Introduction

Verified spatial data registry for reliably storing and fast querying of spatial data

We will aid Kolektivo to integrate a state of the art, web3 native GIS engine that is specifically tailored for mapping, storing, computing, and querying geographic data. Both spatial and eventually temporal (time series) data will be stored by this engine natively in web3.

A GIS engine is required for storing and querying geographical data. For example, if we were to answer the query “find all polygons or points that have temperature > T and falls within radius r,” it would require searching through all points and polygons data, filtering based on a scalar attribute, and calculating whether they intersect with the circle defined by a center (X, Y) and a threshold radius r. Such calculations are efficiently done by GIS engines as they are optimized to store and compute millions of geographic data points.

Current Web2 server based GIS engines pose a significant challenge when it comes to uniting them with smart contracts.

1) The data lives on a central server. This data needs to be oraclized and served to smart contracts. This exposes the pipeline to security threats inherent in oracles.

2) Any query computation has to be done in web2 servers. There is currently no particular way to invoke such calculations on demand from smart contracts.

Together, these limitations suggest a web3 native way to store and compute geographical data.

We have developed a powerful web3 native GIS computation engine using a geometric data structure called Quadtrees to store geographic and time series data both on chain and in decentralized storages such as IPFS. Using this engine, anyone can store geographic polygons, lines, points and trajectories, and polyhedron data in a web3 native manner. Additional attributes data can be stored in IPFS or Ceramic, and can be loaded as needed for computation. In our benchmark experiments, this engine outperforms existing web3 native geographic data solutions in terms of speed and accuracy (minimum (10^18)x gain in terms of resolution due to the efficient division of space by quadtrees).

We will integrate this GIS engine with Kolektivo’s smart contracts in order to store and query verifiable time-space stamps. This will be done in several steps.

1) All Kolektivo data will be divided into essential and non-essential categories with respect to onchain storage. Essential data that needs to be on the chain may include spatiotemporal data such as geographic polygons, points, and possibly trajectories of dynamically updating objects

(if any). Non-essential data includes attributes that are too expensive to store on chain, such as weather data, or a picture or a video at a particular time.

2) All spatiotemporal data will be converted into a form that is suitable for the Quadtree data structure. This requires building a custom data ingestion engine.

3) A custom dapp will upload the essential data on chain and non-essential data on IPFS.

4) A Kolektivo-specific query engine will be built that can be integrated with the DAO contracts, so that DAO contracts can invoke computation and queries on demand.

The data stored in this manner is verifiable (since it is stored on a verifiable ledger), each computation is trusted and verifiable (since they are assigned to native EVMs as opposed to a central server), and query speed is superior to current solutions such as FOAM's Crypto-Spatial coordinates.

Additionally, the GIS engine opens up opportunities for doing other sophisticated calculations on chain and on demand. For example, when lands are associated with success/failure tags and a DAO's token supply would contract and expand based on the 'success' land's area, the GIS engine can step in to calculate the area quickly.

Benchmark Experiments

We have done some benchmarks to figure out which parts of the engine should be on the chain.

TODO: Describe the benchmark experiment: restaurant locations example.

Link to the sheet

The attributes of selected datasets for experiments is given below.

Reference Dataset

a. North America Lakes, which consists of 1199 Polygons and 31207 Points. GeoJSON file size is 3.4MB.
b. Minor Island, which consists of 2796 Polygons and 35519 Points. GeoJSON file size is 4.1MB
c. Admin 1 State Province Lakes which consists of 57 Polygons and 8502 Points. File size is 1.4MB

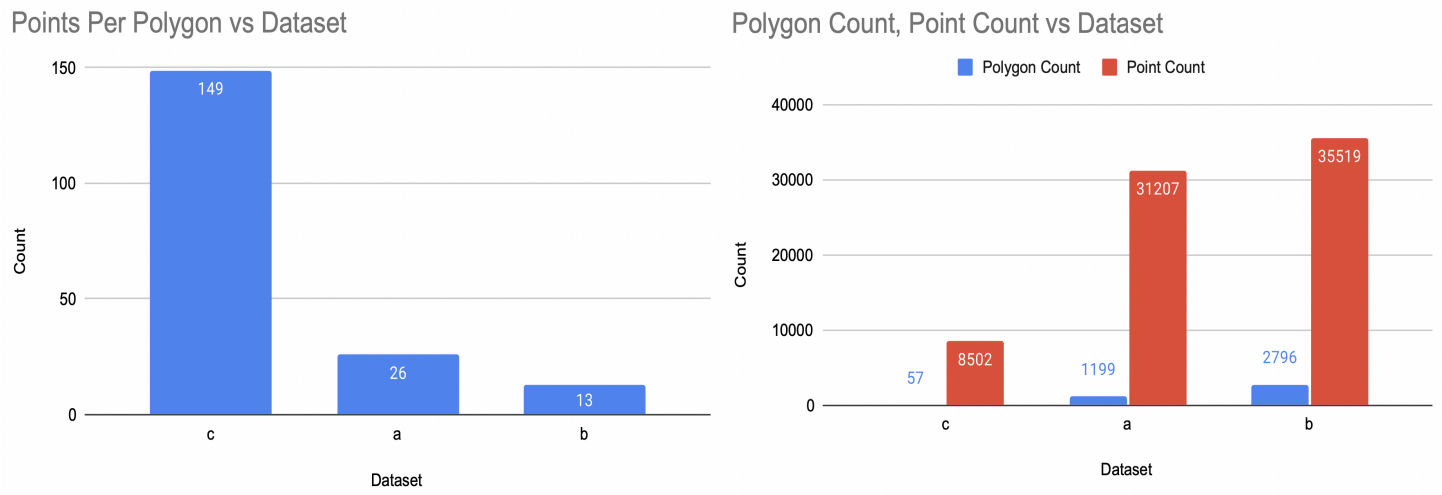


Figure 1: (a) Points per Polygon vs Dataset (b) Polygon Count, Point Count vs Dataset

Problem statement

1. **Point Query:** Finding a specific restaurant from the list of restaurants' coordinates.
2. **Range Query:** Finding restaurants around a square of width 1 unit side.

Configuration

- Constructed Quadtree has a maximum depth of 10 and 100 points per quad capacity.

Web2 Performance

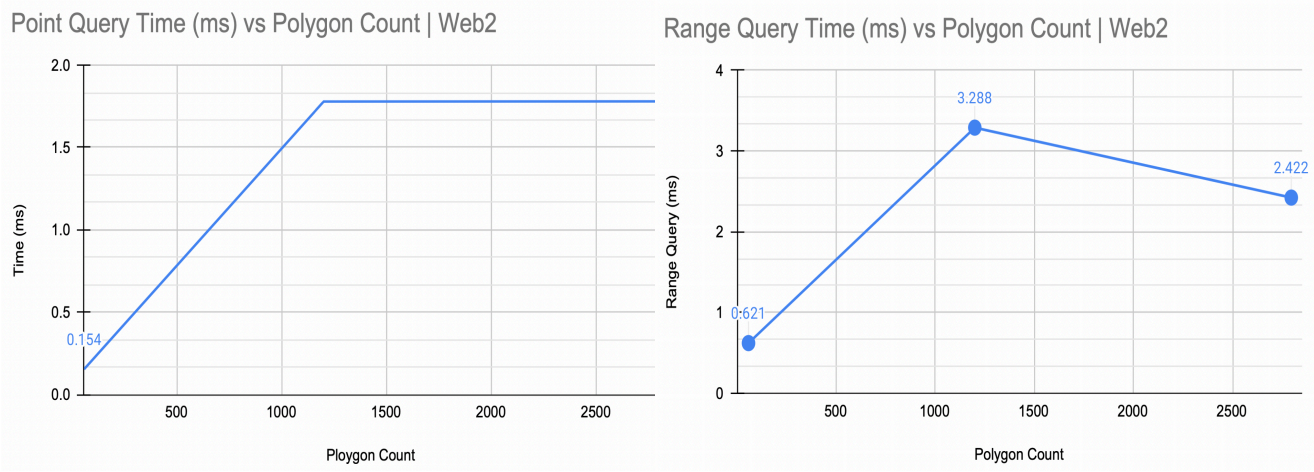


Figure 2: Web2 Performance (a) Point Query (b) Range Query

Web3 Performance in local testnet

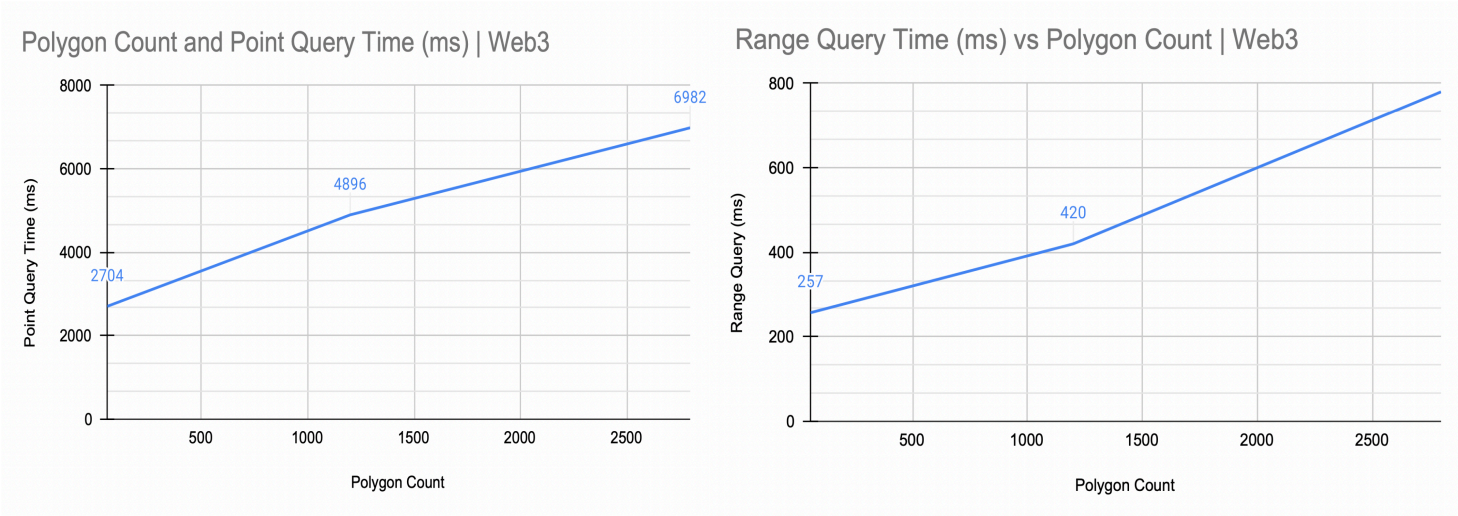
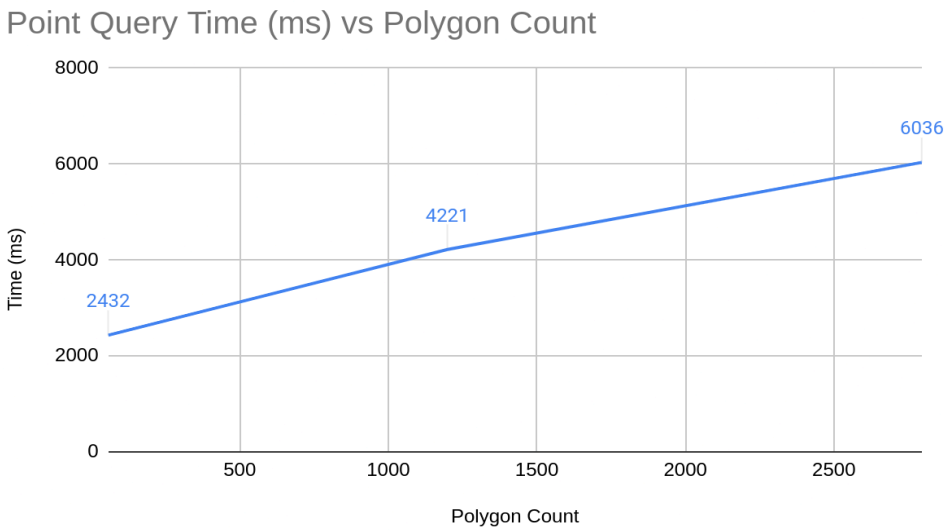


Figure 3: Web3 Performance (a) Point Query (b) Range Query

Note: In a different test where the precision range for the query with 57 polygons was +- 5 units, it took 6058 ms to return the results, i.e the function returned nodes that were within 5 units of the provided coordinate. The leafnode of the dataset was customized by appending numerous child nodes in order to observe how much time it takes for a tiny 5 unit range.

Web3 Performance in Alfajores testnet



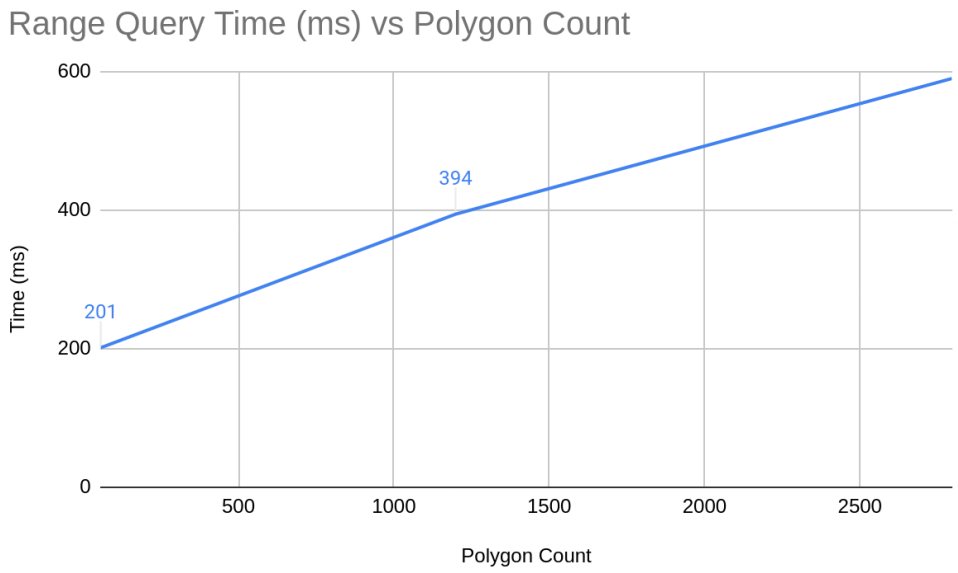
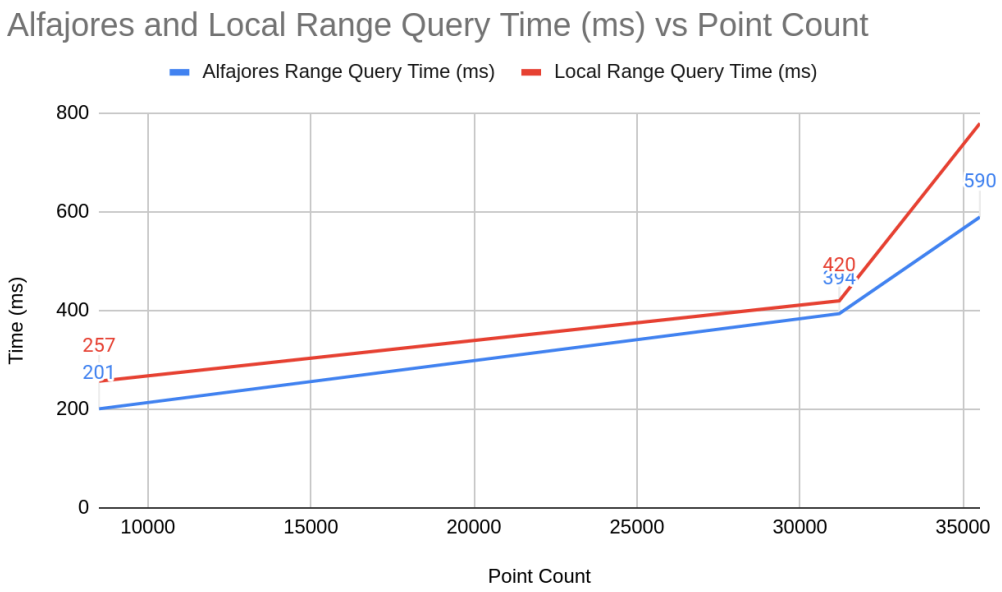


Figure 4:Web3 Performance in Alfajores testnet (a) Point Query (b) Range Query

Observations:

Query execution time in Alfajores testnet is faster compared to local blockchain.



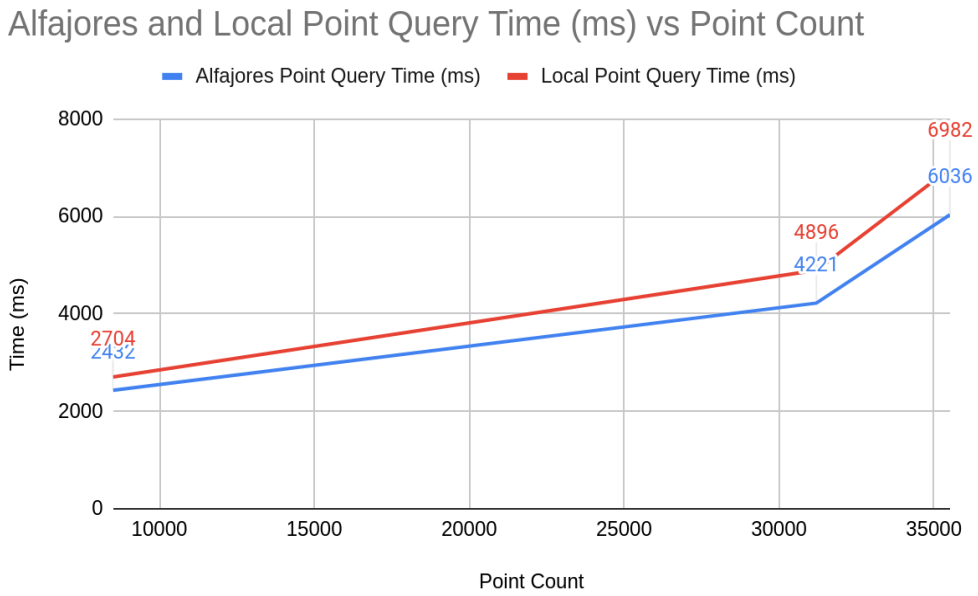


Figure 5: Alfajores Testnet vs Local Blockchain (a) Range Query (b) Point Query

Performance Comparison Between Quadtree and FOAM Implementations

Point Query

Point query in Foam protocol implementation is fast because of its use of smart hashing using a grid's latitude and longitude. Whenever an input is given for a point query the algorithm communicates it to the nearest hash that matches with any of the grid's index. Then it is checked whether the point exists within that grid.

Range Query

Range query in Foam protocol implementation is a linear search using loops since there's no tree like structure or relationship between the grids and it is not possible to keep track of the neighbor grids unless it is stored as an additional data inside the grid. This makes it highly inefficient to search for ranges.

Note: Both the queries in Quadtree as well as Foam incur 0 gas fee as they are "view" functions with read-only access.

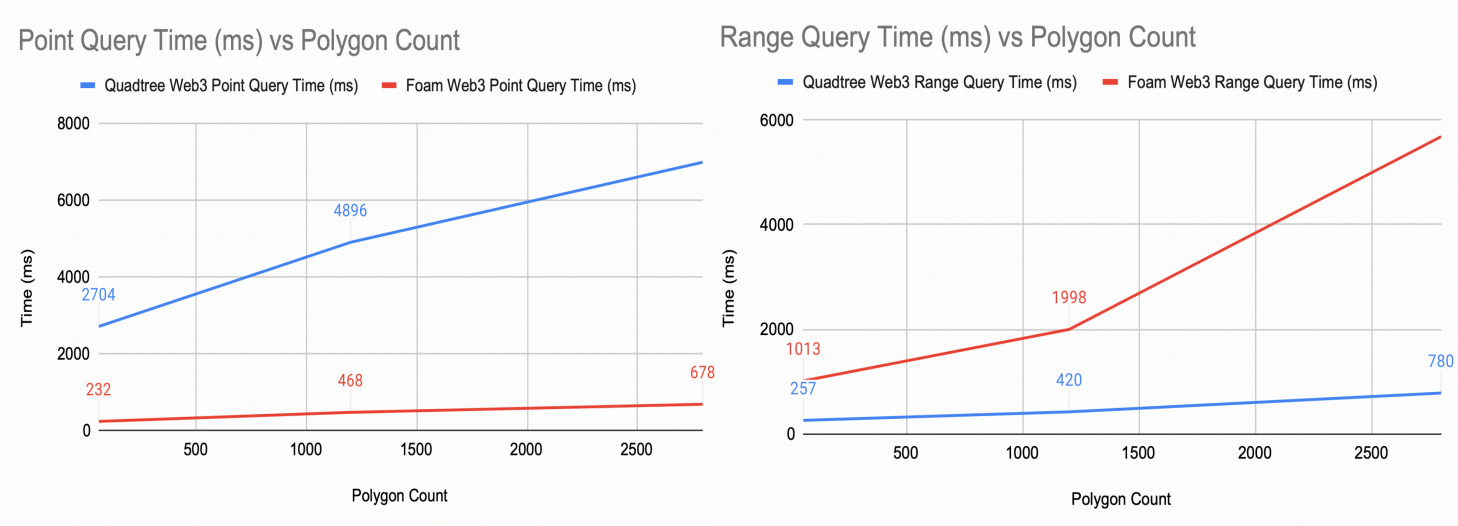


Figure 5: Quadtree vs Foam Implementation (a) Point Query (b) Range Query

Point insertion execution time and required gas cost with respect to number of points

Total inserted points: 16244
Execution time: 1491.562674s = 24.85 minutes.
Total Gas cost in Gwei: 229134431.8
Total cost in USD: $768.63

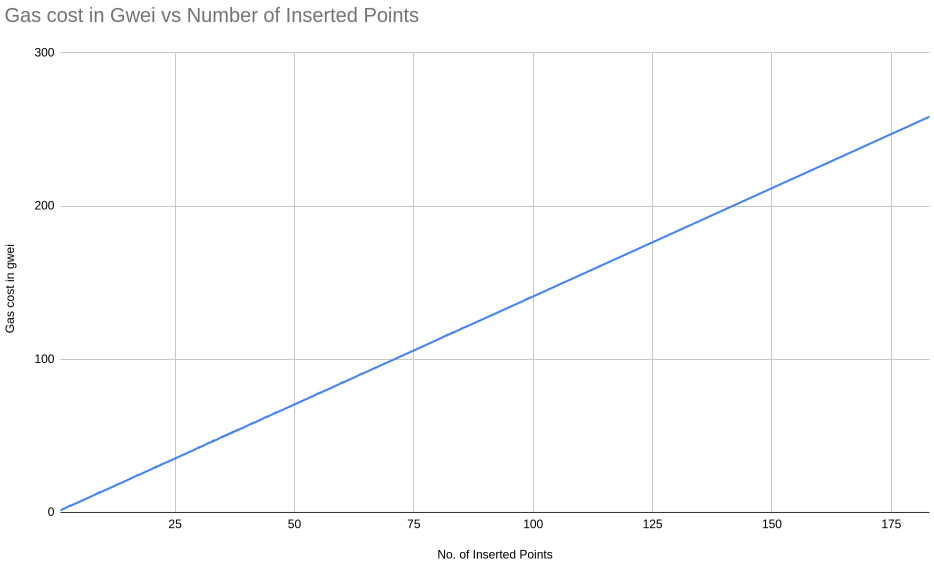


Figure 5: Gas Cost of Point Insertion

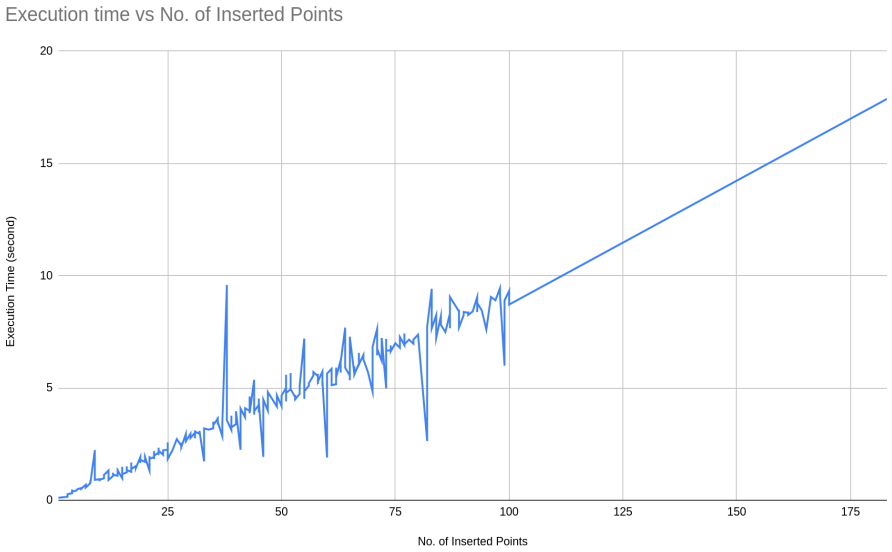


Figure 6: Point Insertion Execution Time

Point insertion in Alfajores testnet

Total time to insert: 36872.75088 s
Total point inserted: 9,240
Total Gas Fee in USD: $207.92
On average to insert 87 points, the gas fee in USD will be $1.95.

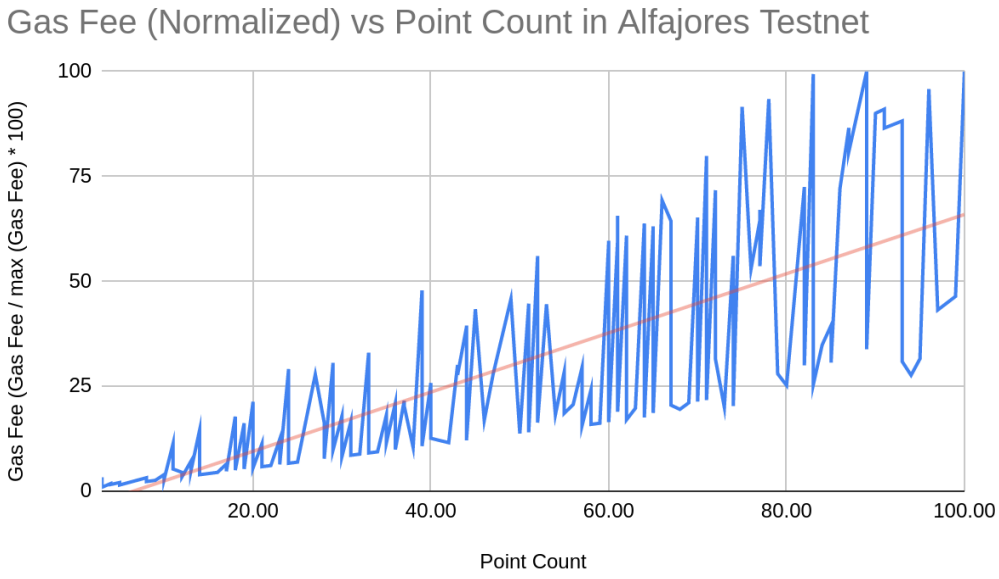


Figure 7: Gas Cost of Point Insertion

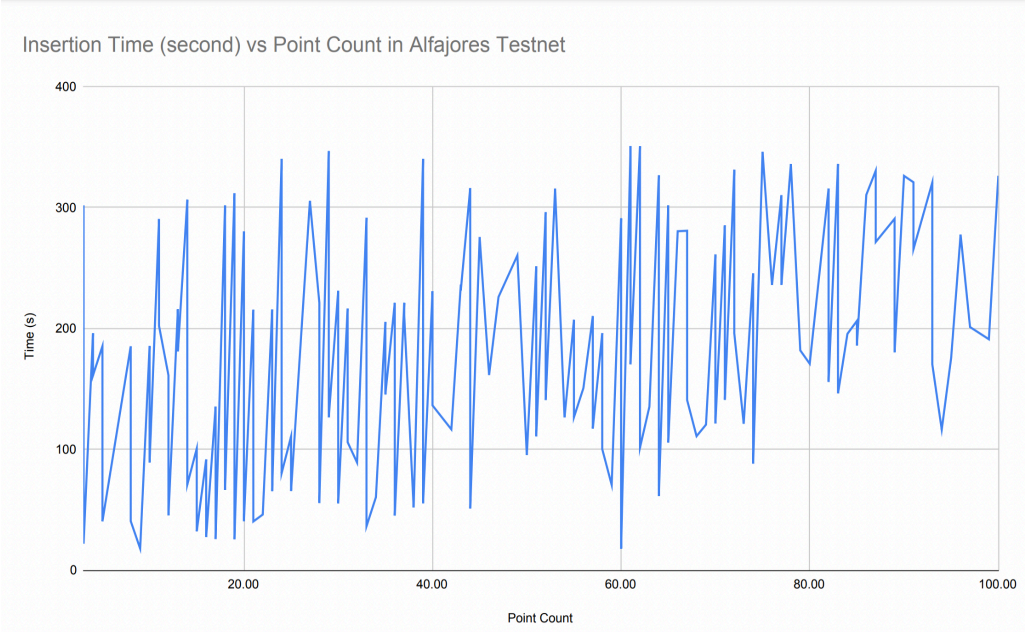


Figure 8: Time required to Insert Point

Observations

Point insertion cost is cheaper in Alfajores testnet compared to local blockchain. Inserting each point in local blockchain costs $0.0472789953 and in Alfajores testnet costs $0.0225021645; which is about half the cost of the former.

Integration of our data structures with Kolektivo's data

Problem statement: We need to find a restaurant's daily earnings (i.e., a scalar data attribute associated with each point of interest).

Solution 1: Each node of a quadtree is stored as a "struct" data structure in the blockchain which contains several data types such as the node's boundary, children node indexes, points it contains as well as an IPFS hash. This hash will correspond to an IPFS file address which can store relevant data.

For finding a restaurant's daily earnings (assuming the earnings is stored in the IPFS), the following steps have to be performed:

1. The node containing the restaurant has to be located in the quadtree
2. The IPFS hash stored has to be obtained from that node
3. Using the retrieved IPFS hash, the earnings for a particular date for the restaurant has to be fetched from the IPFS file

For (1), a point query will be performed which will have O(logN) time complexity and the time taken to find the point will depend on the depth of the quadtree and the number of points it contains. For (2), a simple read operation will be performed on the node (the index of the node

is retrieved through point query in (1)) to retrieve the IPFS CID. In (3) After getting the IPFS CID, Corresponding data of that particular point will be fetched from IPFS.

Solution 2: The limitation of solution1 is updating relevant data of a point will change the IPFS hash. So, the hash must be updated in the blockchain. It consumes blockchain storage, time and most importantly transaction cost for updating the hash time to time.

Ceramic network will be used for storing data. A StreamID is an immutable identifier for a stream in the Ceramic network. This streamID will be stored instead of IPFS hash.

Solution 3: OrbitDB is a distributed, peer-to-peer database which uses IPFS data storage and its pubsub (publish — subscribe) protocol to automatically sync up with other peers. Here, OrbitDB will be used to store all relevant data e.g. Latitude, Longitude, Restaurant Name, Address, Dates, Earning, Manu, Service etc.

Latitude	Longitude	Restaurant Name	Date	Income
52.189	12.009	RioCoffee	14.04.2022	$1200
53.146	13.897	Cheese Mania	14.04.2022	$1456
54.001	13.989	Cielo	14.04.2022	$2020

Table: Restaurant Database in OrbitDB

To find a restaurant's daily earnings, the steps that will be performed :

1. A point query of latitude and longitude will be performed to get daily earnings from OrbitDB. The time complexity to retrieve data from the OrbitDB database is O(N-entries).

Decentralized Storage

IPFS Performance

The reading operations latency is getting worse as the data size increases.

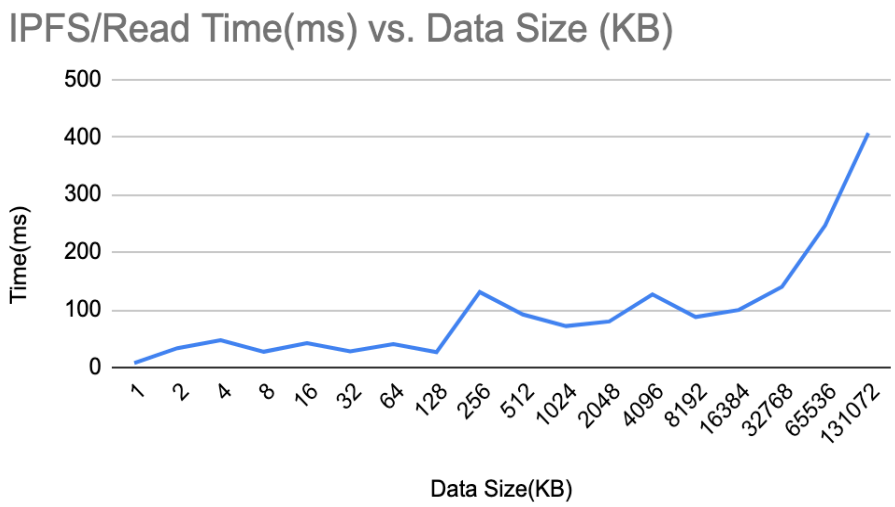


Figure 9: IPFS Data Loading Performance

Ceramic Performance

Data size more than 256KB exceeds the maximum block size of ceramic. Because of smaller data sizes, the latency of fetching data is approximately constant. Figure 11 shows the data loading performance of ceramic.

OrbitDB Performance

A peer-to-peer database for storing indexed documents of three fields (Name, Email and Phone) is used to measure the performance. As the phone number is unique, a query function of the phone number is called to get a value or entry from the database. Here's the execution time to get all relevant data of a query with an increasing number of data entries.

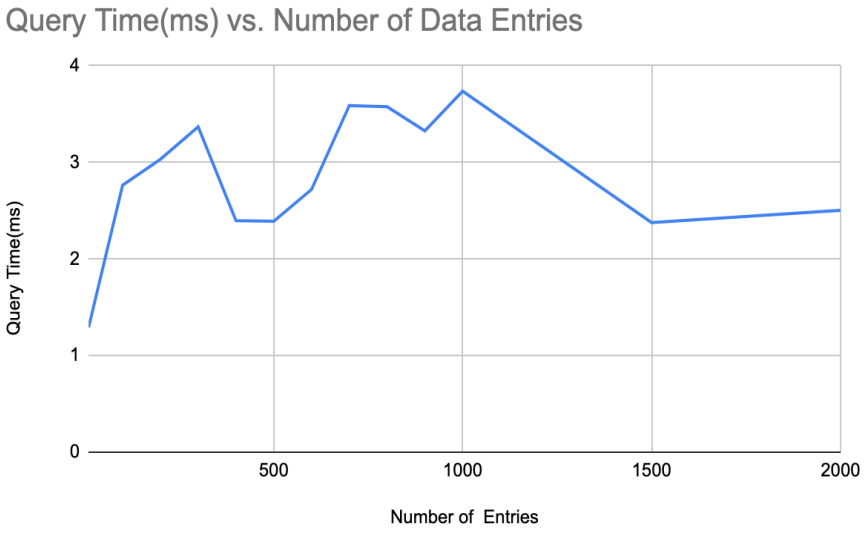


Figure 10: OrbitDB Performance

Comparison

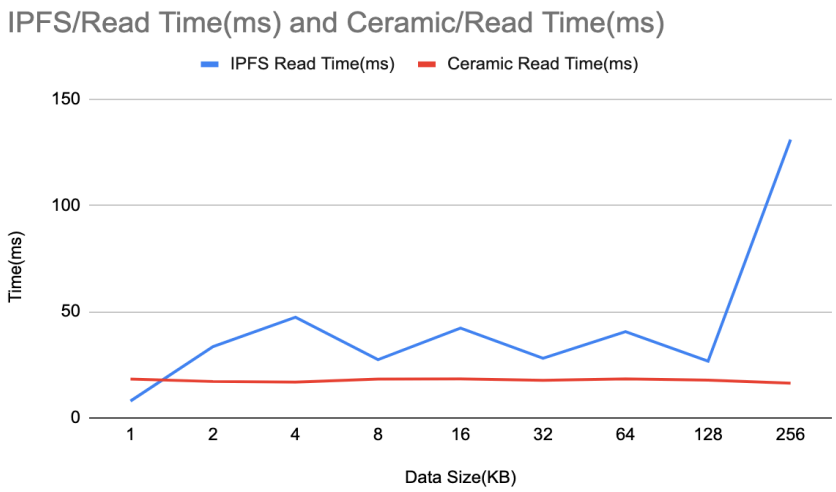


Figure 11: IPFS vs Ceramic

Fetching Data from IPFS to Blockchain using Provable Oracle Service:

- IPFS Test File Links:
    a. Generic large data file: https://ipfs.io/ipfs/QmZgsvrA1o1C8BGCrx6mHTqR1Ui1XqbCrtbMVrRLHtuPVD?filename=big-api-response.json
    b. Geospatial data file: https://ipfs.io/ipfs/QmXpyiwutCFXmHtvQaTMwST2cvFkWpxj7n2C4JDPU1FsmG
    c. https://ipfs.io/ipfs/Qmd4PvCKbFbbB8krxajCSeHdLXQamdt7yFxFxzTbedwiYM

Data Size	Execution Time	Transaction Cost (gas)
13B	14789 ms	156589
26B	22911 ms	156167
192B	16895 ms	156613

The execution time of fetching data from IPFS to Blockchain using Provable Oracle service doesn't depend on data size Only.

It depends on:

1. A data source type
2. Query type
3. Network type

Directions found for the data structure from the benchmarks

1. Tree based data structure: Complex searches require a hierarchical relationship, which is highly efficient in tree based structures. These search requirements cannot be fulfilled by hashmap based data structures such as Foam protocol.
2. Customized Point region (PR) Quadtree: This is a bit similar to what Foam protocol implements for indexing except for the fact that in our PR quadtrees we can incorporate the boundaries in the index. This will reduce the time in lookups and reduce the number of calculations required for tree traversal, making queries produce results even faster.

Kolektivo's Requirements

Technical capability wise, Kolektivo requires the following GIS functionality.

1) Point query: given a point's coordinates, find which quad (or region) it belongs to.
2) Range query: retrieve all points within a general range (region, quad, etc.).
3) Calculate area of a few polygons that are tagged 'good' or 'bad'
    a) This requires an edited version of the range query.
    b) First, find all 'good' polygons or points in the queried area.
        i) First, do a range query
        ii) Then filter all 'good' points or polygons
    c) Calculate the combined area of the convex polygon created by all such polygons or points.
        i) In a quadtree format, it's much easier to calculate the area by the sum of quads at a particular resolution that's considered high.
    d) This combined quad area for the convex polygon can serve as a scalar value that signals the contraction and expansion of the token supply.

Observations

- Point query: FOAM outperforms quadtrees.

- Range Query: Quadtree outperforms FOAM.
- Token supply contraction and expansion: since range queries are needed, quadtrees will be the definite solution here.
- Gas fees
    - We have designed point and range queries so that no gas fees are incurred.
    - Points and nodes insertion requires a one time cost. The total cost for inserting a 9k points set is approximately $202.
- Decisions
    - Range queries seem to be more useful for Kolektivo's needs. So we should stick to quadtrees.
    - Alfajores testnet performs