charged by each transaction. The test code and querying result are shown in Listing. 7 and Fig. 7.

Listing 7. Fee Query Code

```
{
    fees(first:5) {
        id
        takerRelayerFee
        makerRelayerFee
        takerProtocolFee
        makerProtocolFee
        timestamp
        eventType
        logNumber
        blockNumber
    }
}
```



```
{
  "data": {
    "fees": [
      {
        "id":
"0×154178a4f9c710ca63e95f5e6fea906bae1c75d3262911be2f9cd181bbdce381-99",
        "takerRelayerFee": "0",
        "makerRelayerFee": "750",
        "takerProtocolFee": "0",
        "makerProtocolFee": "0",
        "timestamp": "1644001442",
        "eventType": "FEE",
        "logNumber": "99",
        "blockNumber": "14141302"
      },
      {
        "id":
"0×7688244e9a8e058db8d49a5315ace7ad04248d315ea2de3f7f64116c94845e04-265"
        "takerRelayerFee": "0",
        "makerRelayerFee": "750",
        "takerProtocolFee": "0",
        "makerProtocolFee": "0",
        "timestamp": "1643992164",
        "eventType": "FEE",
        "logNumber": "265",
```

Fig. 7. Result of fee query

V. CONCLUSION

The project successfully built a subgraph based on OpenSea smart contract, which was deployed to the Graph network for querying and retrieving data from the OpenSea marketplace. You can visit our subgraph through the GitHub link, https://github.com/QingchengYu/NFT_data_warehouse.

Contribution of team members:

1) Jiarong Shi: Subgraph initialization and deployment, Entity schema definition, Mapping.ts writing, Query test
2) Xinyuan Fu: Environment setup and Subgraph initialization and deployment
3) Zhixuan Sun: OpenSea smart contract and subgraph interaction
4) Zerui Li: Subgraph deployment and OpenSea smart contract data interaction
5) Yutao Zhou: Defined GraphQL schema in `schema.graphql` file
6) Qingcheng Yu: Environment setup, Subgraph initialization, and deployment and write schema in local machine

The report was written by everyone.

References

[1] Etherscan. Opensea smart contract. [EB/OL]. https://etherscan.io/address/0x7be8076f4ea4a4ad08075c2508e481d6c946d12b#code/.

[2] Etherscan. Regular transfer event logs. [EB/OL]. https://etherscan.io/tx/0x9660bd19edec4f443068094d3ee9cf2c9b78fbc4a7888bb1a98d154a98041d0a#eventlog/.

[3] itsjerryokolo. Subgraph for the opensea marketplace. [EB/OL]. https://thegraph.com/hosted-service/subgraph/itsjerryokolo/opensea/.

[4] thegraph.com. Creating a subgraph. [EB/OL]. http://timmurphy.org/2009/07/22/line-spacing-in-latex-documents/.