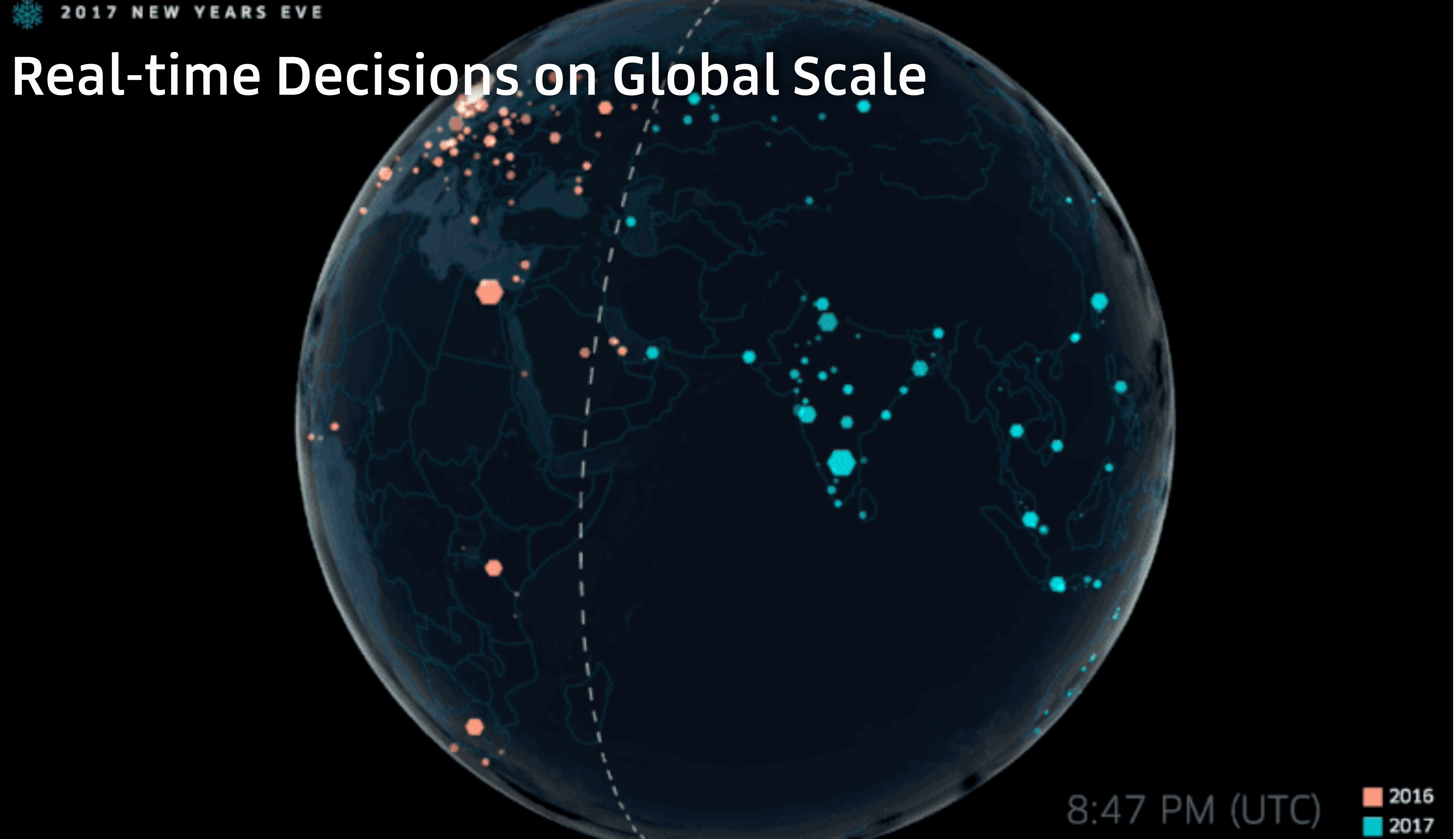
UBER

# Scaling Uber's Elasticsearch as an Geo-Temporal Database

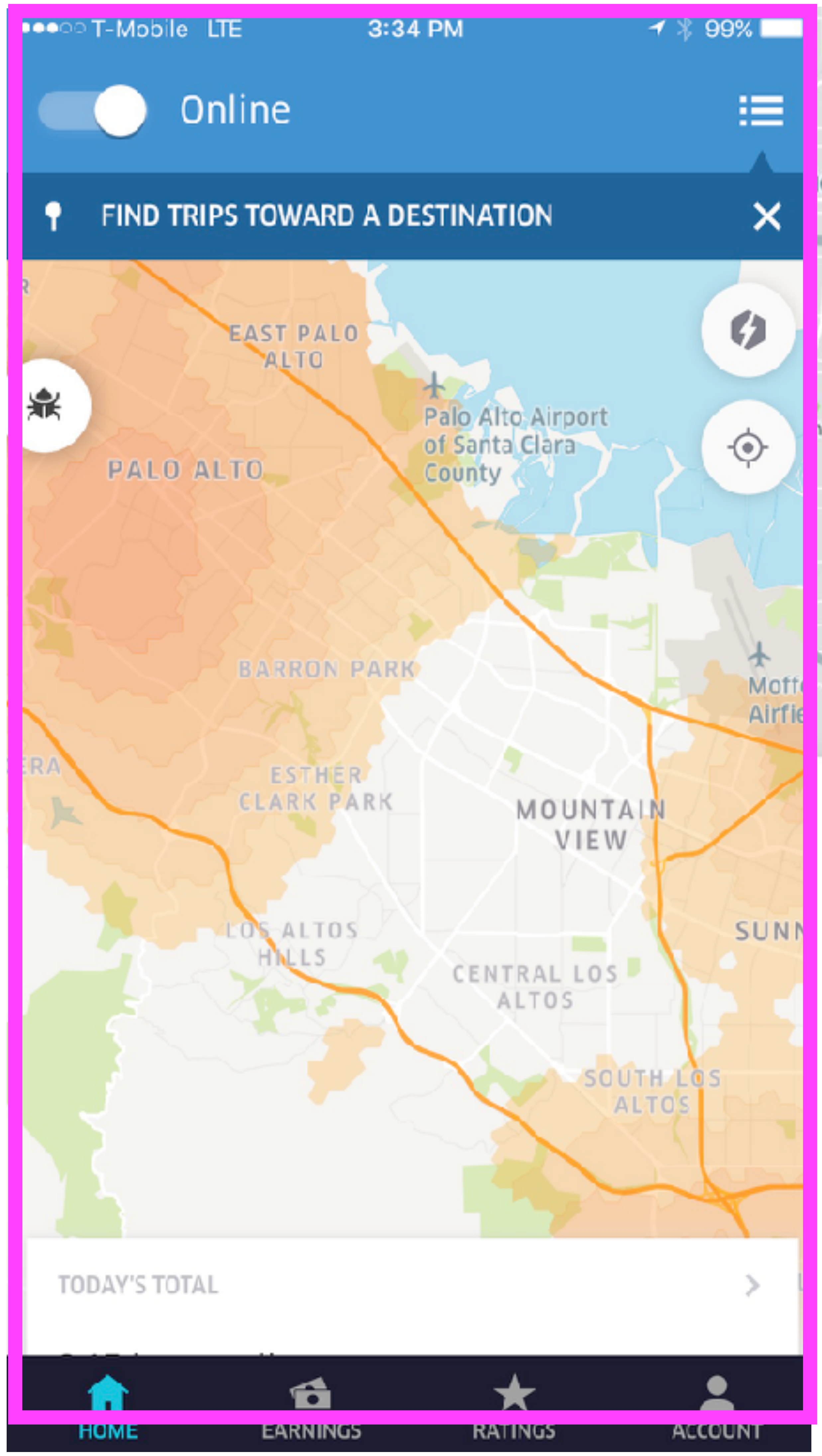
Danny Yuan @ Uber

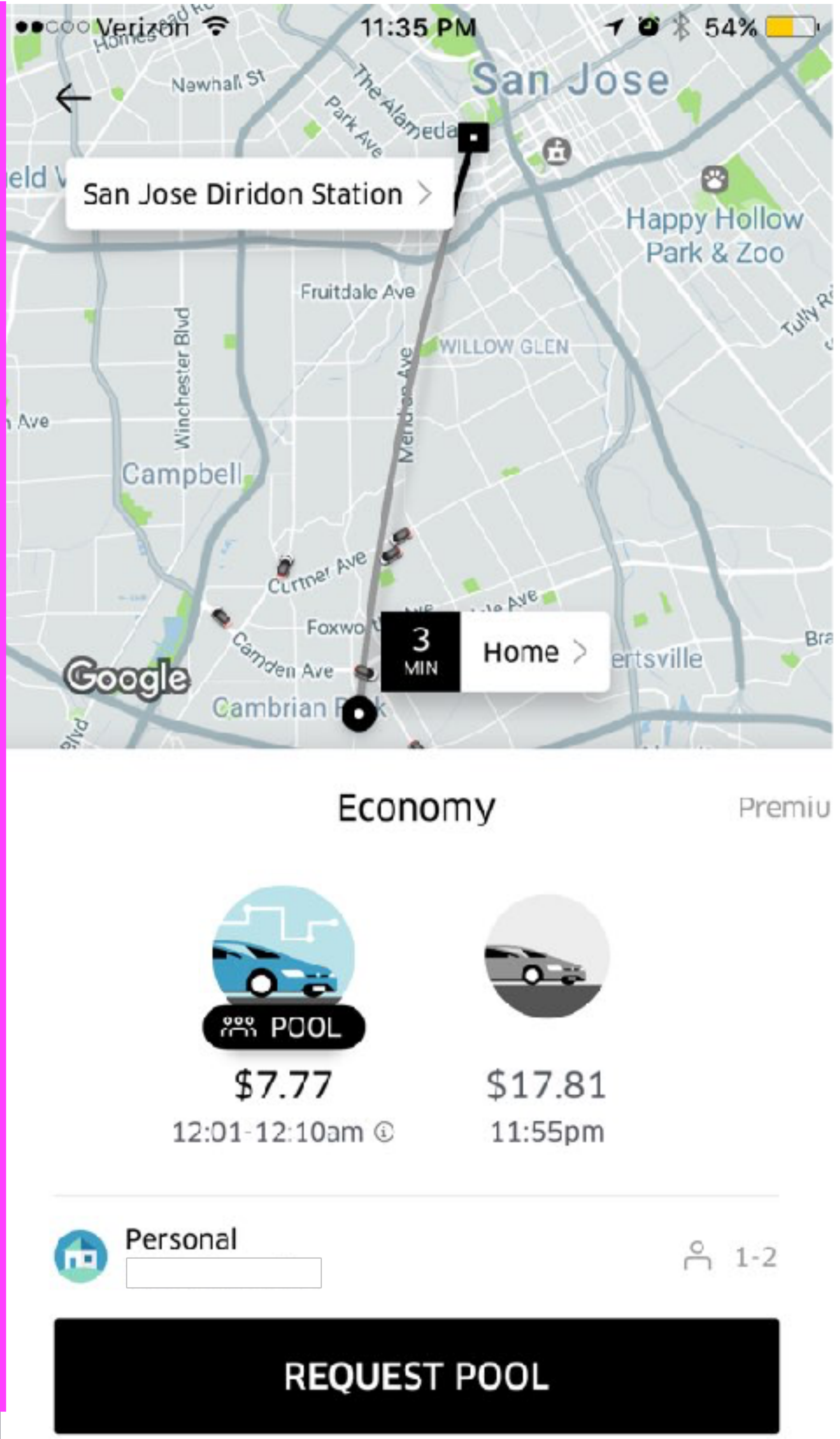
# Use Cases for a Geo-Temporal Database

# Real-time Decisions on Global Scale

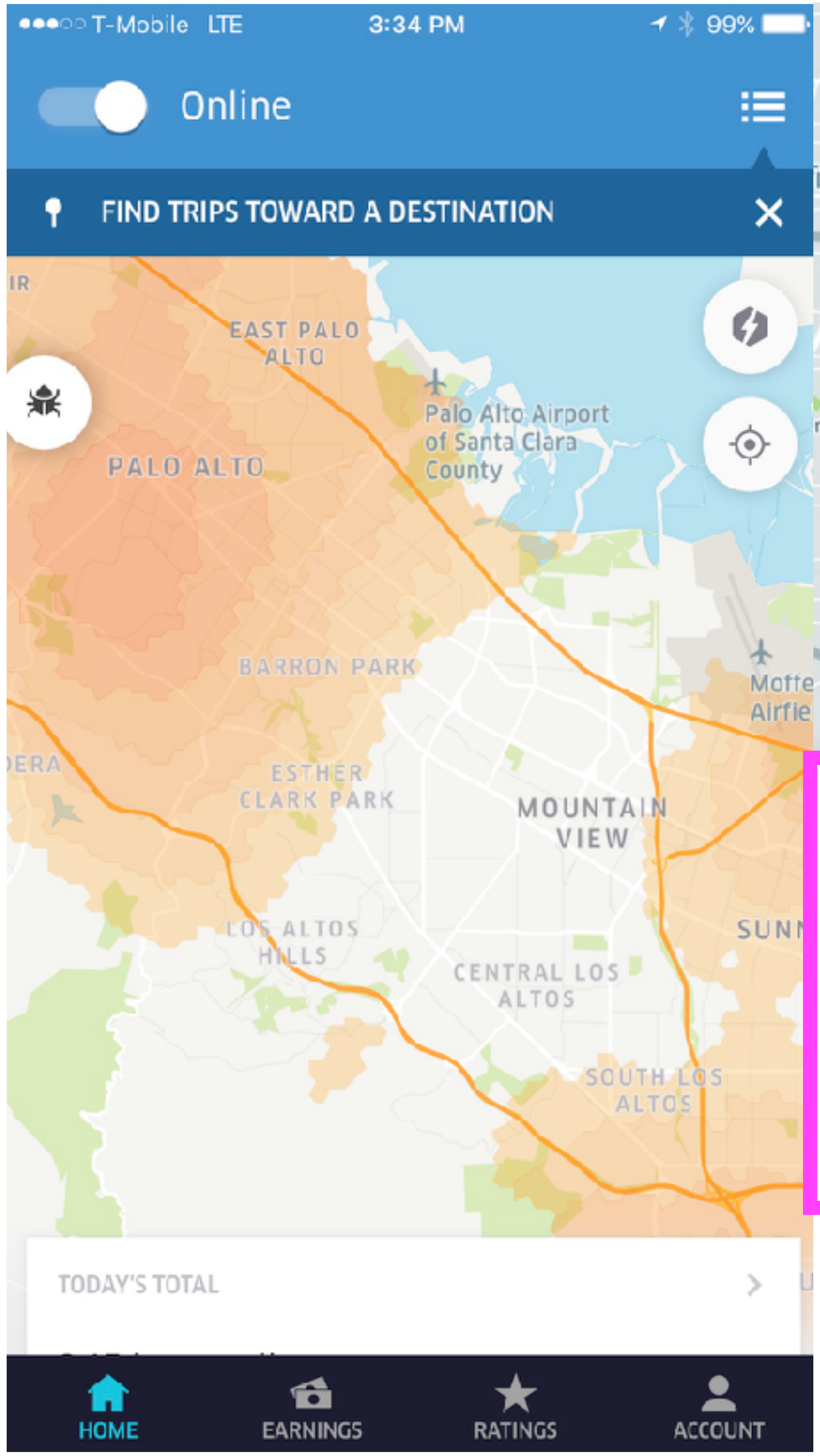


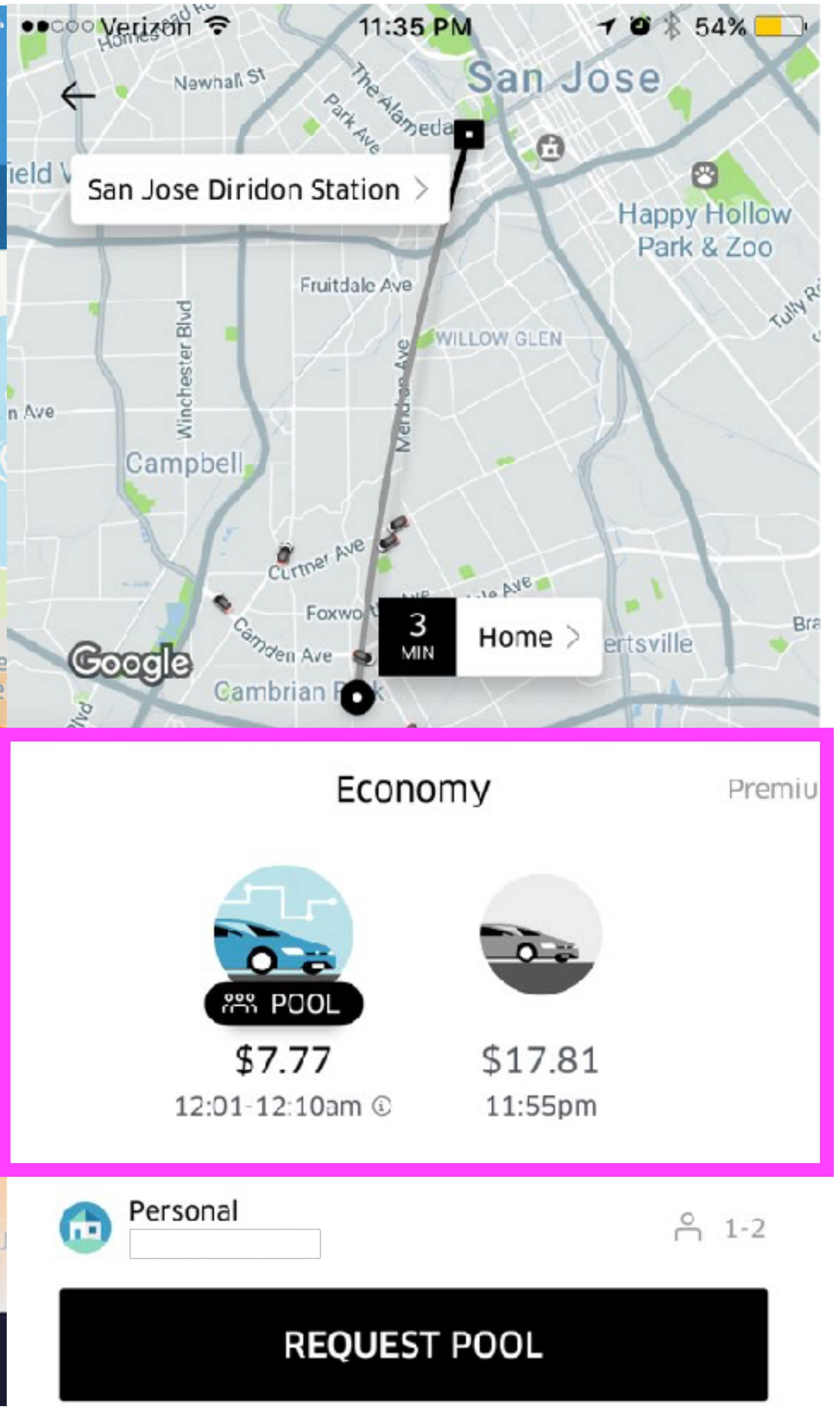
# Dynamic Pricing: Every Hexagon, Every Minute



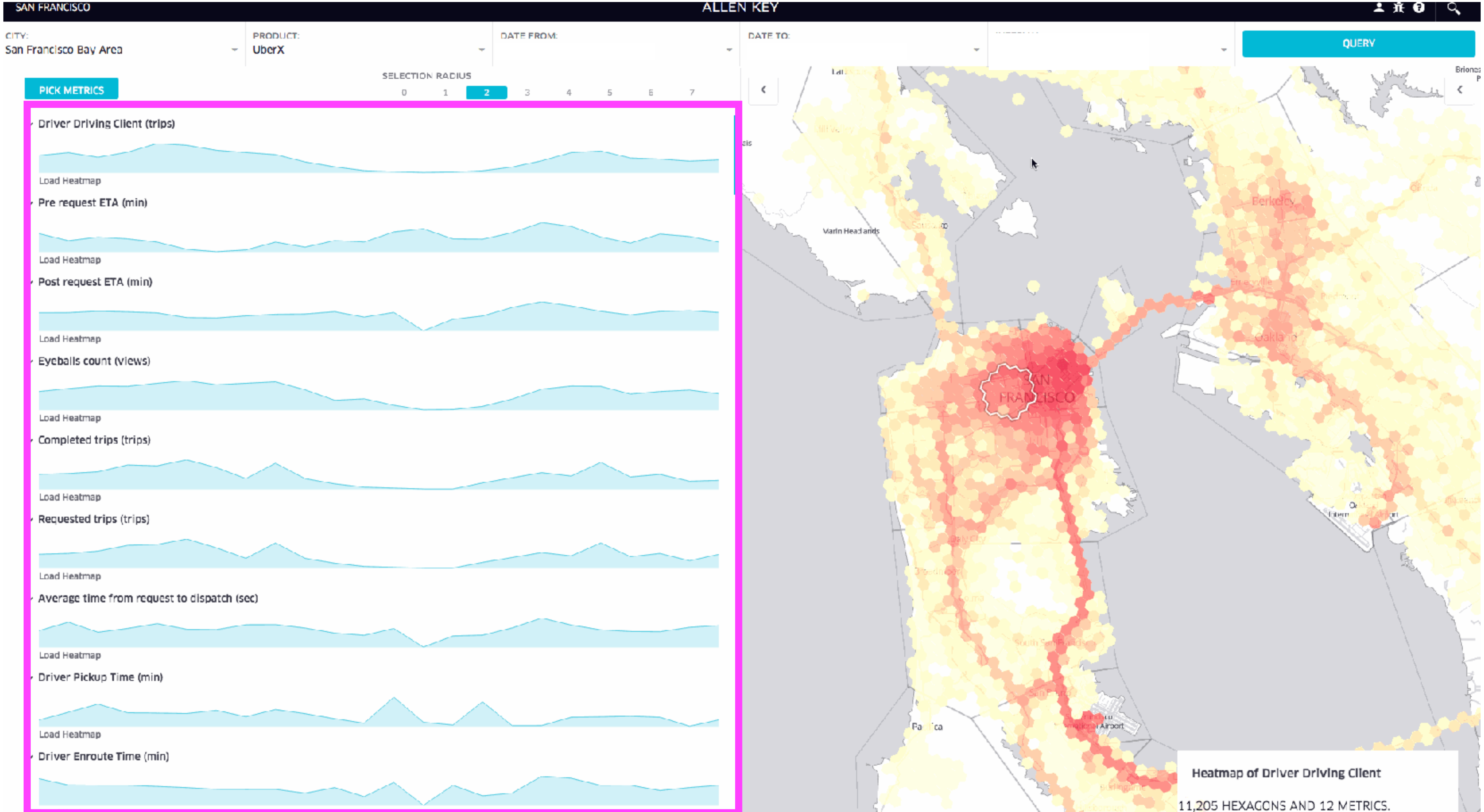



# Dynamic Pricing: Every Hexagon, Every Minute

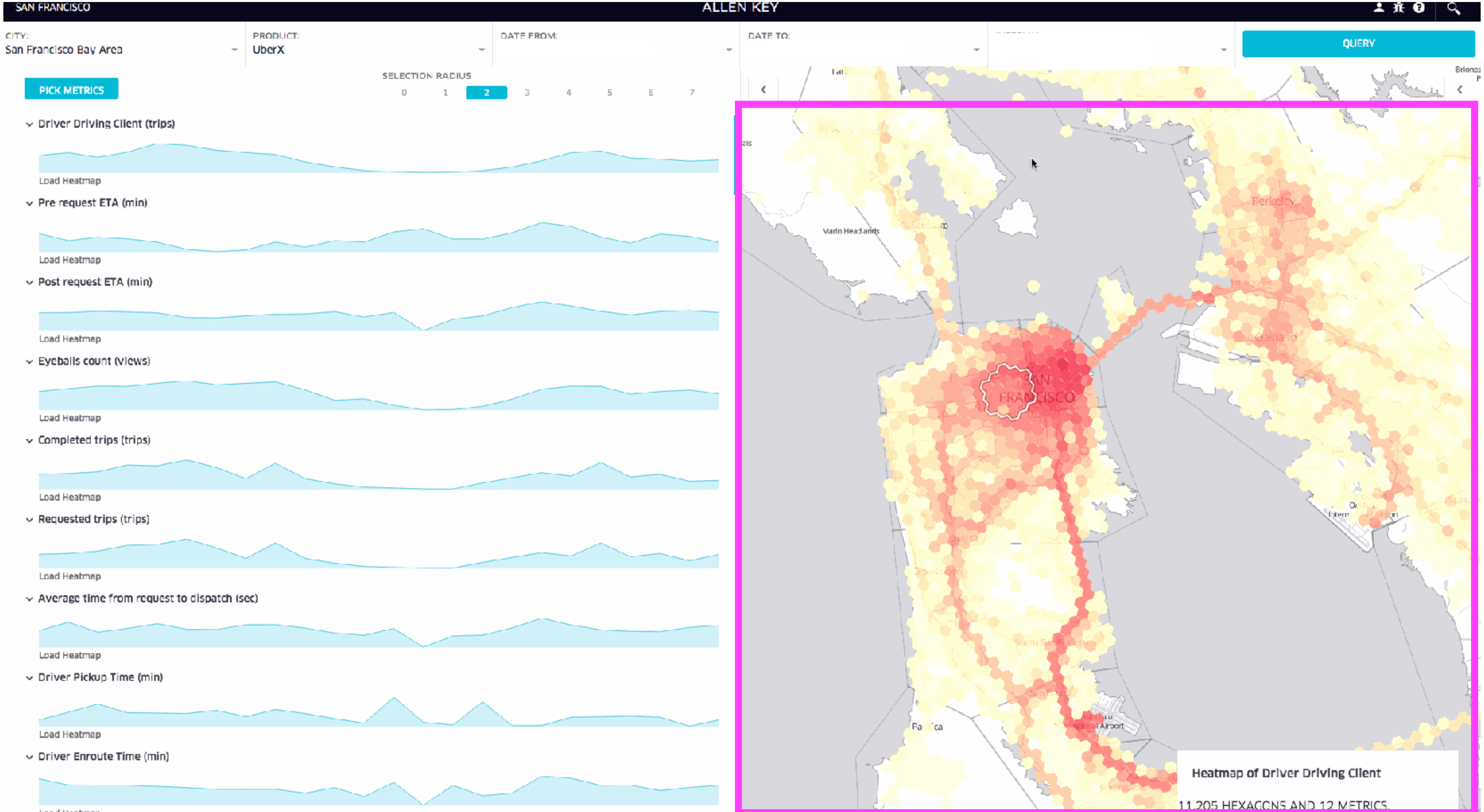


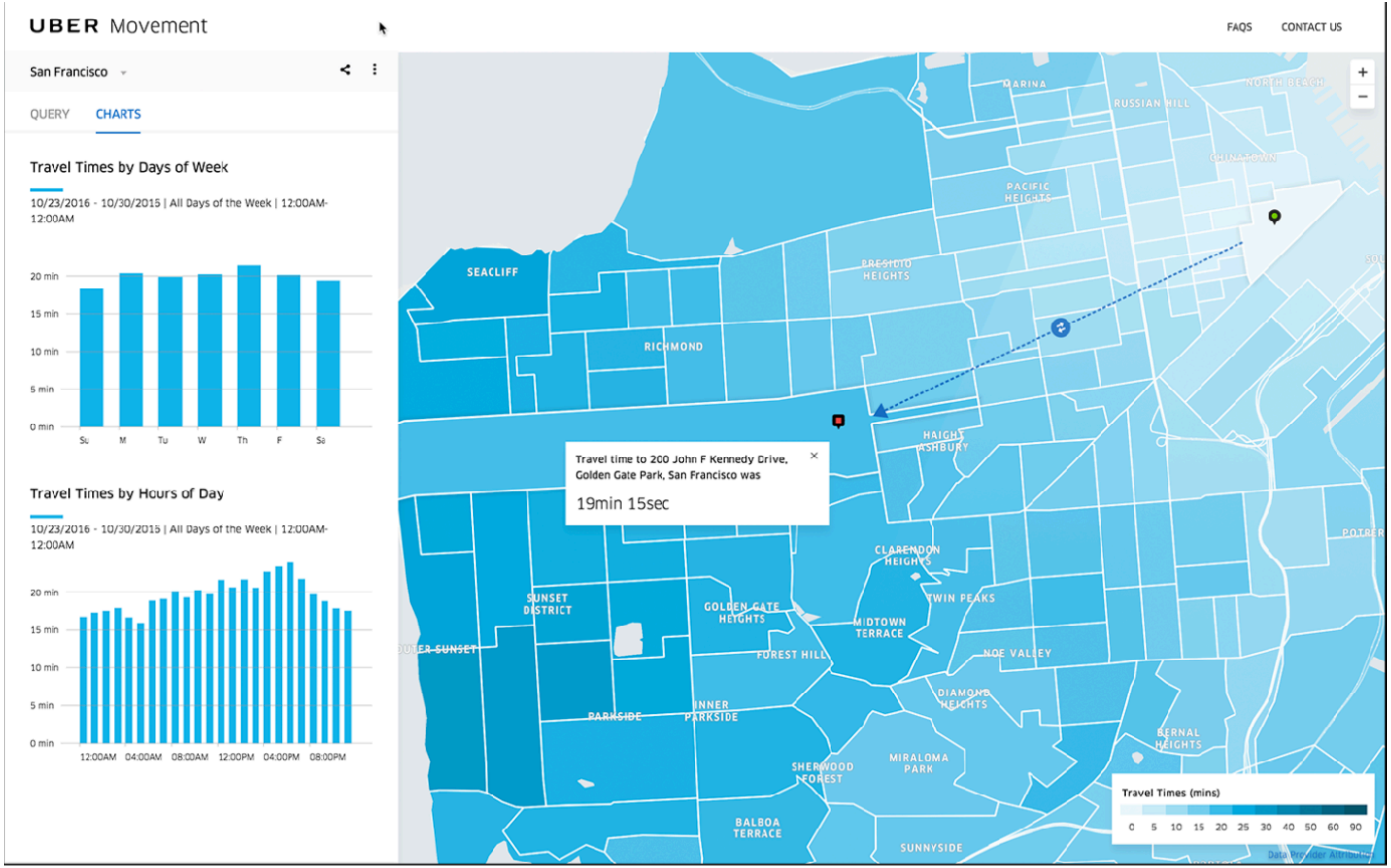
# **Metrics**: how many UberXs were in a trip in the past 10 minutes



# **Metrics**: how many UberXs were in a trip in the past 10 minutes



# Market Analysis: Travel Times



# **Forecasting**: Granular Forecasting of Rider Demand

# How Can We Produce Geo-Temporal Data for Ever Changing Business Needs?

# Key Question: What Is the Right Abstraction?

# **Abstraction**: Single-Table OLAP on Geo-Temporal Data

# **Abstraction**: Single-Table OLAP on Geo-Temporal Data

```
SELECT   <agg functions>, <dimensions>
FROM     <data_source>
WHERE    <boolean filter>
GROUP BY <dimensions>
HAVING   <boolean filter>
ORDER BY <sorting criterial>
LIMIT    <n>
```

# **Abstraction**: Single-Table OLAP on Geo-Temporal Data

```
SELECT   <agg functions>, <dimensions>
FROM     <data source>
WHERE    <boolean filter>
GROUP BY <dimensions>
HAVING   <boolean filter>
ORDER BY <sorting criterial>
LIMIT    <n>
```

# Why Elasticsearch?

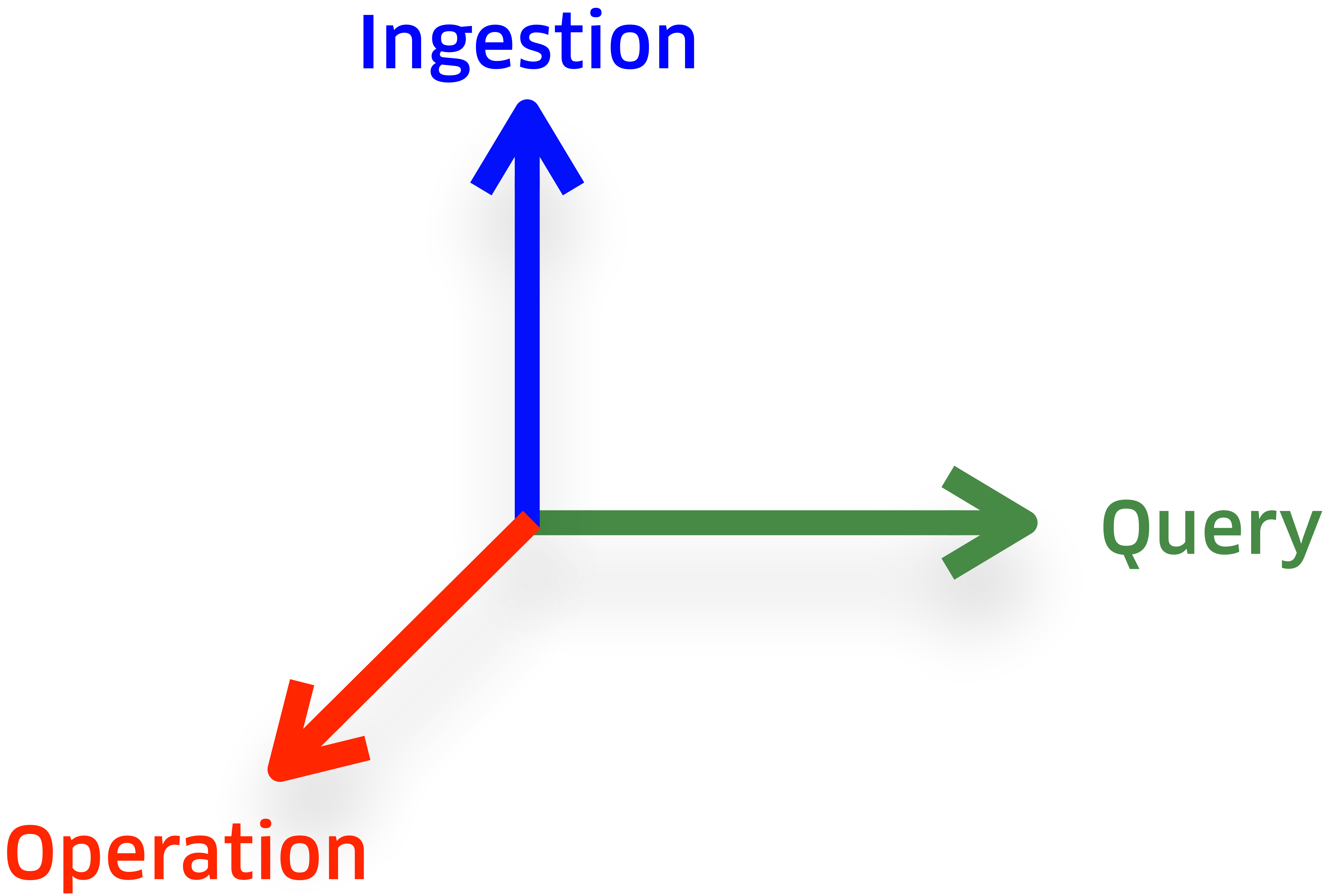
- Arbitrary boolean query
- Sub-second response time
- Built-in distributed aggregation functions
- High-cardinality queries
- Idempotent insertion to deduplicate data
- Second-level data freshness
- Scales with data volume
- Operable by small team

# Current Scale: An Important Context

- **Ingestion**: 850K to 1.3M messages/second
- **Ingestion volume**: 12TB / day
- **Doc scans**: 100M to 4B docs/ second
- **Data size**:  1 PB
- **Cluster size**: 700 ElasticSearch Machines
- **Ingestion pipeline**: 100+ Data Pipeline Jobs

# Our Story of Scaling Elasticsearch

# Three Dimensions of Scale



# Driving Principles

- Optimize for fast iteration
- Optimize for simple operations
- Optimize for automation and tools
- Optimize for being reasonably fast

# The Past: We Started Small

# Constraints for Being Small

- Three-person team
- Two data centers
- Small set of requirements: common analytics for machines

# First Order of Business: Take Care of the Basics

# Get Single-Node Right: Follow the 20-80 Rule

- One table <–> multiple indices by time range
- Disable `_source` field
- Disable `_all` field
- Use `doc_values` for storage
- Disable analyzed field
- Tune JVM parameters

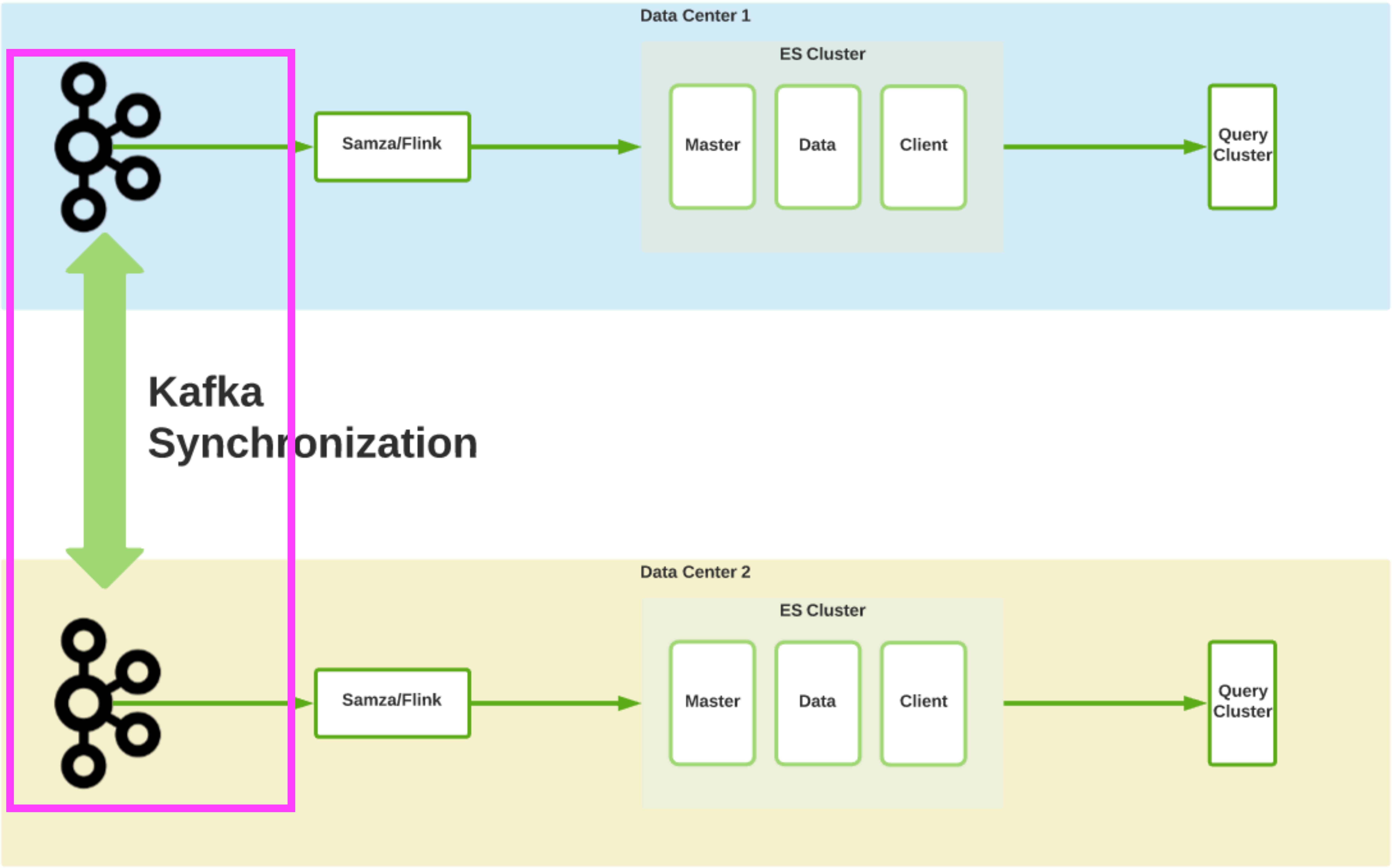
# Make Decisions with Numbers

- What's the maximum number of recovery threads?
- What's the maximum size of request queue?
- What should the refresh rate be?
- How many shards should an index have?
- What's the throttling threshold?
- **Solution:** Set up end-to-end stress testing framework

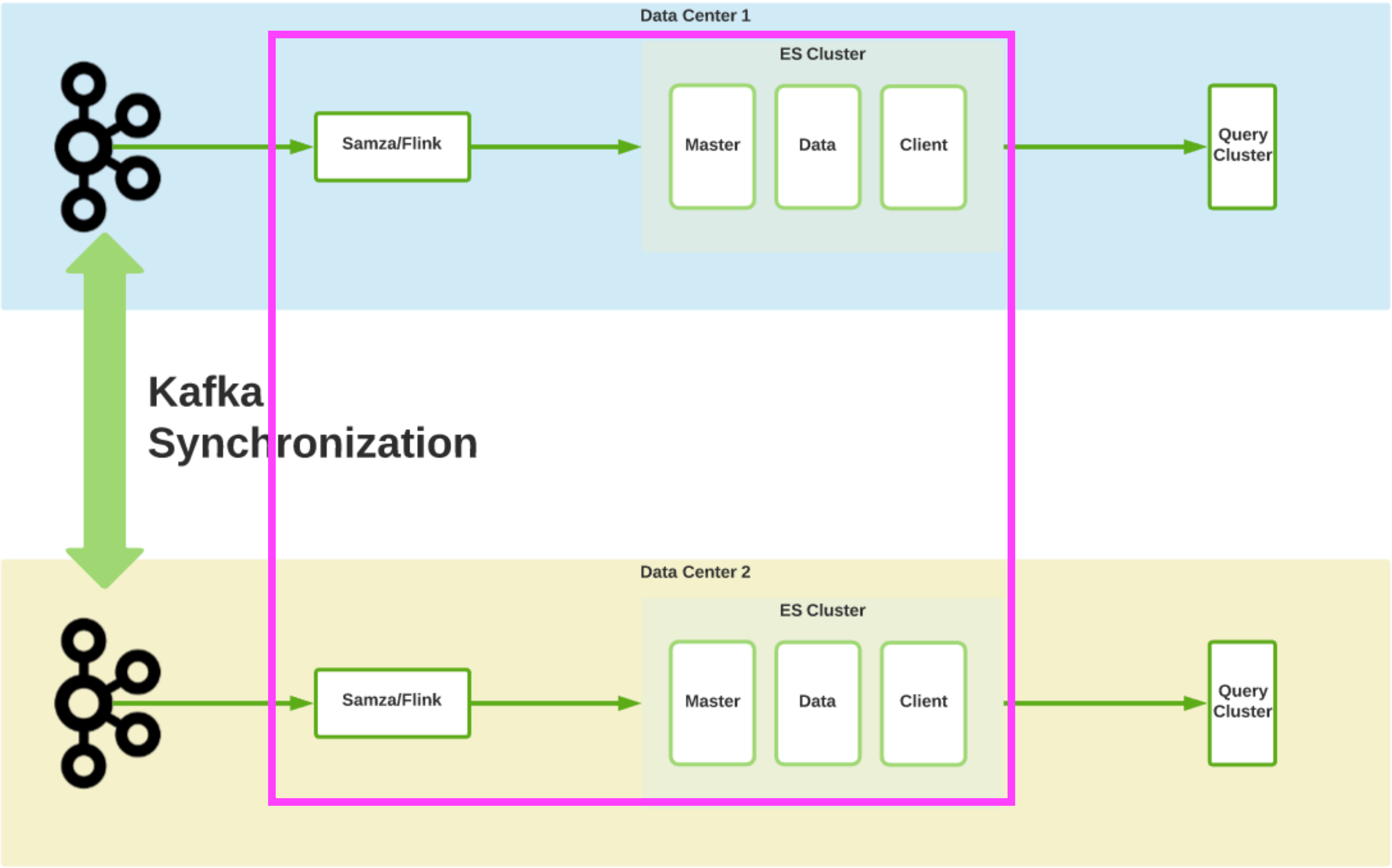
# Deployment in Two Data Centers

- Each data center has exclusive set of cities
- Should tolerate failure of a single data center
  - Ingestion should continue to work
  - Querying any city should return correct results

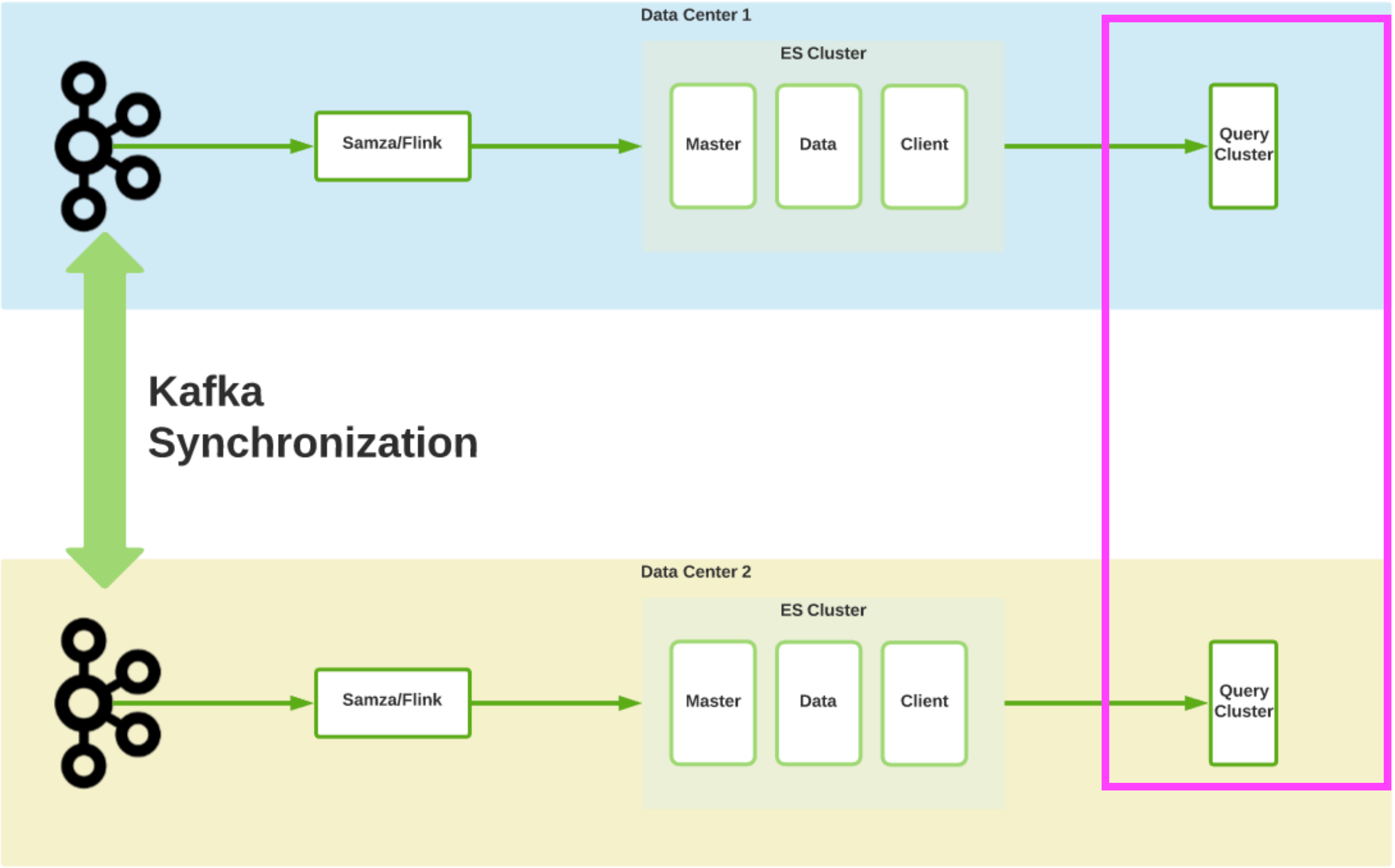
# Deployment in Two Data Centers: trade space for availability



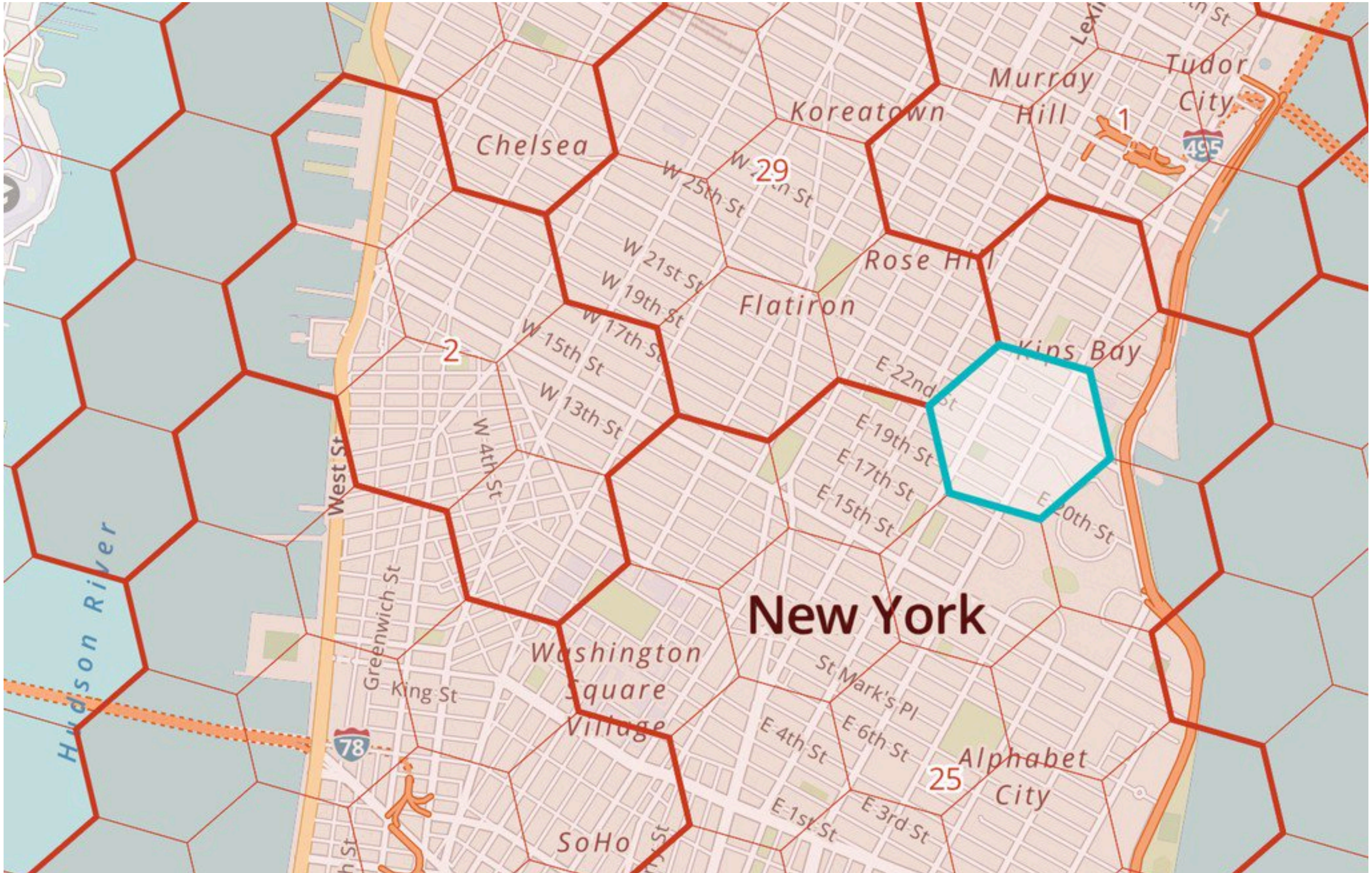
# Deployment in Two Data Centers: trade space for availability



# Deployment in Two Data Centers: trade space for availability

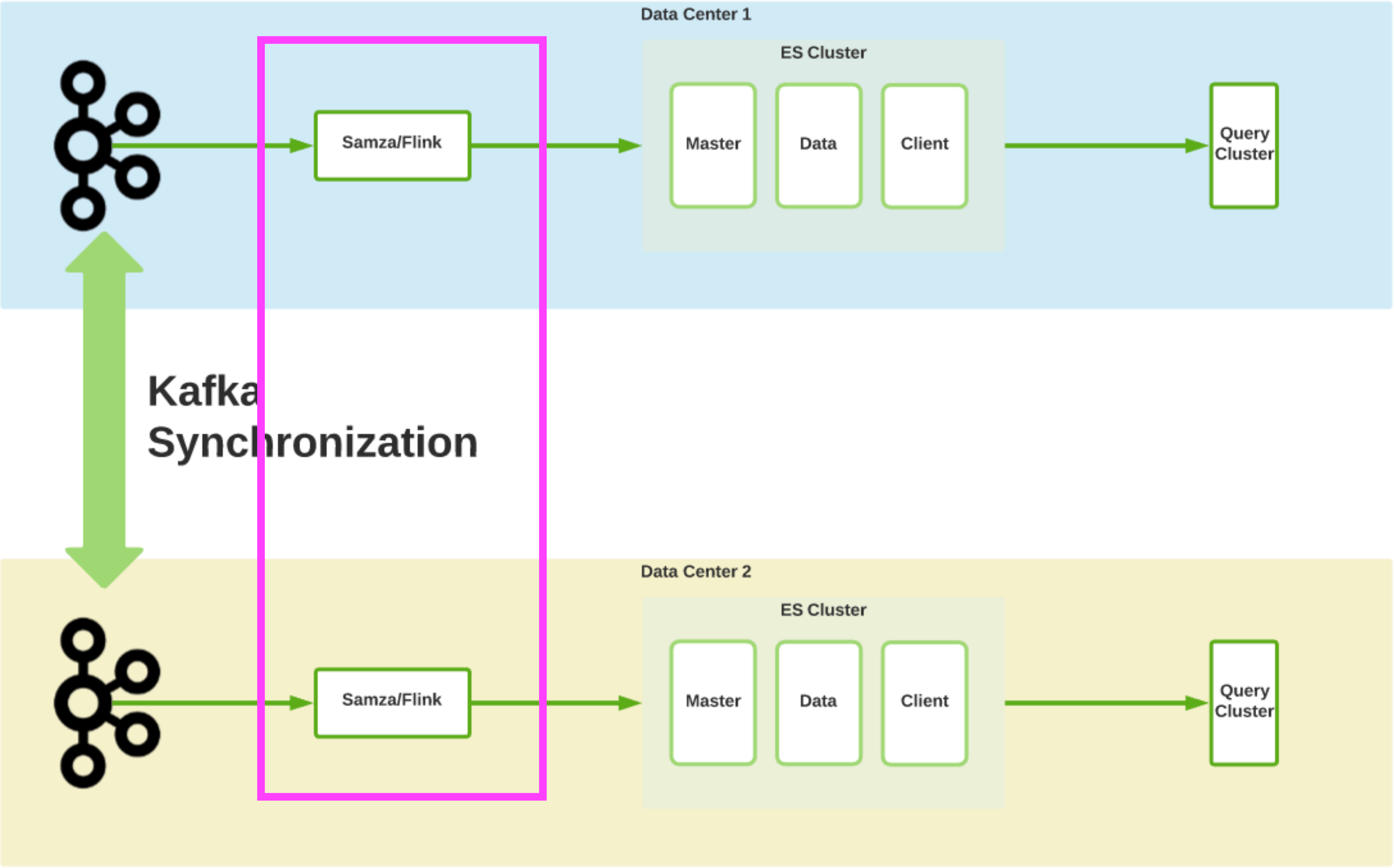


# Discretize Geo Locations: H3



# Optimizations to Ingestion

# Optimizations to Ingestion



# Dealing with Large Volume of Data

- An event source produces more than 3TB every day
- **Key insight**: human does not need too granular data
- **Key insight**: stream data usually has lots of redundancy

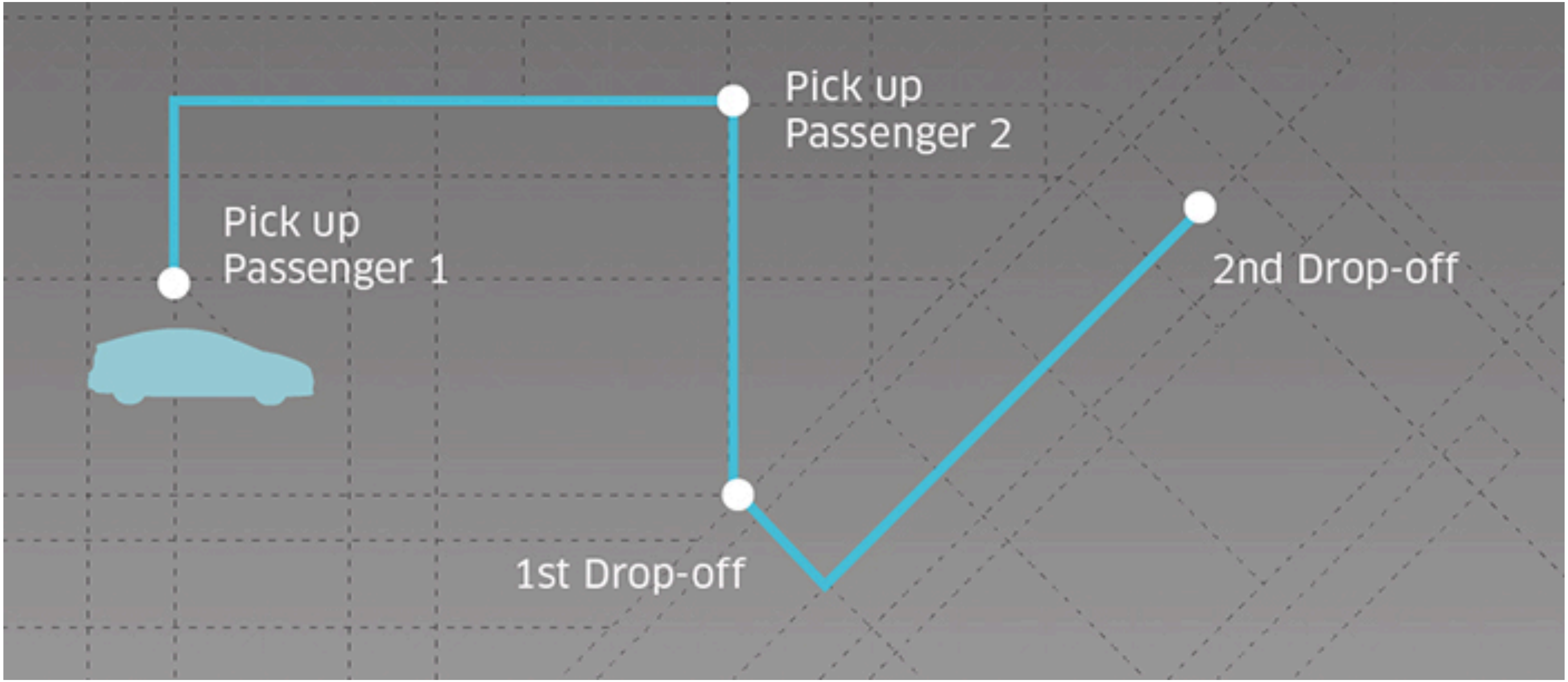
# Dealing with Large Volume of Data

- Pruning unnecessary fields
- Devise algorithms to remove redundancy
- 3TB –> 42 GB, more than 70x of reduction!
- Bulk write

# Data Modeling Matters

# Example: Efficient and Reliable Join

- **Example**: Calculate Completed/Requested ratio with two different event streams

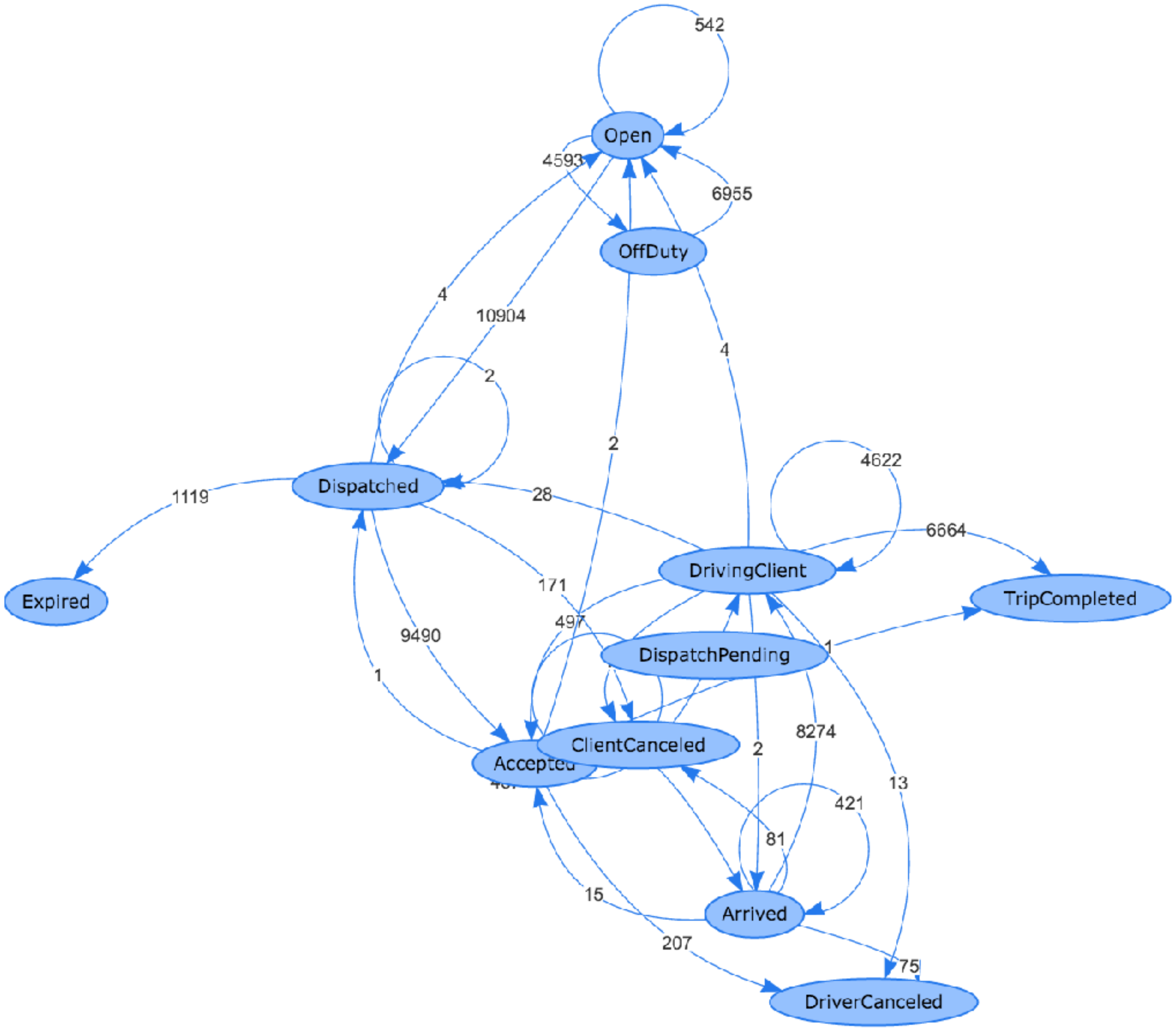


# Example: Efficient and Reliable Join: Use Elasticsearch

- Calculate Completed/Requested ratio from two Kafka topics
  - Can we use streaming join?
  - Can we join on the query side?
  - **Solution**: rendezvous at Elasticsearch on trip ID

| TripID | Pickup Time | Completed |
|---|---|---|
| 1 | 2018-02-03T... | TRUE |
| 2 | 2018-02-3T... | FALSE |
| | | |
| | | |

# Example: aggregation on state transitions

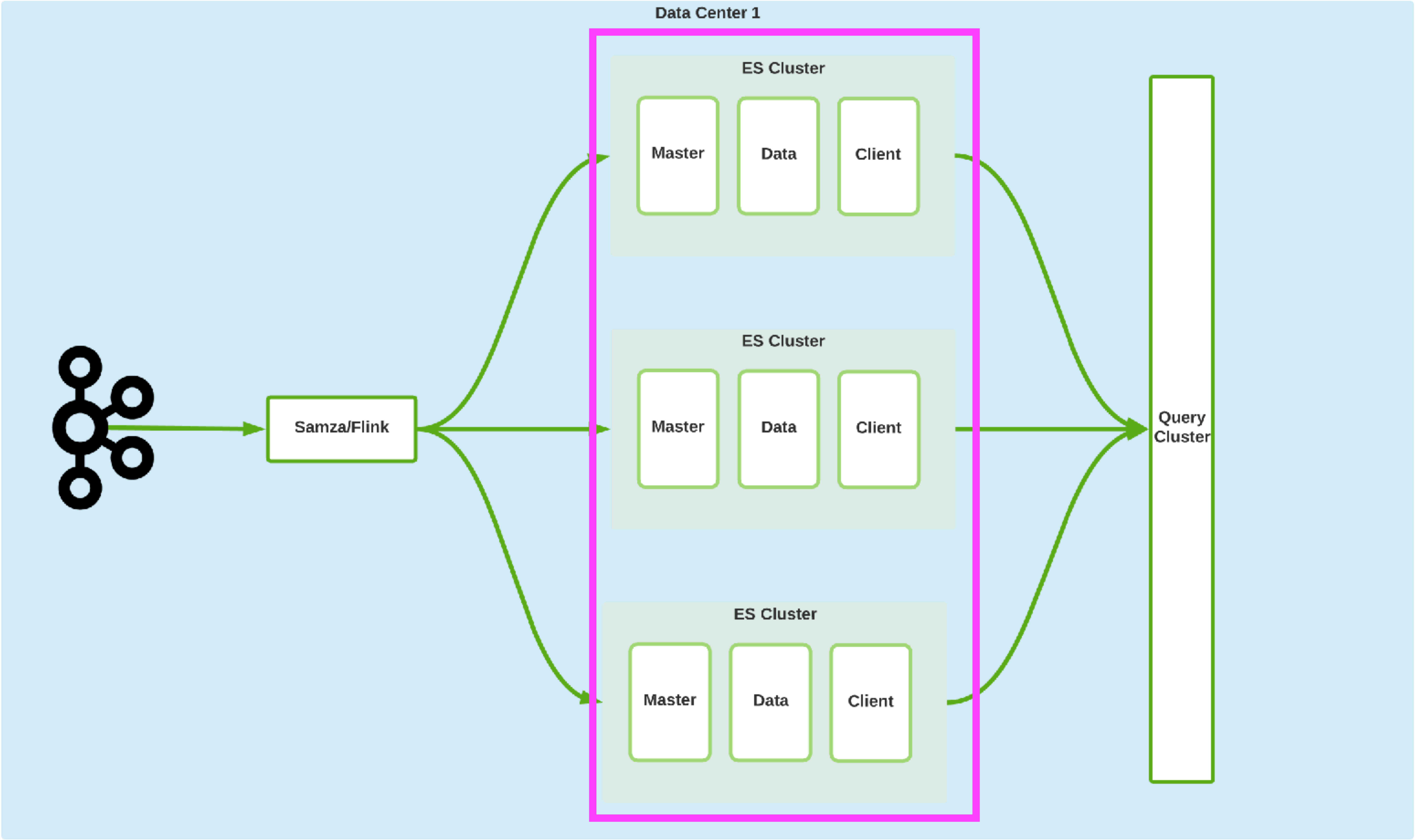


# Optimize Querying Elasticsearch

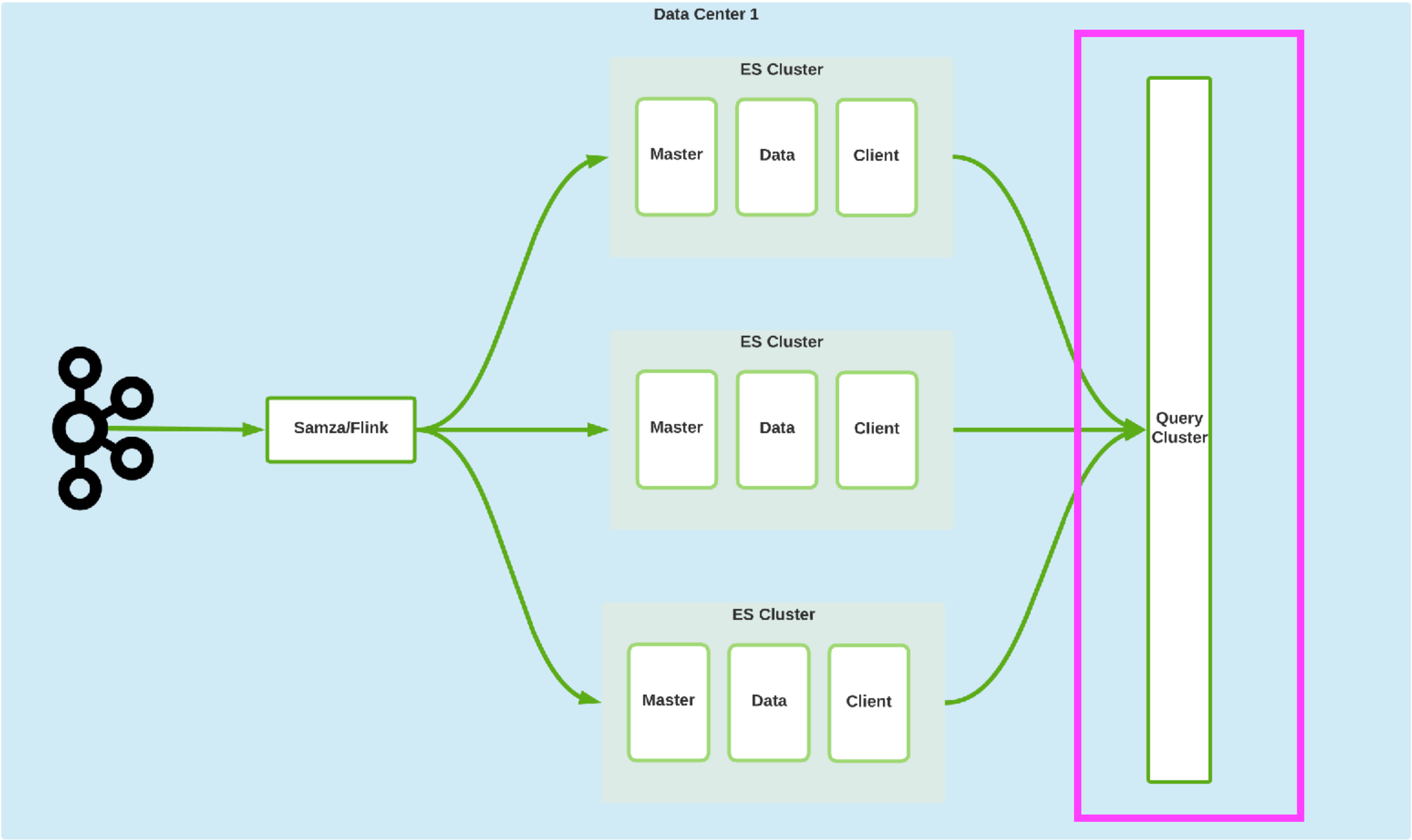
# Hide Query Optimization from Users

- Do we really expect every user to write Elasticsearch queries?
- What if someone issues a very expensive query?
- **Solution**: Isolation with a query layer

# Query Layer with Multiple Clusters

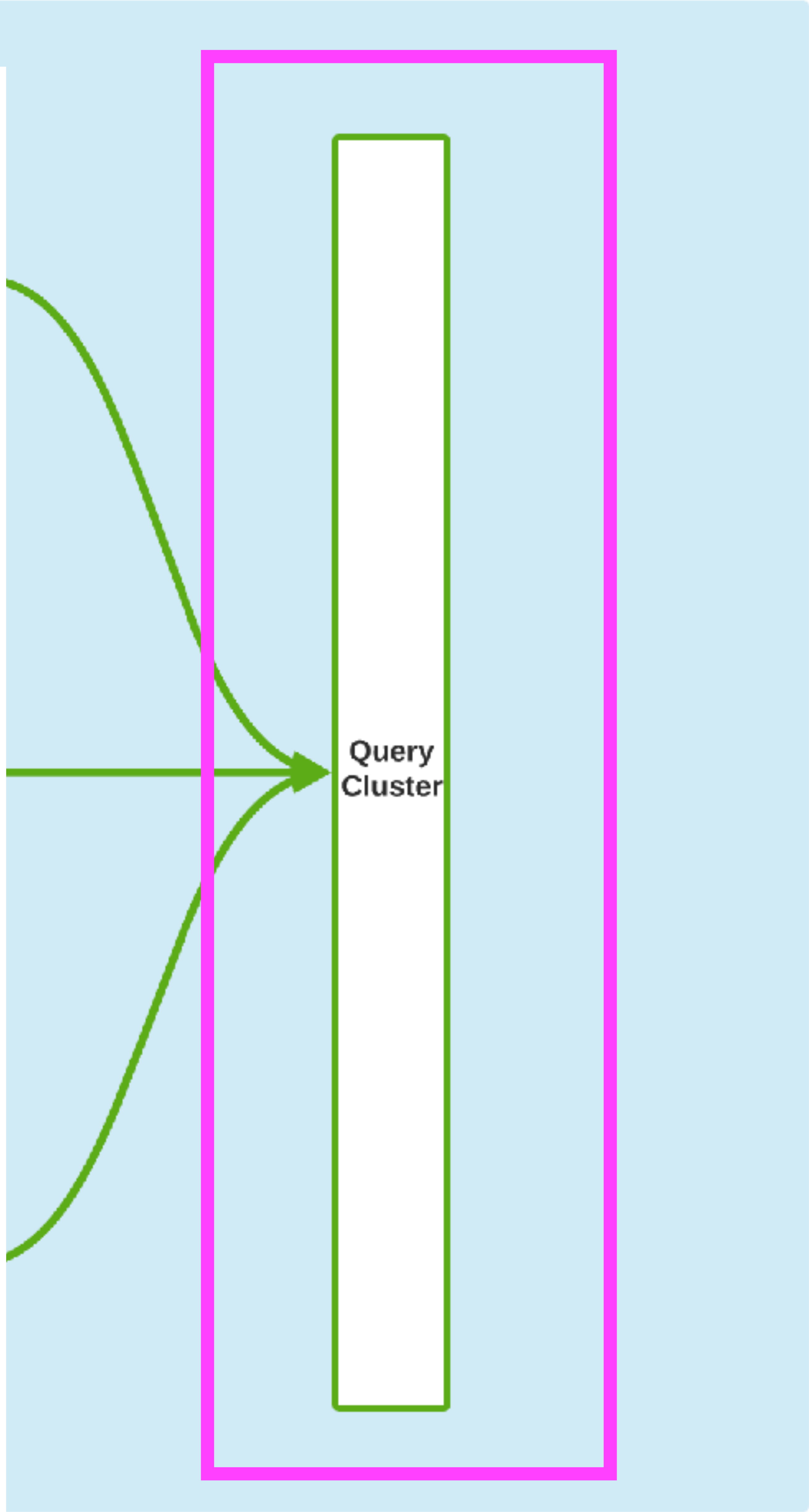


# Query Layer with Multiple Clusters



# Query Layer with Multiple Clusters

- Generate efficient Elasticsearch queries
- Rejecting expensive queries
- Routing queries - hardcoded first



# Efficient Query Generation

- "GROUP BY a, b"

```
{
  "aggs": {
    "a": {
      "terms": {
        "field": "a"
      },
      "aggs": {
        "b": {
          "terms": {
            "field": "b"
          }
        }
      }
    }
  }
}
```

# Rejecting Expensive Queries

```sql
SELECT count(*), hexagon, minute_of_day, city
FROM   trips
GROUP BY hexagon, minute_of_day, city
```

- 10,000 hexagons / city x 1440 minutes per day x 800 cities
- Cardinality: 11 Billion (!) buckets –> Out Of Memory Error

# Routing Queries

```
 "DEMAND":
     "CLUSTERS": {
       "TIER0": {
         "CLUSTERS": ["ES_CLUSTER_TIER0"],
       },
       "TIER2": {
         "CLUSTERS": ["ES_CLUSTER_TIER2"]
       }
     },
     "INDEX": "MARKETPLACE_DEMAND-",
     "SUFFIXFORMAT": “YYYYMM.WW",
     "ROUTING": “PRODUCT_ID”,
```

# Routing Queries

```
"DEMAND":
    "CLUSTERS": {
      "TIER0": {
        "CLUSTERS": ["ES_CLUSTER_TIER0"],
      },
      "TIER2": {
        "CLUSTERS": ["ES_CLUSTER_TIER2"]
      }
    },
    "INDEX": "MARKETPLACE_DEMAND-",
    "SUFFIXFORMAT": “YYYYMM.WW",
    "ROUTING": “PRODUCT_ID”,
```

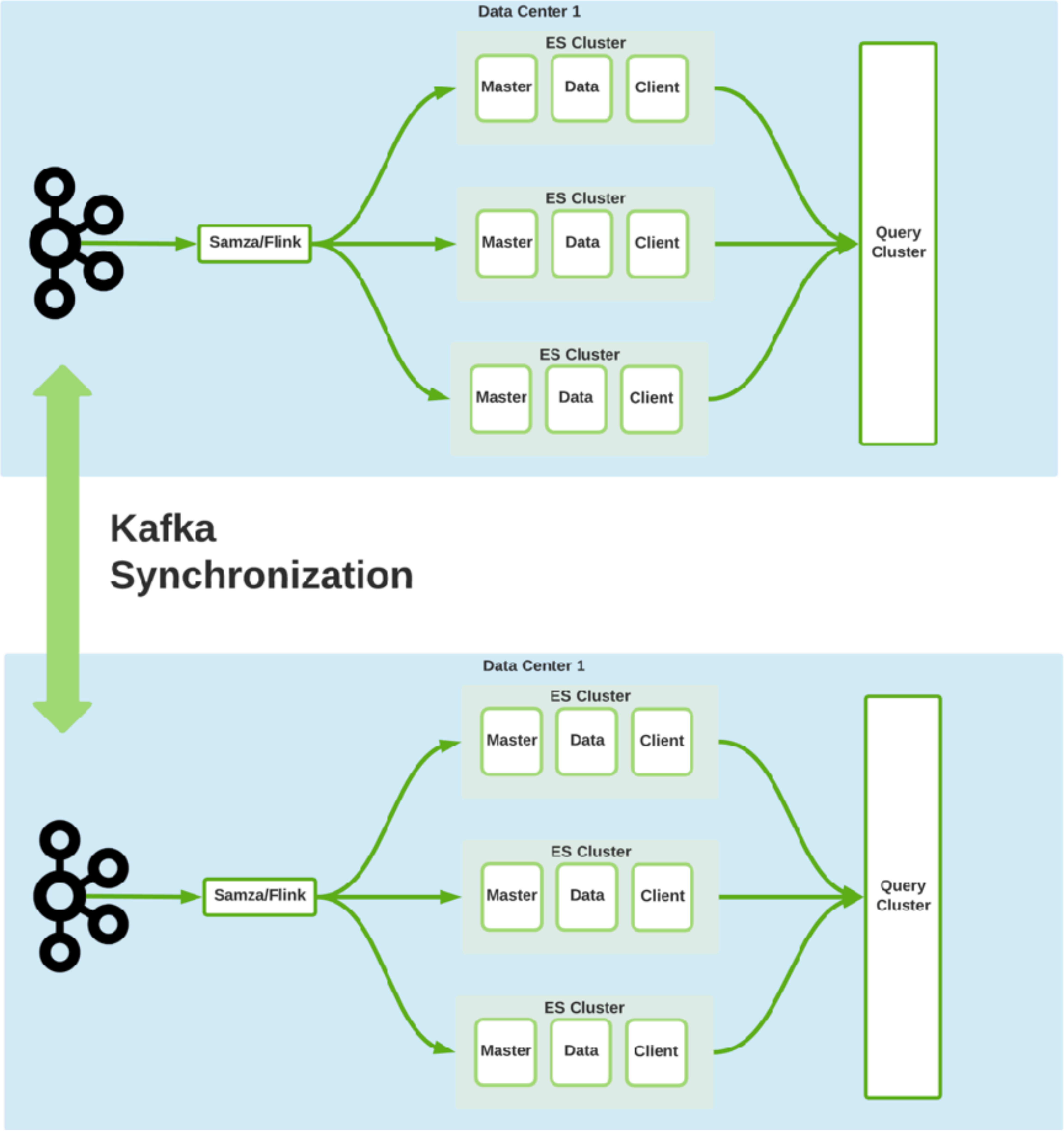
# Routing Queries

```
"DEMAND":
    "CLUSTERS": {
      "TIER0": {
        "CLUSTERS": ["ES_CLUSTER_TIER0"],
      },
      "TIER2": {
        "CLUSTERS": ["ES_CLUSTER_TIER2"]
      }
    },
    "INDEX": "MARKETPLACE_DEMAND-",
    "SUFFIXFORMAT": “YYYYMM.WW",
    "ROUTING": “PRODUCT_ID”,
```

# Routing Queries

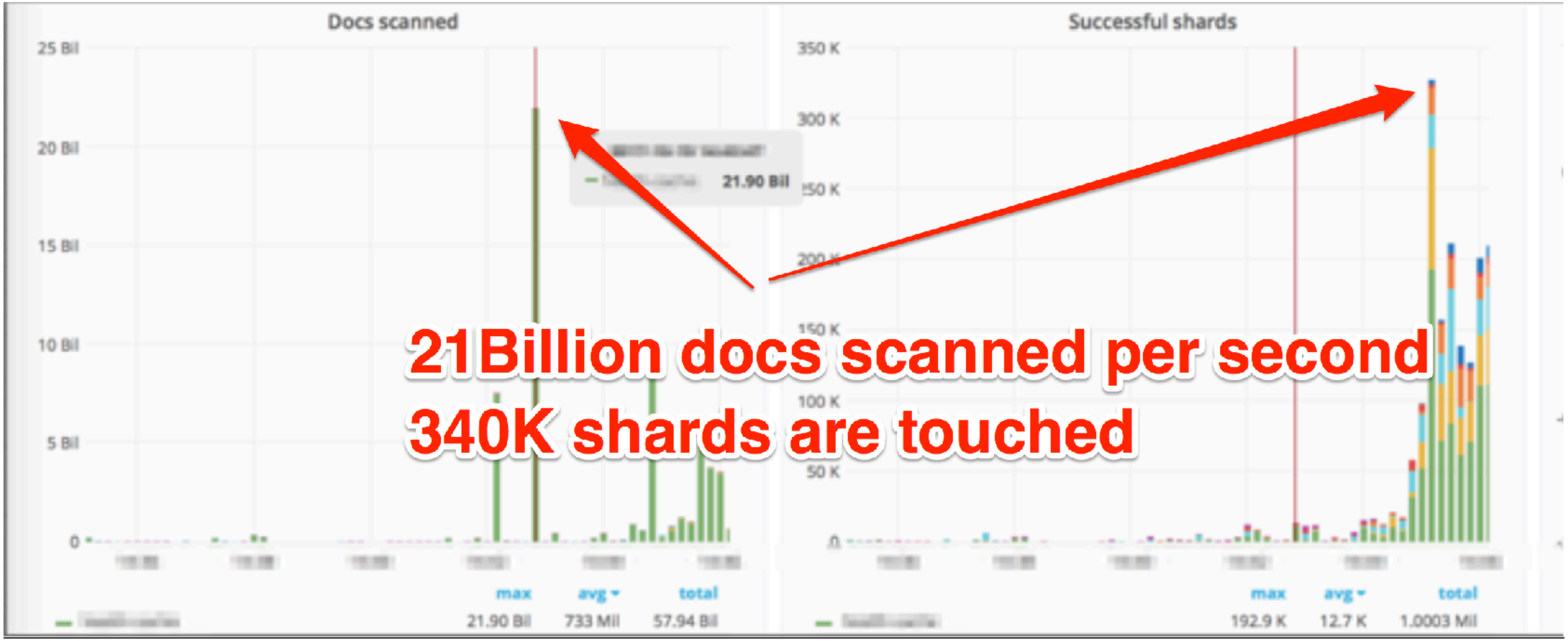
```
"DEMAND":
    "CLUSTERS": {
      "TIER0": {
        "CLUSTERS": ["ES_CLUSTER_TIER0"],
      },
      "TIER2": {
        "CLUSTERS": ["ES_CLUSTER_TIER2"]
      }
    },
    "INDEX": "MARKETPLACE_DEMAND-",
    "SUFFIXFORMAT": “YYYYMM.WW",
    "ROUTING": “PRODUCT_ID”,
```

# Summary of First Iteration

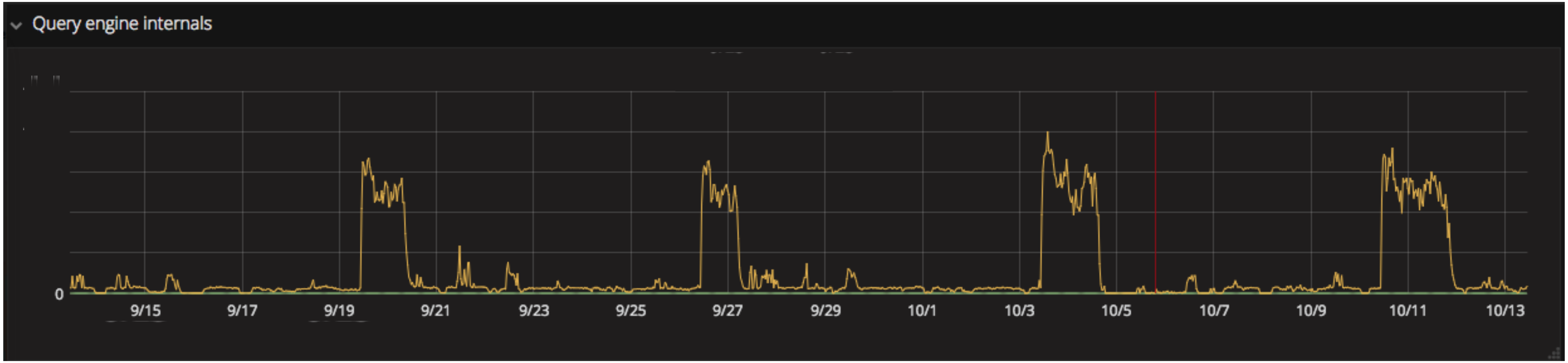


# Evolution: Success Breeds Failures

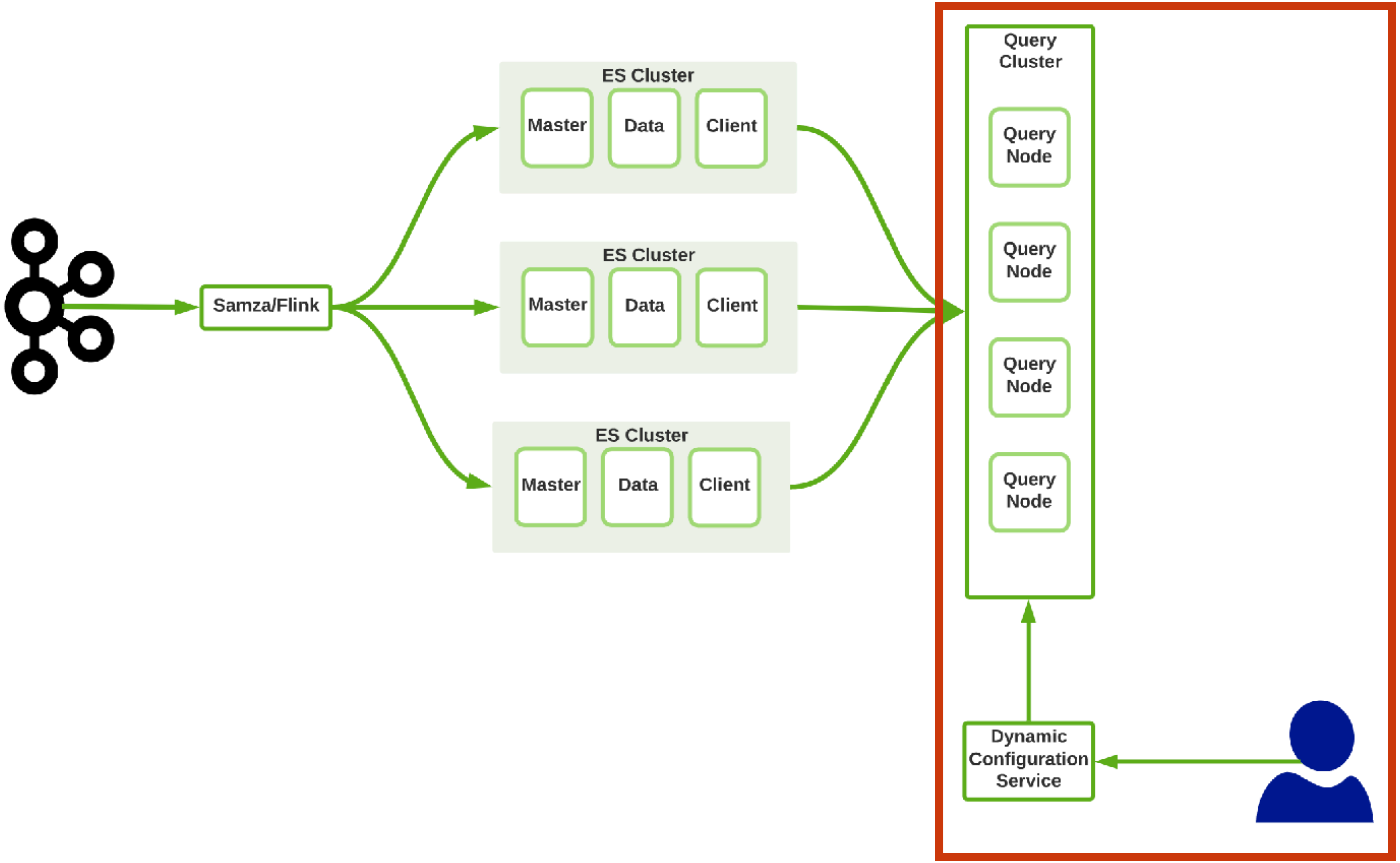
# Unexpected Surges



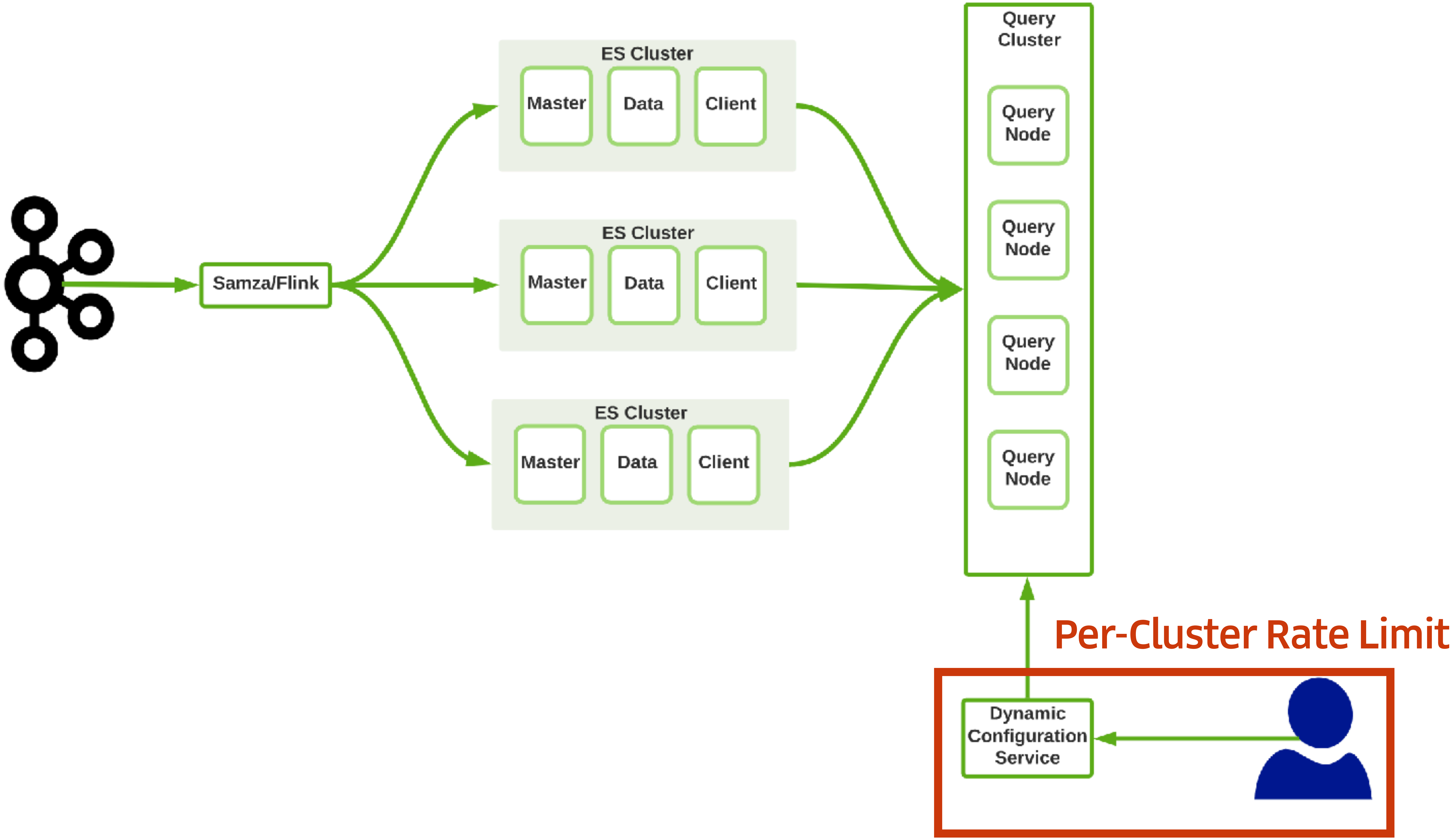
# Applications Went Haywire



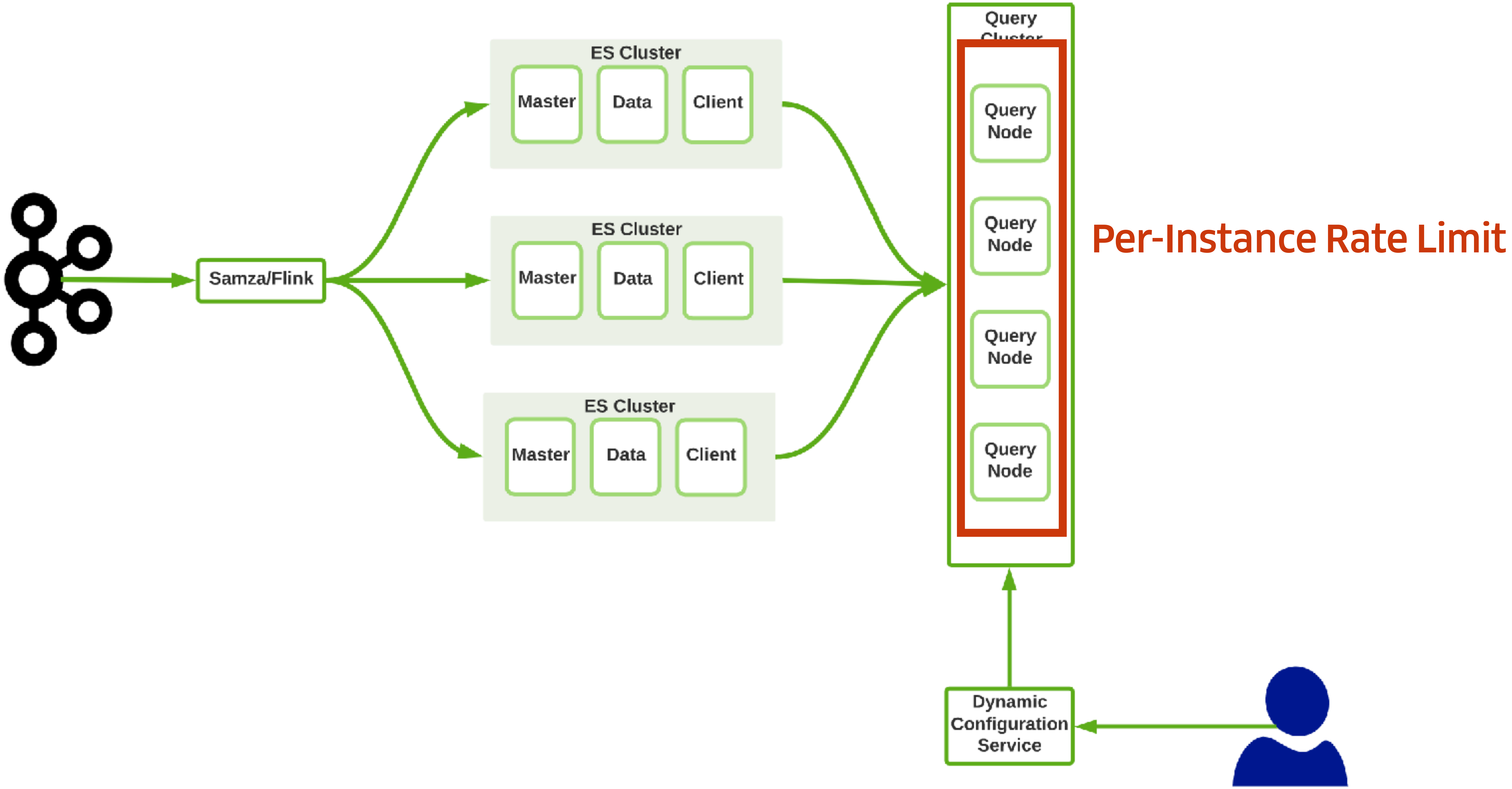
# Solution: Distributed Rate limiting



# Solution: Distributed Rate limiting



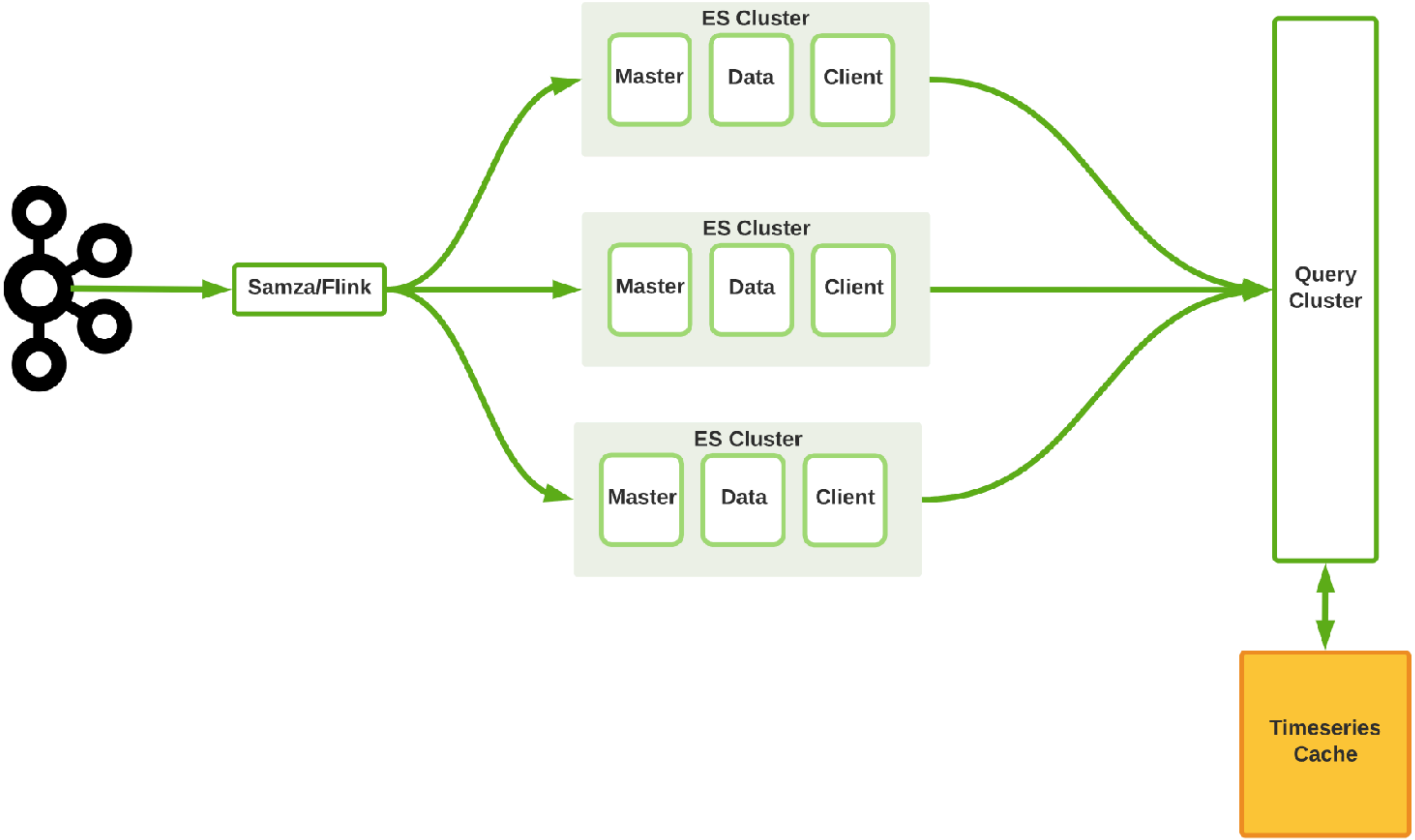
# Solution: Distributed Rate limiting



# Workload Evolved

- Users query months of data for modeling and complex analytics
- **Key insight:** Data can be a little stale for long-range queries
- **Solution**: Caching layer and delayed execution

# Time Series Cache



# Time Series Cache

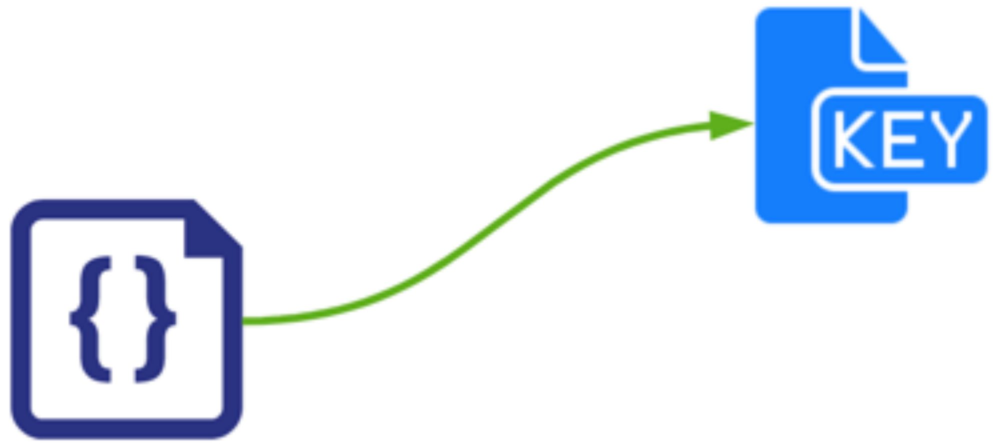
- Redis as the cache store
- Cache key is based on normalized query content and time range

# Time Series Cache

- Redis as the cache store
- Cache key is based on normalized query content and time range

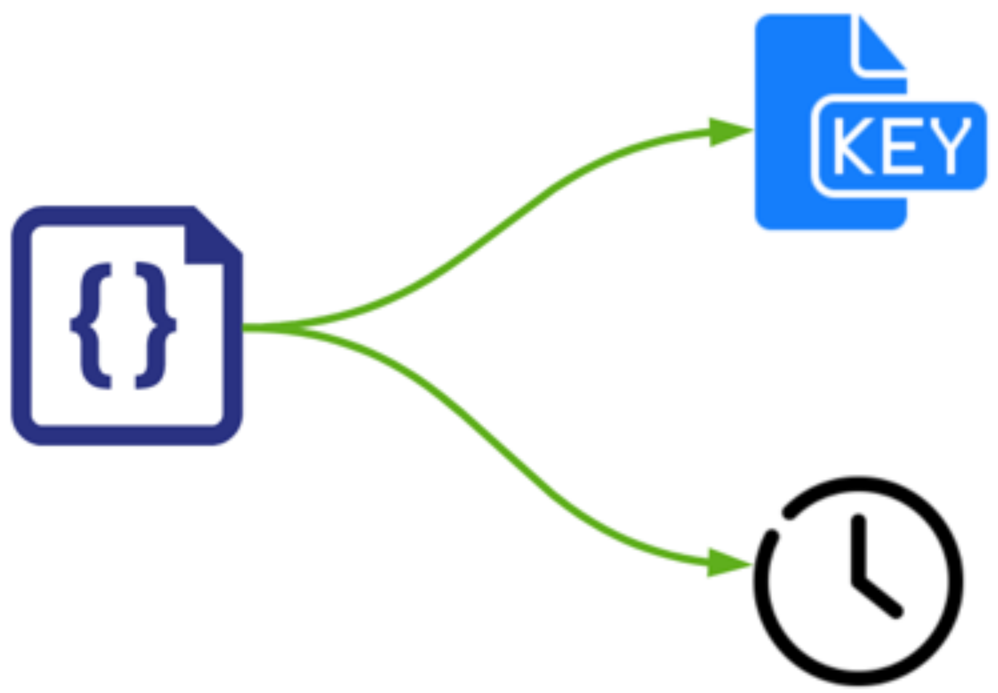
# Time Series Cache

- Redis as the cache store
- Cache key is based on normalized query content and time range



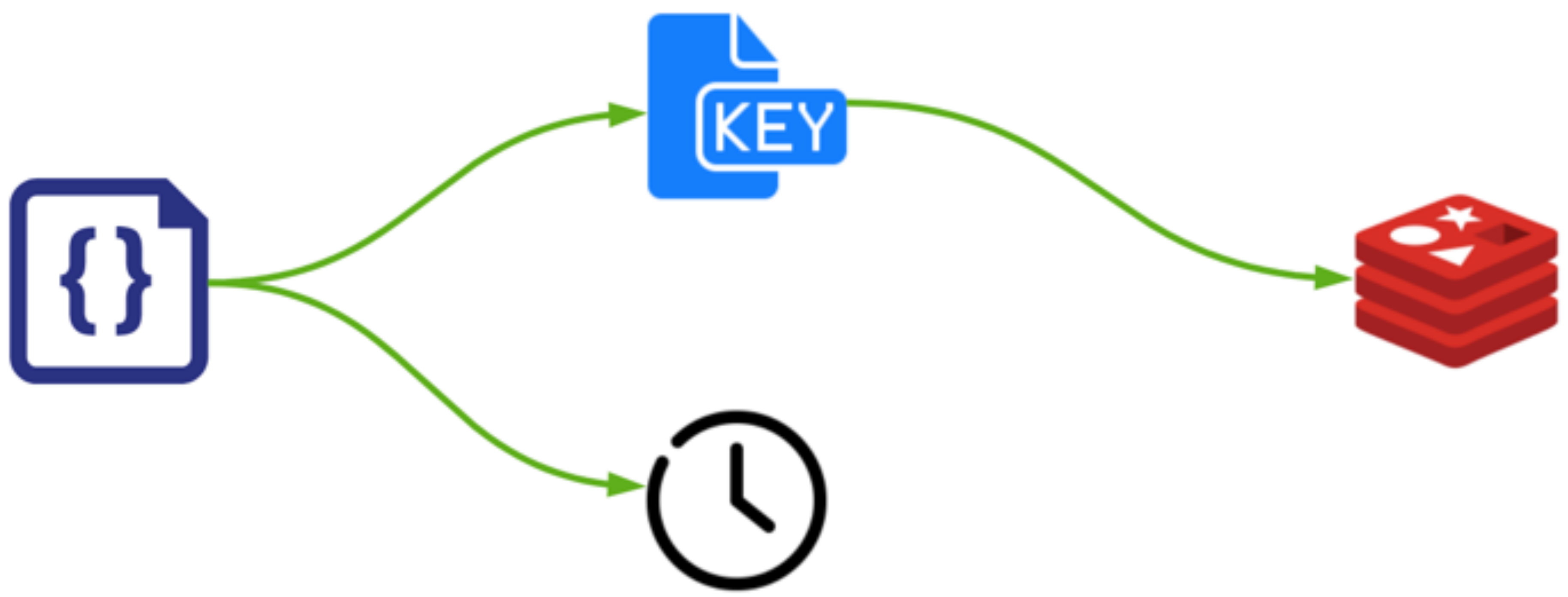
# Time Series Cache

- Redis as the cache store
- Cache key is based on normalized query content and time range



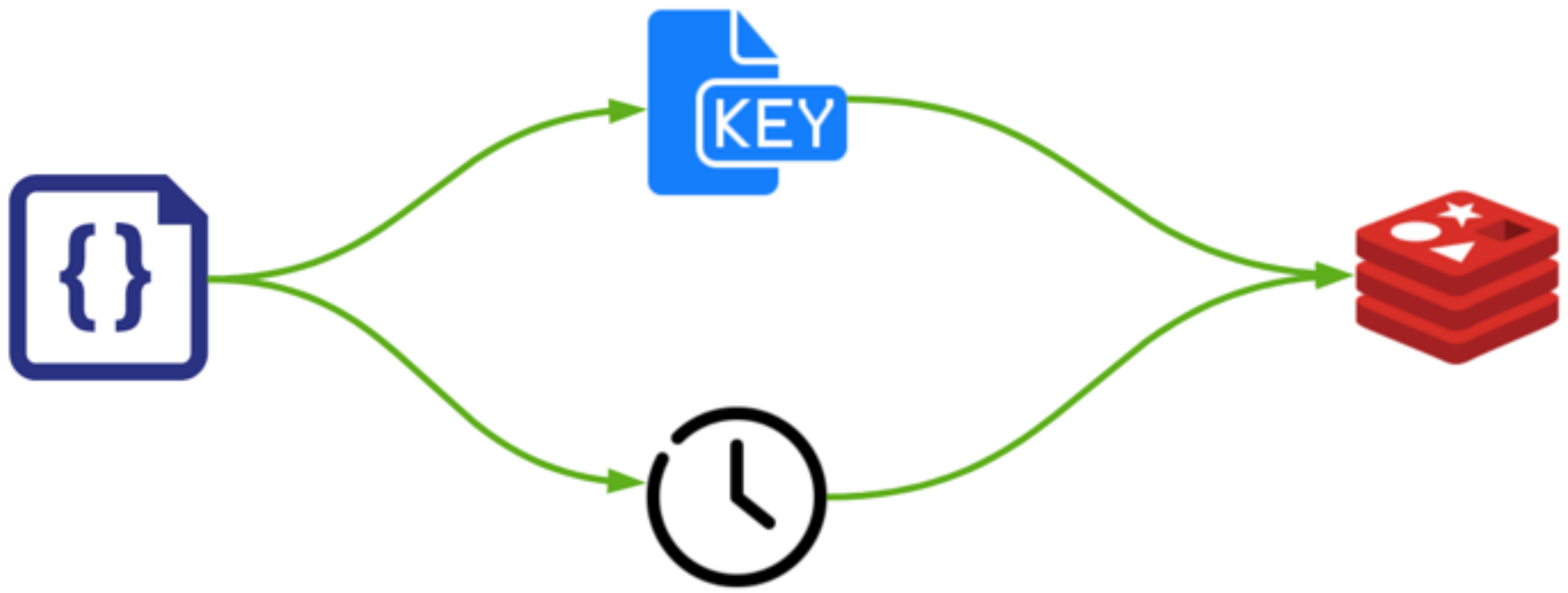
# Time Series Cache

- Redis as the cache store
- Cache key is based on normalized query content and time range



# Time Series Cache

- Redis as the cache store
- Cache key is based on normalized query content and time range



# Delayed Execution

- Allow registering long-running queries
- Provide cached but stale data for such queries
- Dedicated cluster and queued executions
- **Rationale**: three months of data vs a few hours of staleness
- Example: [-30d, 0d] –> [-30d, -1d]

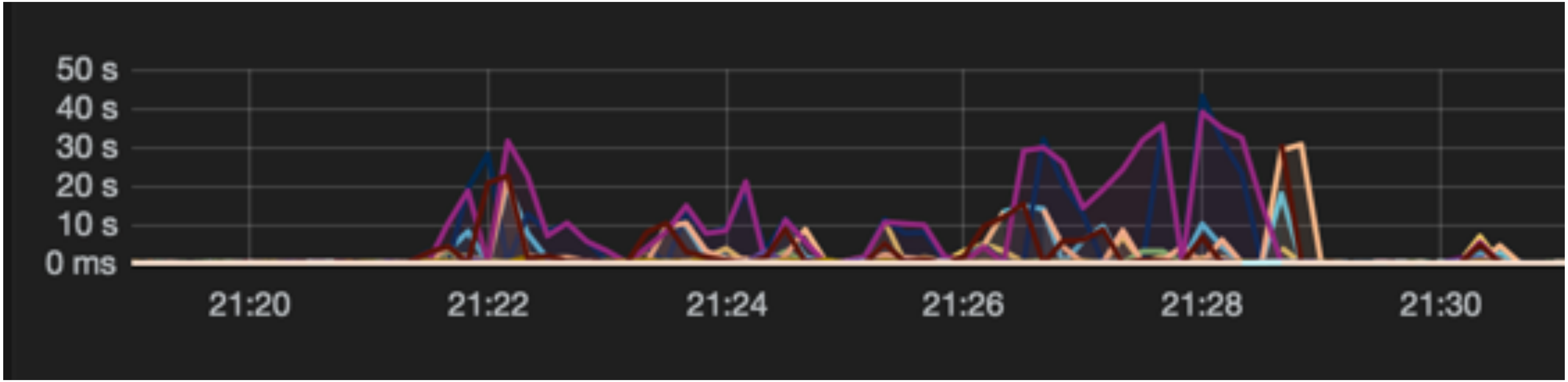
# Scale Operations

# Driving Principles

- Make the system transparent
- Optimize for MTTR - mean time to recover
- Strive for consistency
- Automation is the most effective way to get consistency

# Challenge: Diagnosis

- Cluster slowed down with all metrics being normal
- Requires additional instrumentation
- ES Plugin as a solution



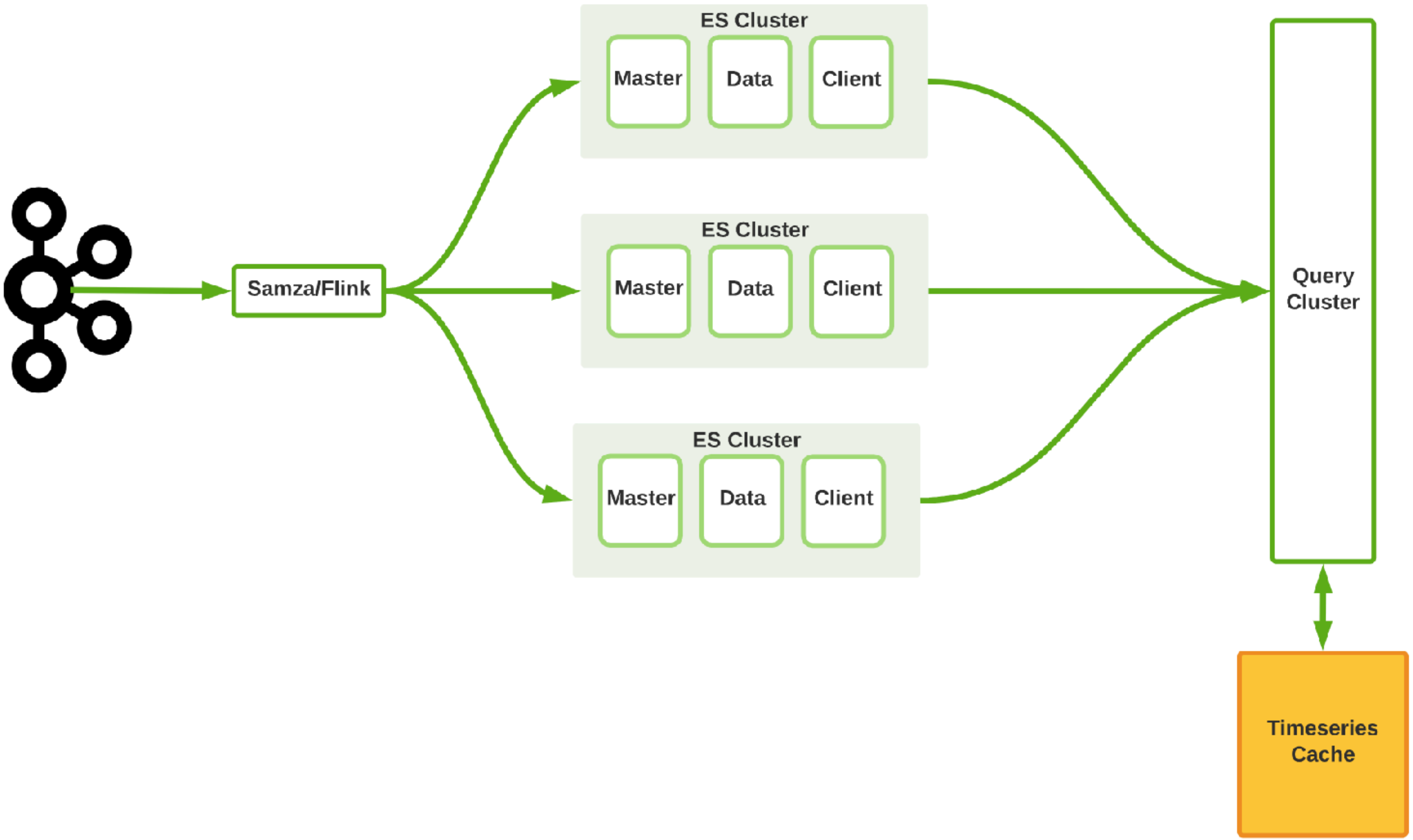
# Challenge: Cluster Size Becomes an Enemy

- Elasticsearch cluster becomes harder to operate as its size increases
- MTTR increases as cluster size increases
- Multi-tenancy becomes a huge issue
- Can’t have too many shards

# Federation

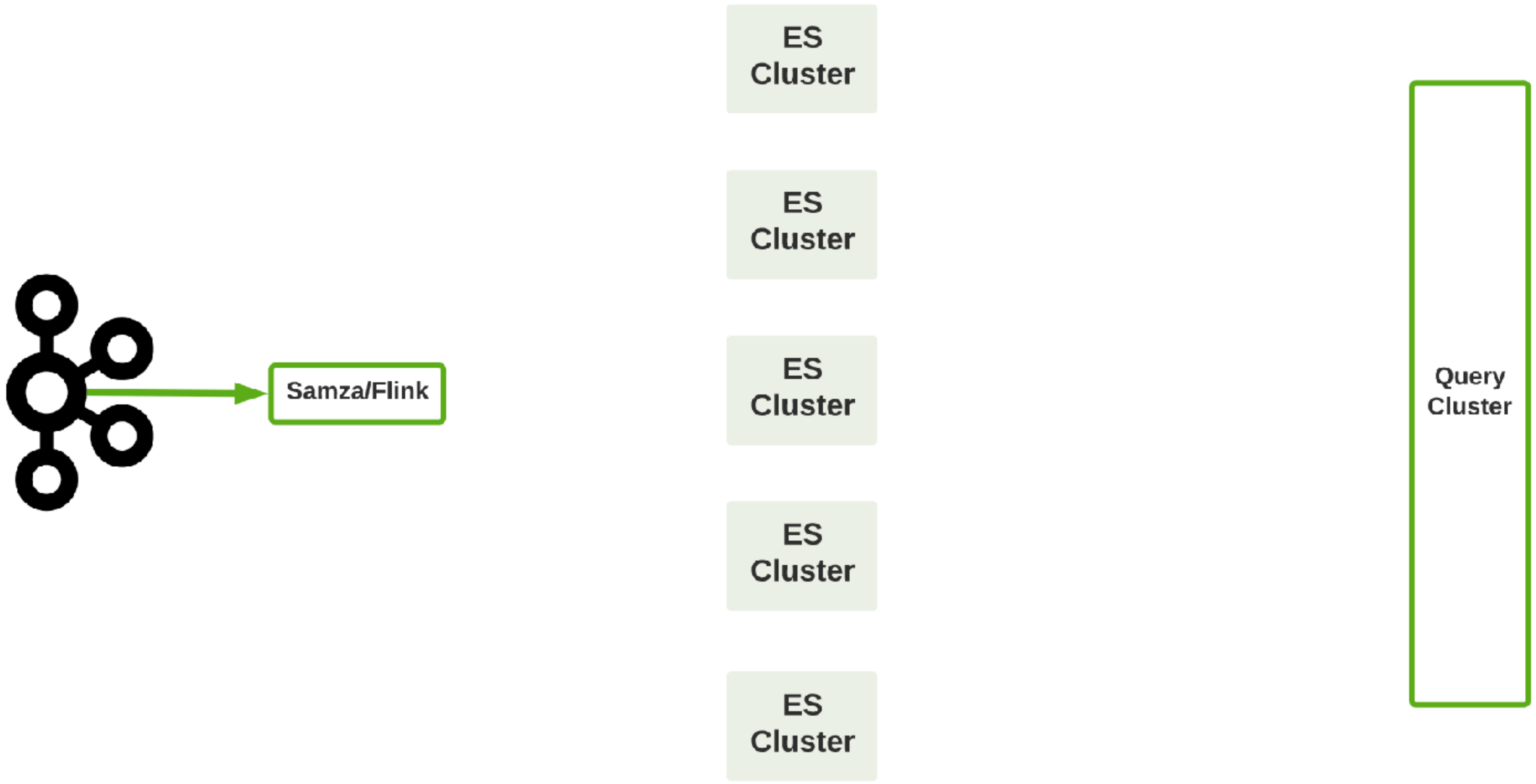
- 3 clusters –> many smaller clusters
- Dynamic routing
- Meta-data driven

# Federation



Samza/Flink
ES Cluster
Master
Data
Client
ES Cluster
Master
Data
Client
ES Cluster
Master
Data
Client
Query Cluster
Timeseries Cache

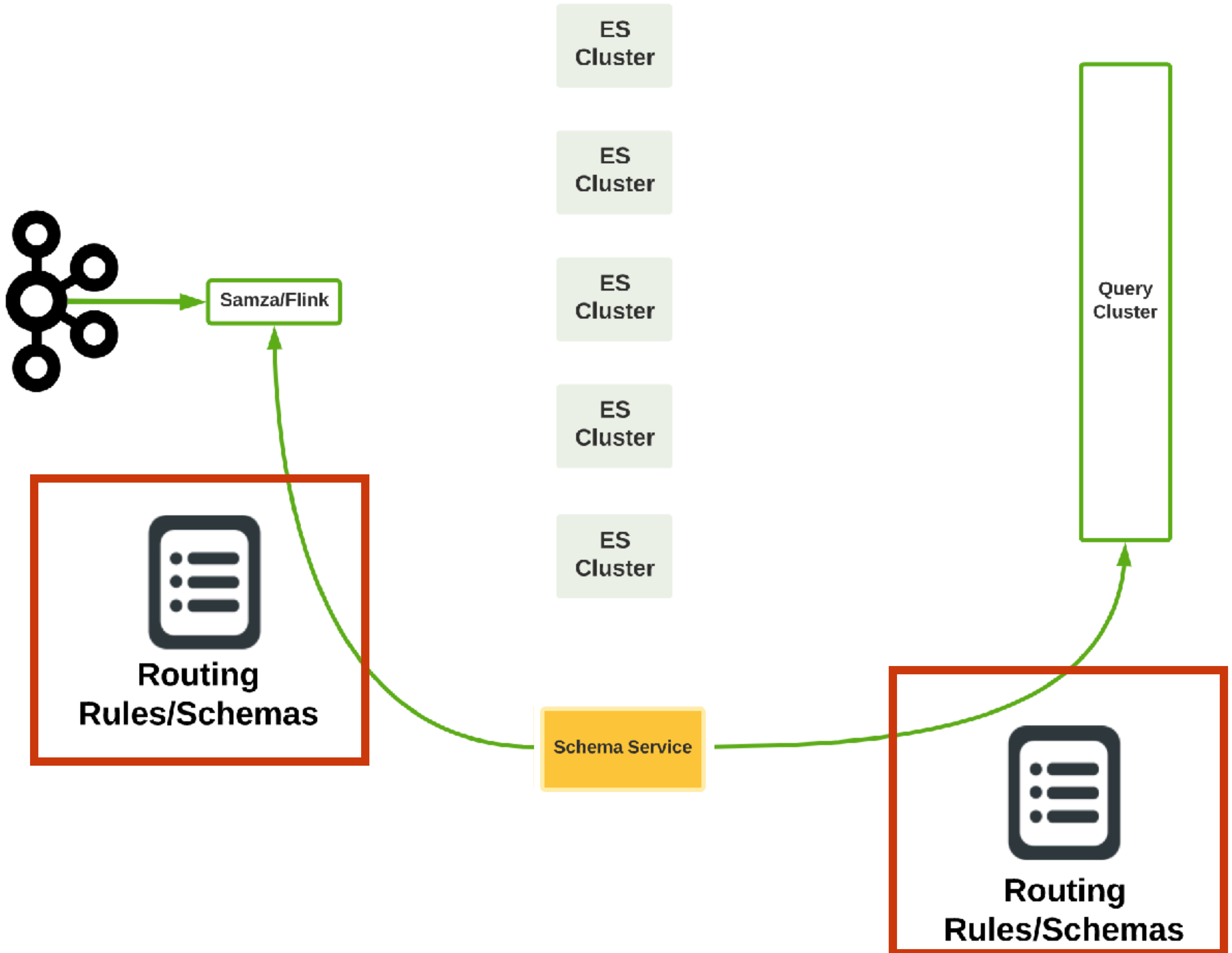
# Federation


Samza/Flink
ES Cluster
ES Cluster
ES Cluster
ES Cluster
ES Cluster
Query Cluster

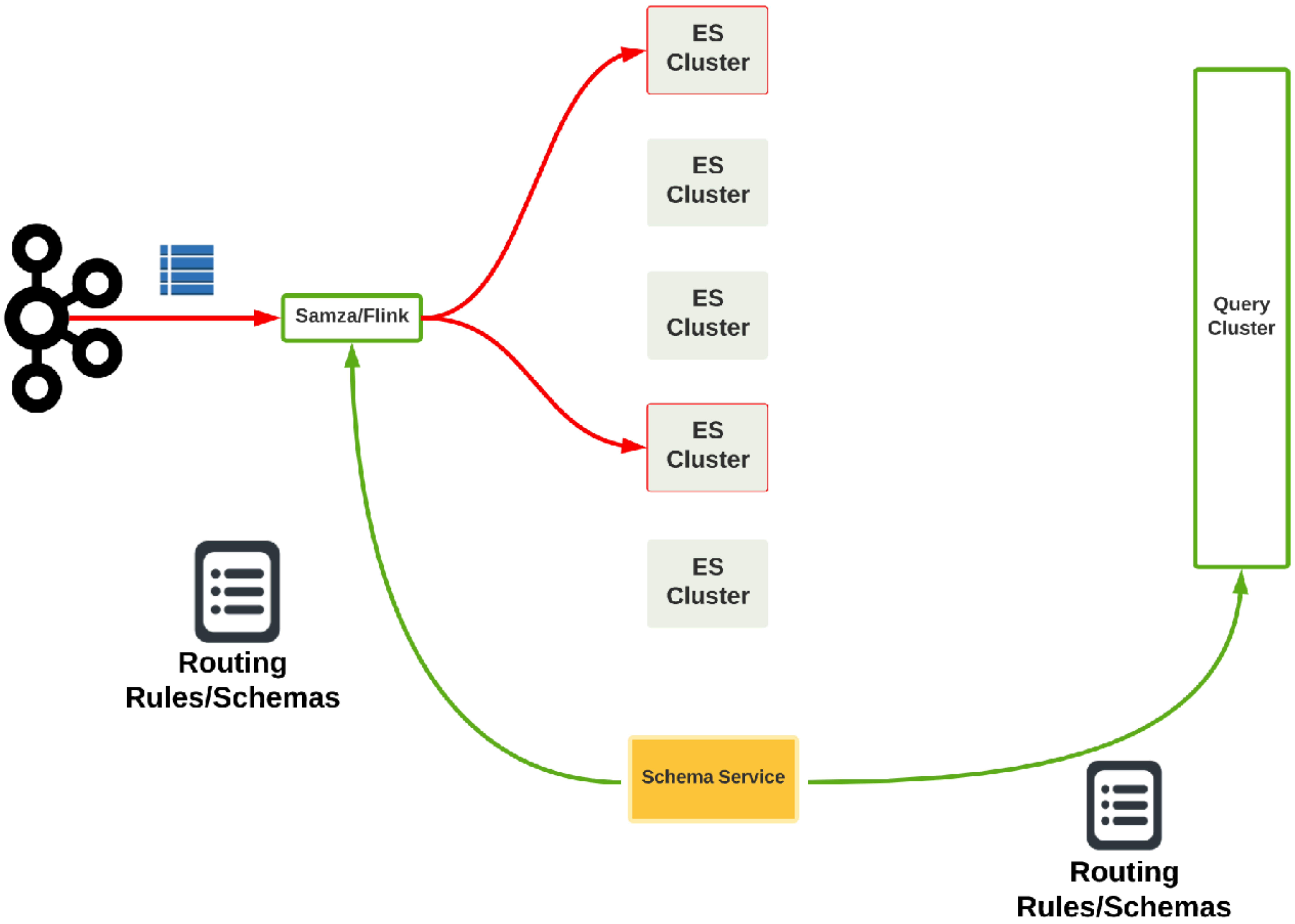
# Federation


Samza/Flink
ES Cluster
ES Cluster
ES Cluster
ES Cluster
ES Cluster
Query Cluster
Schema Service

# Federation

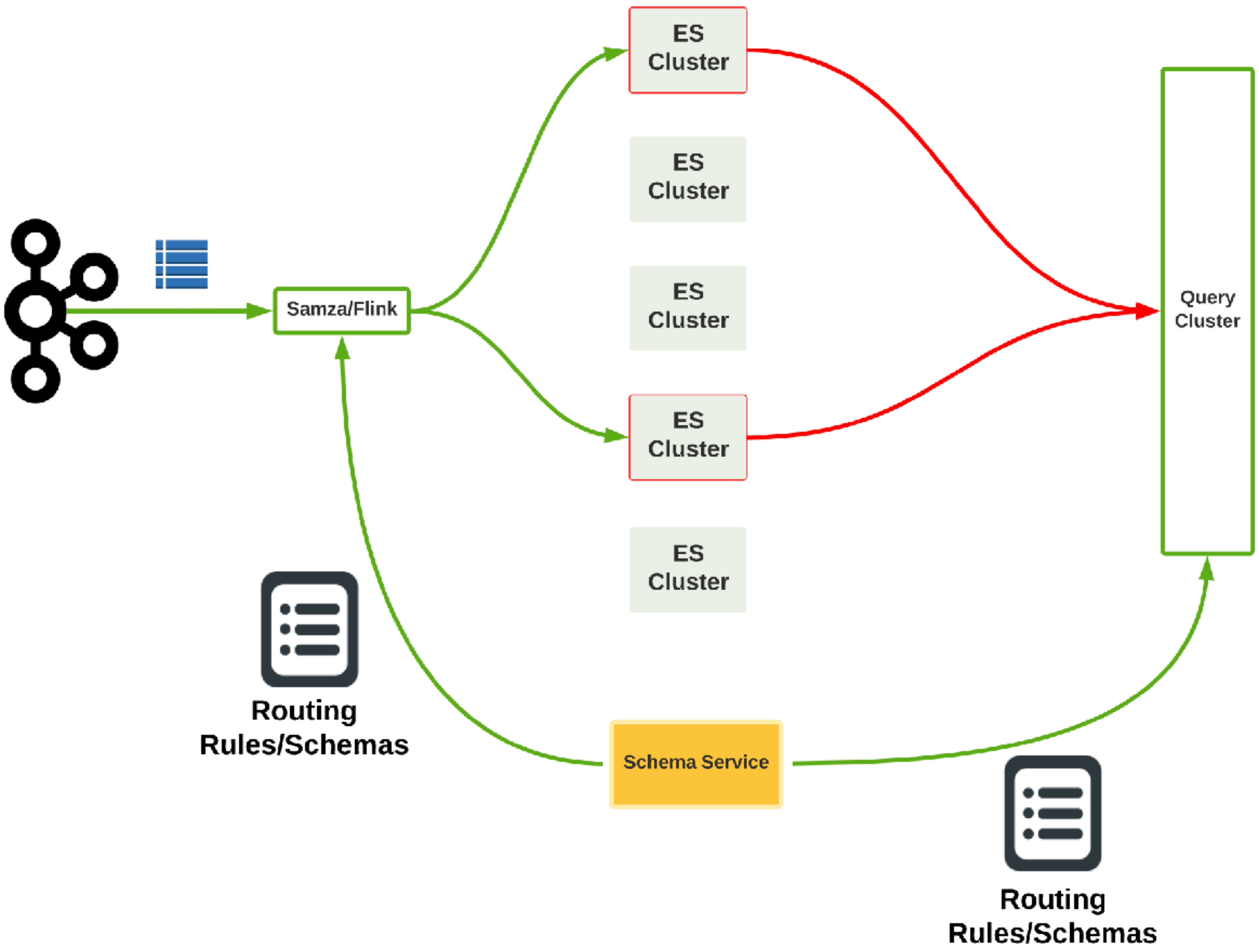

ES Cluster
ES Cluster
ES Cluster
ES Cluster
ES Cluster
Query Cluster
Samza/Flink
Routing Rules/Schemas
Schema Service
Routing Rules/Schemas

# Federation



Samza/Flink
ES Cluster
ES Cluster
ES Cluster
ES Cluster
ES Cluster
Query Cluster
Routing Rules/Schemas
Schema Service
Routing Rules/Schemas

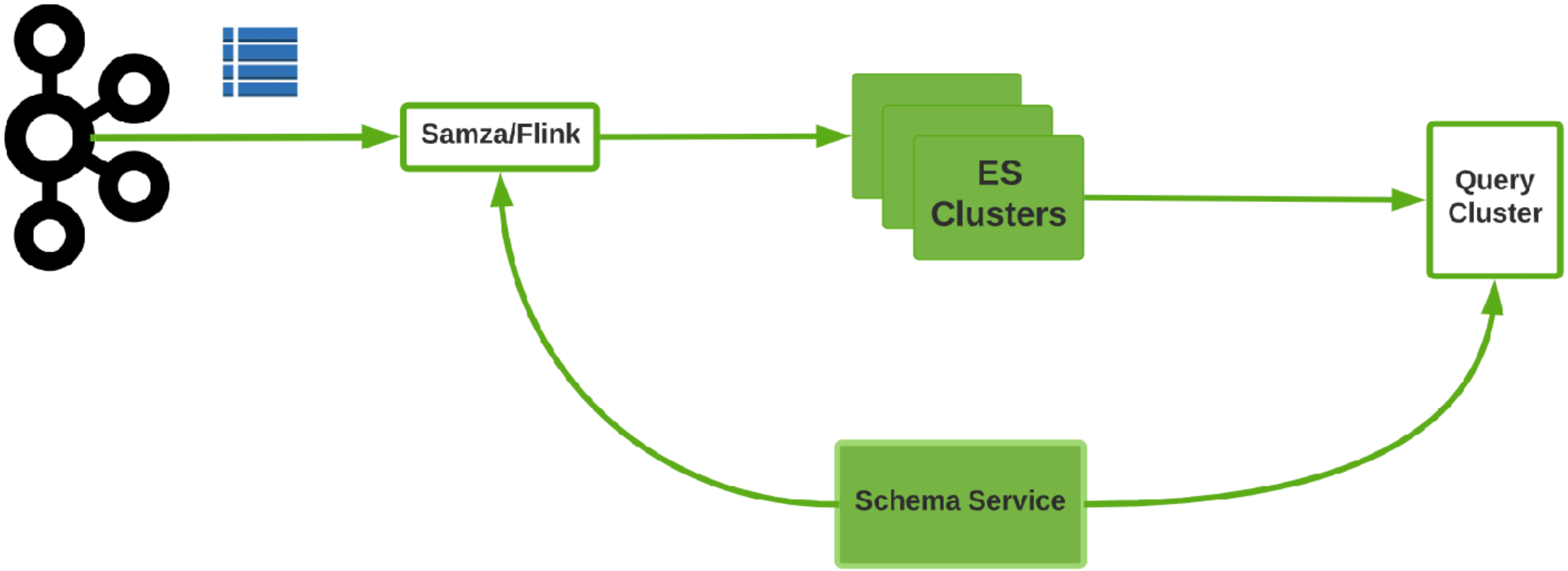
# Federation



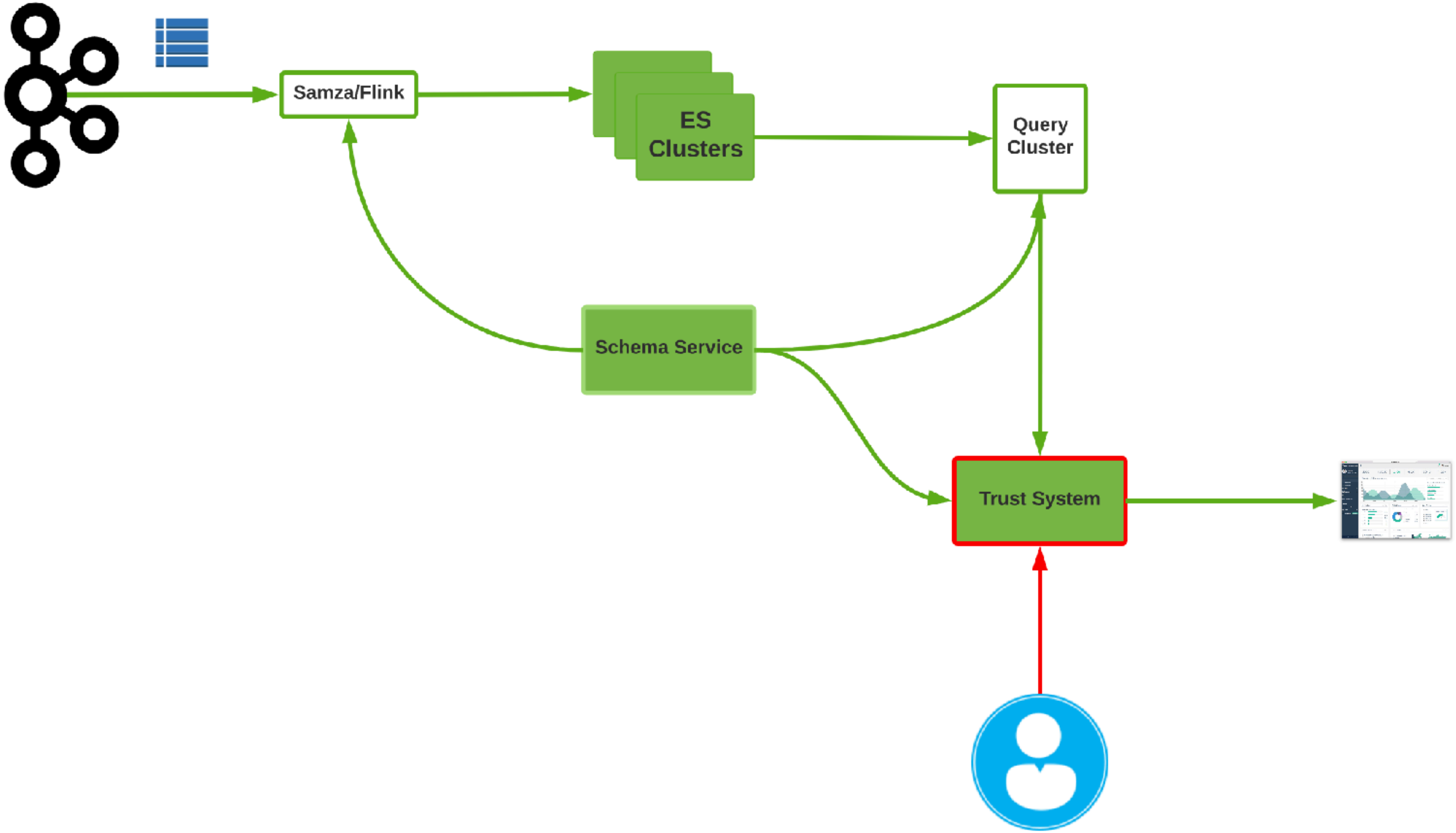
ES
Cluster
ES
Cluster
ES
Cluster
ES
Cluster
ES
Cluster
Samza/Flink
Query
Cluster
Routing
Rules/Schemas
Schema Service
Routing
Rules/Schemas

# How Can We Trust the Data?

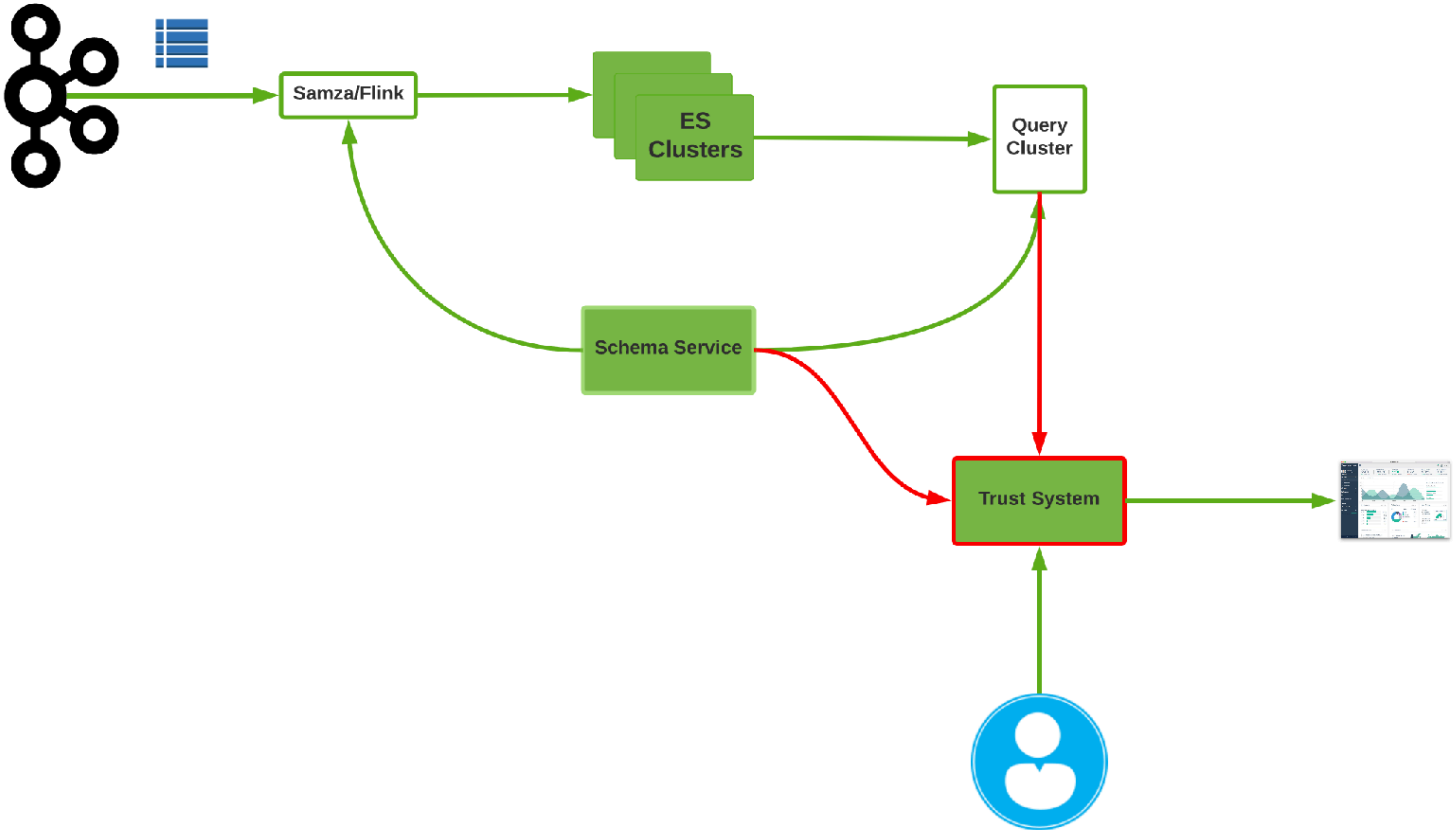
# Self-Serving Trust System



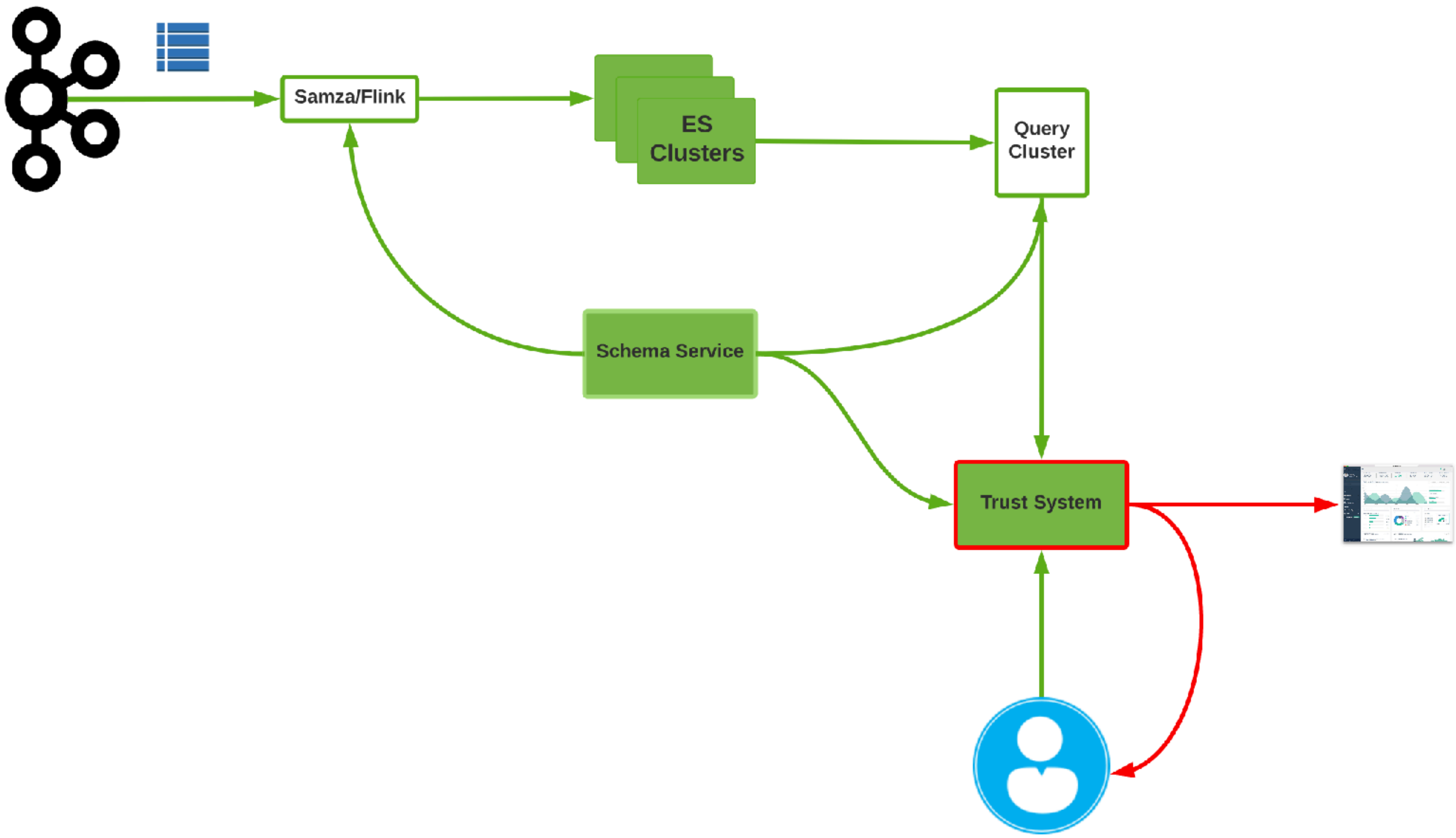
# Self-Serving Trust System



# Self-Serving Trust System



# Self-Serving Trust System

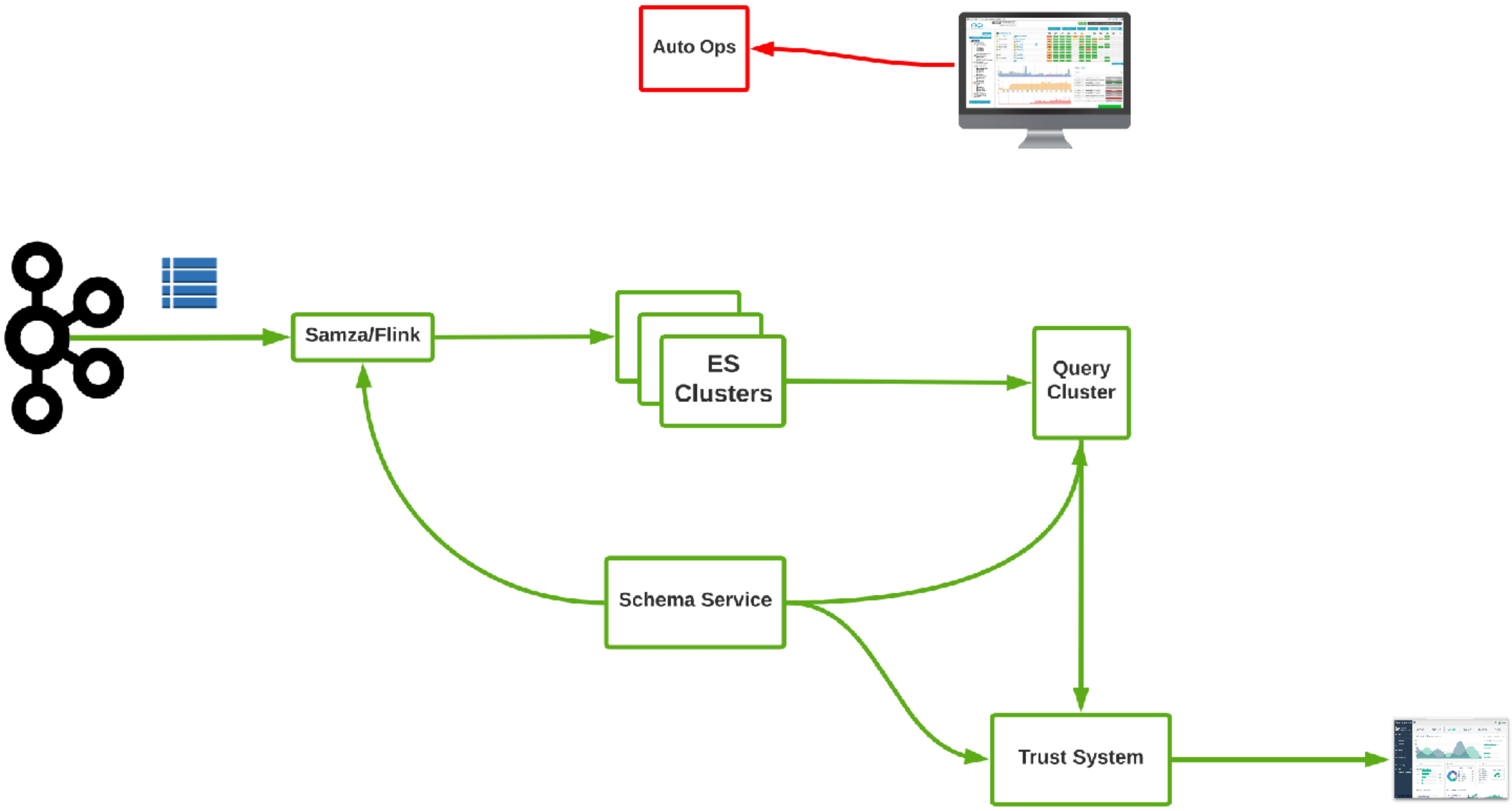


# Too Much Manual Maintenance Work

# Too Much Manual Maintenance Work

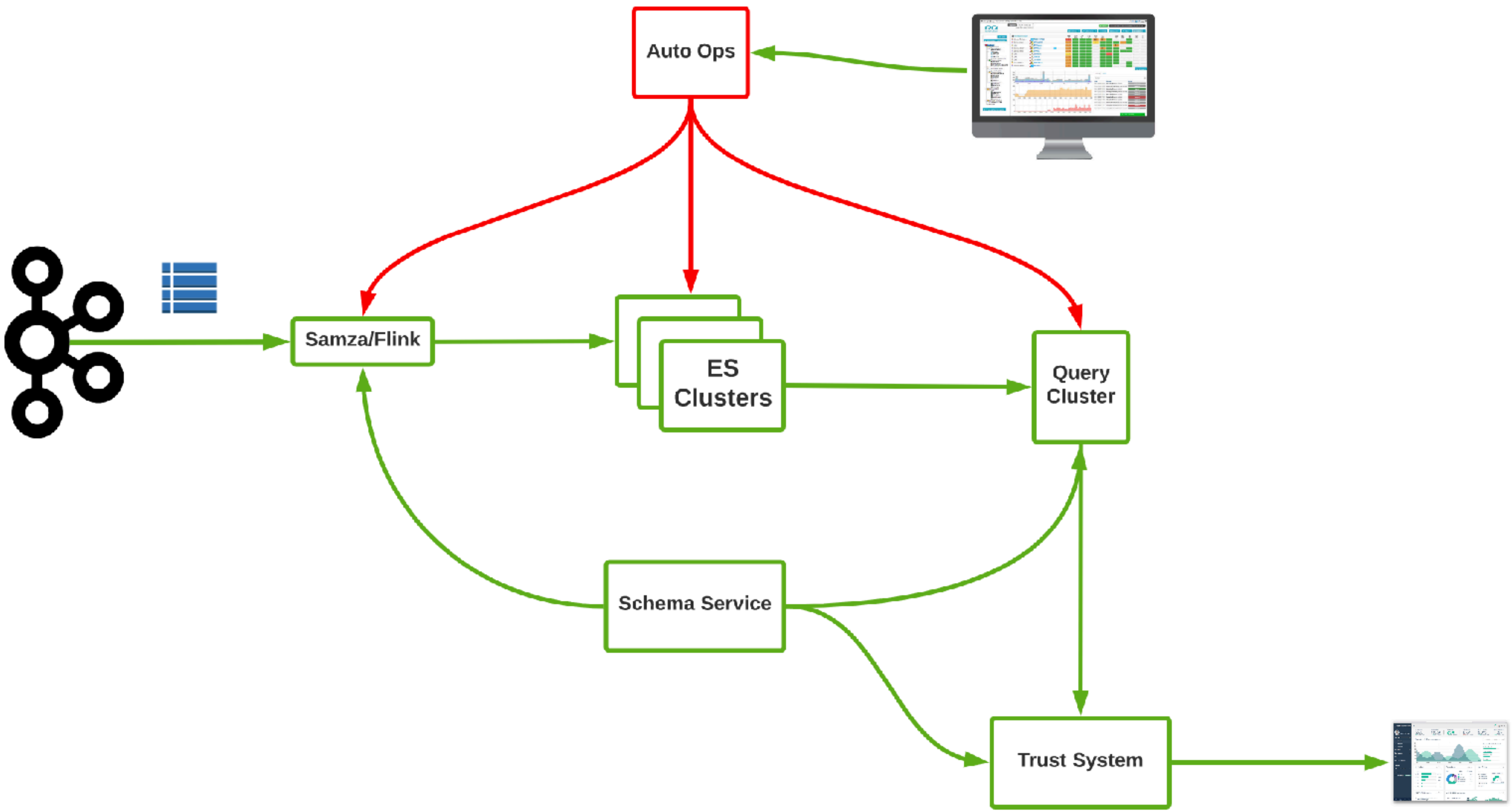
- Adjusting queue size
- Restart machines
- Relocating shards

# Auto Ops



Auto Ops
Samza/Flink
ES
Clusters
Query
Cluster
Schema Service
Trust System

# Auto Ops



Auto Ops
Samza/Flink
ES
Clusters
Query
Cluster
Schema Service
Trust System

# Ongoing Work for the Future

# Future Work

- Strong reliability
- Strong consistency among replicas
- Multi-tenancy

# Summary

- Three dimensions of scaling: ingestion, query, and operations
- Be simple and practical: successful systems emerge from simple ones
- Abstraction and data modeling matter
- Invest in thorough instrumentation
- Invest in automation and tools