



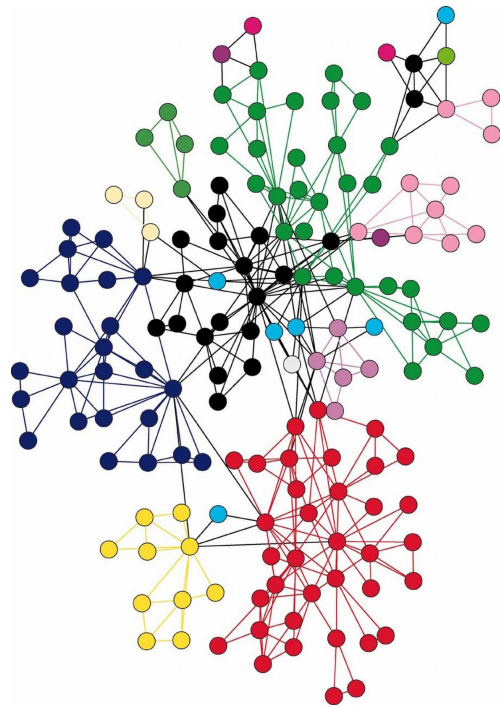
New Tech Stack for AGM business

Vinh, Olivia, Suhas

Neo4j - A graph database

Introducing: neo4j

- NoSQL database management system that represents data as a **graph**.
- Neo4j **outperforms** relational database system when it comes to graphs.



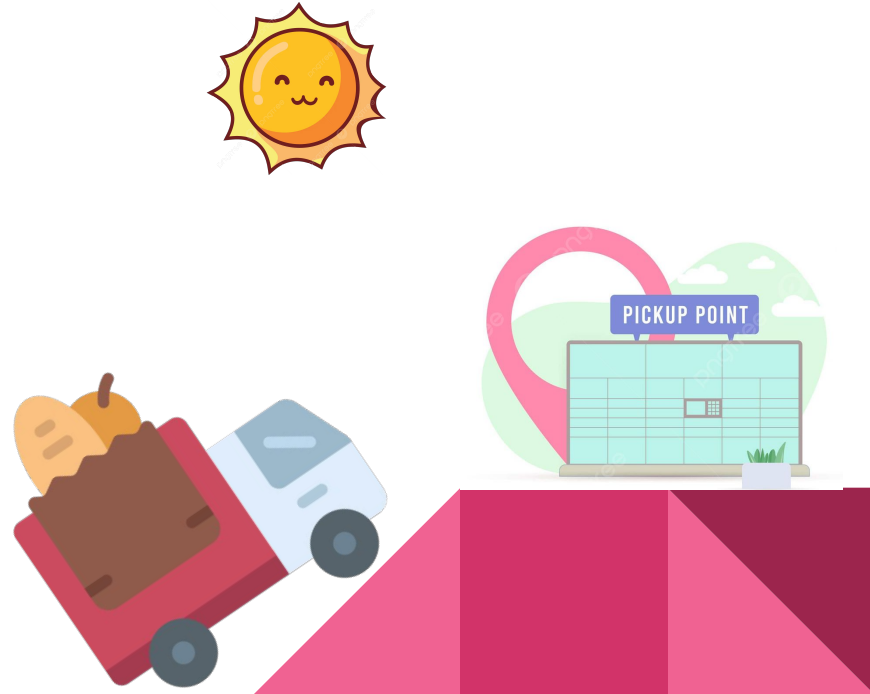
Neo4j in industry

- NASA: Getting to Mars faster with knowledge graph
- COCOM: COVID-19 Contact Tracing
- Zurich: Fraud Investigation
- Vanguard: Improving code quality
- eBay: Intelligent Commerce
- CISCO: Graph Analysis saves 4 millions employees hours
- Support wide ranges of algorithms
 - Graph algorithms
 - AI/ML



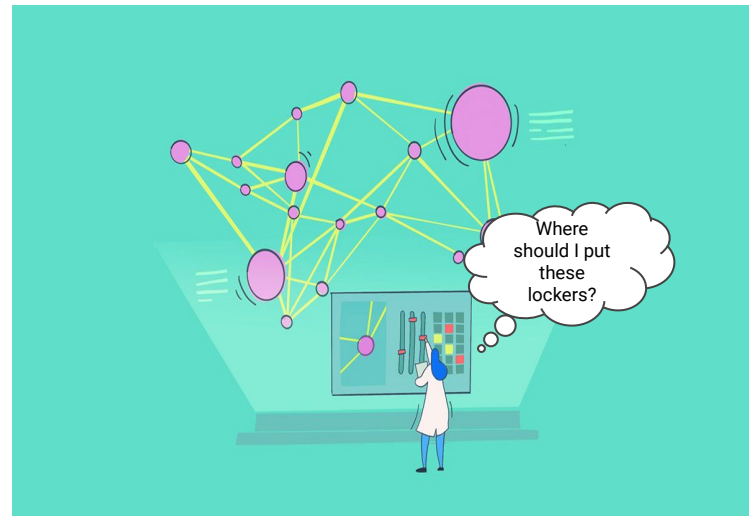
Future of Food Delivery: Convenient Pickup Lockers

- Food delivery services yield high customer satisfaction, but implementation logistics and cost are a high barrier to market entry.
- Pickup lockers (like the ASUC Amazon Hub) provide similar levels of convenience with reduced operational overhead



Challenges and Constraints for Pickup Lockers

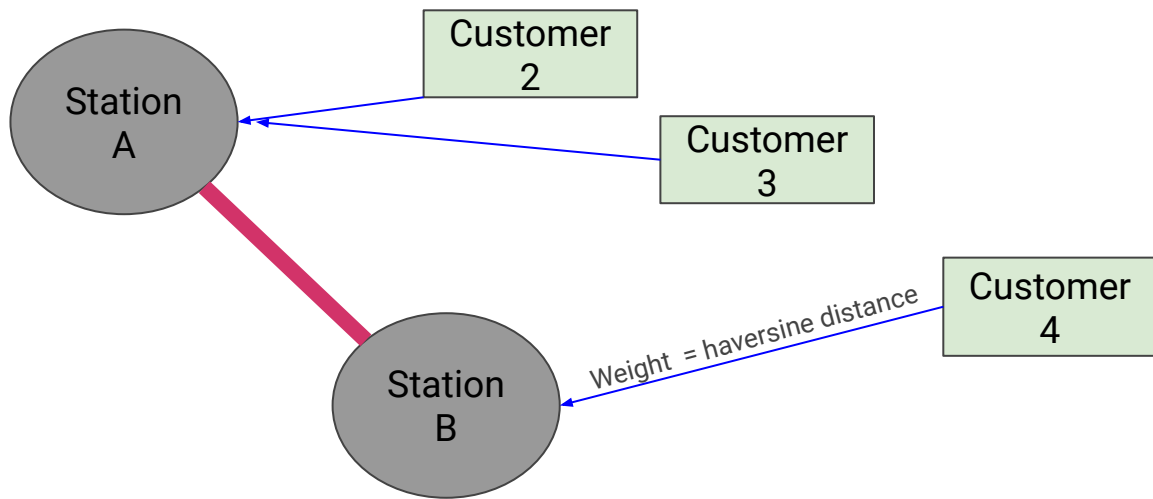
- Convenience
- Limited Resources
- Approach to designing PoC: Measure BART Station Centrality



Example: Station A vs. Station B

- Stations and Customers are **nodes**.
- Traveling time and distances between: **edges**.

Question: Is it better to place a locker at Station A or Station B?




Answer: choose the station with a higher centrality!

Neo4j Performance

- Optimized for querying relationship
- Real-Time data processing
- Horizontal scaling up
- Performance tuning
- Capable handling billions of nodes and relationships

Comparison with RDBMS:

- SQL is powerful for relational operations but is less intuitive for graph traversals.
 - graph traversals can be less efficient compared to graph databases. Poor performance.
 - Faces scaling challenges when dealing with highly connected graph data.
- 

Findings and Recommendations

- Neo4j can solve our business problems efficiently.
- Easy to scale up
- Flexibility on changes
- Superior than RDBMS for graph's challenges

→ **We should adapt Neo4j into our tech stacks**



Route planning for customer pickup



Goal: Want to minimize the transportation barrier to utilizing pickup lockers

Solution: Reduce customer's cognitive load by using providing shortest public transit route between customer location and pickup lockers. To do this, build an app-integrated route planning feature that works with delivery alerts to streamline customer experience.

Feature Components:

- Compute shortest path between a customer-inputted location and their selected pickup locker with Neo4j and Dijkstra's
- Pre-compute and store previous/likely paths with MongoDB
- Integrate customer travel time estimates with real-time delivery status updates using Redis

Route planning for customer pickup



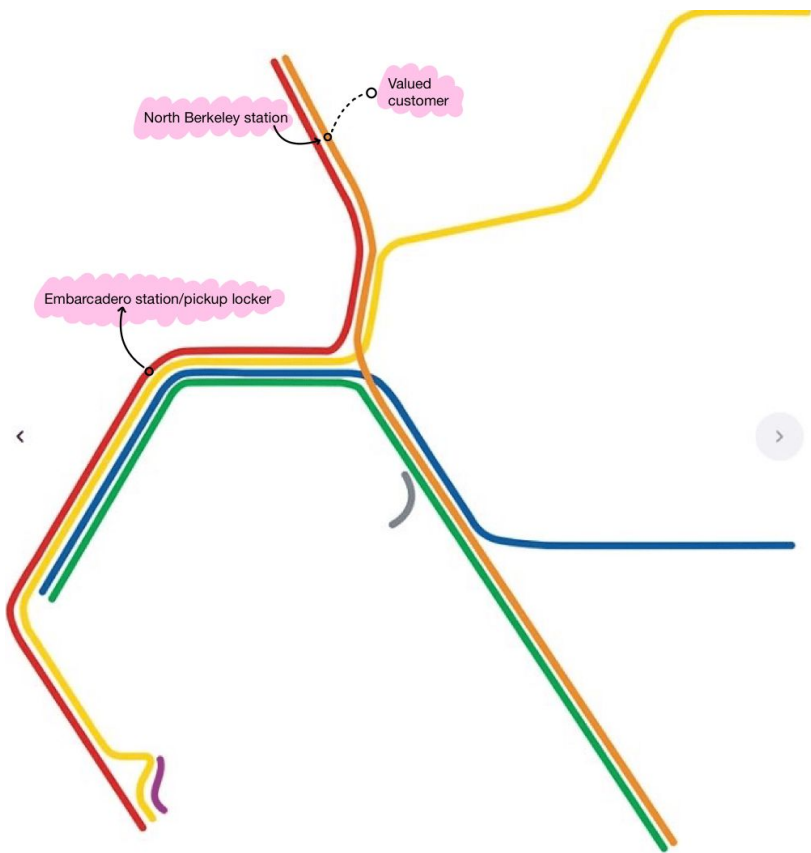
Goal: Want to minimize the transportation barrier to utilizing pickup lockers

Solution: Reduce customer's cognitive load by using providing shortest public transit route between customer location and pickup lockers.

Feature Components:

- Compute shortest path and likely paths between a customer and their preferred pickup lockers
- Real-time delivery status updates using Redis

Compute Shortest Path w/ Dijkstra's



- Graph nodes types include customers, departure/arrival stations with line information, station lockers.
- Graph edges represent travel time in seconds.
- Run Dijkstra's to compute shortest path:

Customer ID: 316
Haversine Distance from customer location to North Berkeley :
0.31 miles
Walk time to from customer location to North Berkeley :
7.33 minutes

Total Cost: 2558
Minutes: 42.6

```
from customer_id 316, 0, 0
depart North Berkeley, 878, 878
red North Berkeley, 0, 878
red Downtown Berkeley, 120, 998
red Ashby, 180, 1178
red MacArthur, 240, 1418
red 19th Street, 180, 1598
red 12th Street, 120, 1718
red West Oakland, 300, 2018
red Embarcadero, 420, 2438
arrive Embarcadero, 0, 2438
Embarcadero AGM Pickup Locker, 120, 2558
```

Path Storage in mongoDB®

BSON doc structure
allows schema flexibility!

- **Problem:** paths are costly to compute, differ in length/route.
- **Solution:** every time a customer transit path is computed, store it in MongoDB
 - MongoDB > RDBMS:
 - JSON-like BSON document storage structure allows for **flexible schema**, RDBMS would require more storage space, be slower and very sparse (all stations/arrivals/lines, few with non-null values in each row)
 - Automatic sharding enables fast horizontal scaling on geospatial data and compound indexing. Enables superior read/write performances in large datasets compared to RDBMS.
- **App integration:** Can utilize total_minutes and projected order delivery time to alert customer of the optimal departure time.
 - MongoDB's ability to handle high loads of read/write queries makes this an optimal platform to store and query data for many customers requesting shortest public transit routes.

```
{
  "customer_id": 4880,
  "haversine_distance": {
    "to_Civic_Center": 0.18,
    "walk_time_to_Civic_Center": 4.31
  },
  "total_cost": 378,
  "total_minutes": 6.30,
  "travel_plan": [
    {"from": "customer_id 4880", "duration": 0, "cost": 0},
    {"depart": "Civic Center", "duration": 60, "cost": 258},
    {"arrive": "Civic Center", "duration": 0, "cost": 258},
    {"Civic Center AGM Pickup Locker", "duration": 120, "cost": 378}
  ]
}
```

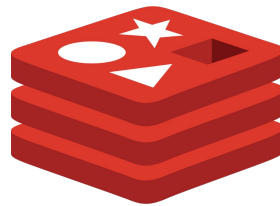
Path Storage in mongoDB®

BSON doc structure
allows schema flexibility!

- **Problem:** paths are costly to compute
- **Solution:** pre-compute path and store it in MongoDB
- **App integration:** Can utilize total_minutes and projected order delivery time to alert customer of the optimal departure time.

```
{
  "customer_id": 4880,
  "haversine_distance": {
    "to_Civic_Center": 0.18,
    "walk_time_to_Civic_Center": 4.31
  },
  "total_cost": 378,
  "total_minutes": 6.30,
  "travel_plan": [
    {"from": "customer_id 4880", "duration": 0, "cost": 0},
    {"depart": "Civic Center", "duration": 60, "cost": 258},
    {"arrive": "Civic Center", "duration": 0, "cost": 258},
    {"Civic Center AGM Pickup Locker", "duration": 120, "cost": 378}
  ]
}
```

Real-time delivery updates with



redis

Pros of Redis:

- **Low-latency data access**
- **Publisher/subscriber messaging** for real-time communication. List ingestion is reliable in cases of connection loss
- **In-memory data store:** ideal for maintaining real-time data



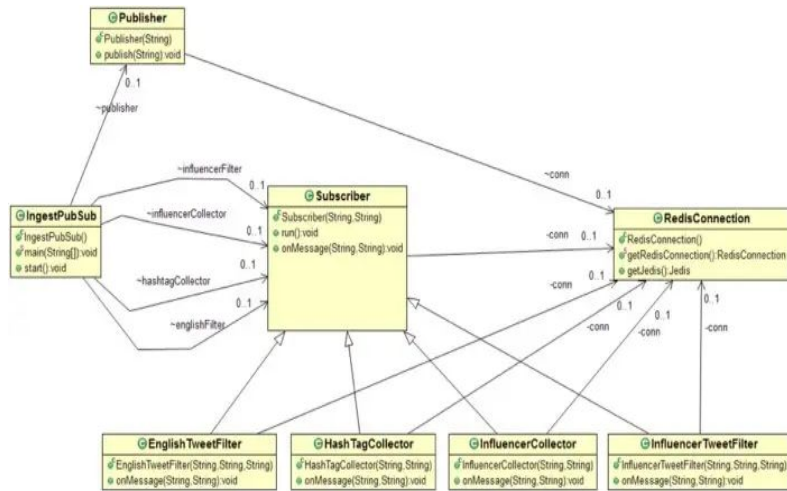
Real-time delivery updates with



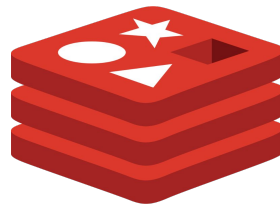
redis

Use Cases:

- **Provide real-time delivery tracking/updates** by notifying customers on changes in order status (placed, shipped, transit), vehicle ETA and pending arrival, using Redis' publisher/subscriber commands.
- **Track real-time parameters** (i.e. average speed, location, fuel/charge level, # of stops) **to inform route augmentation** as needed (fuel level used to determine which fueling/charging stations to visit and re-route appropriately)



Real-time delivery updates with



redis

Use Cases:

- Provide real-time delivery tracking/updates
- Update/augment route

A relational database is not be appropriate because:

- Vertical scaling **doesn't prioritize latency**, and schema rigidity results in **time-consuming alterations**
- ACID transactions can introduce overhead/latency when processing bursts of concurrent transactions (multiple customers updated when common delivery van is en route), which Redis excels at.

