Building a Feature Store for Hypergrowth

Brian Seo, Sr. Software Engineer, Doordash

FEATURE STORE SUMMIT 2022

Organized by HOPSWORKS

Agenda

1. What Caused Our Hypergrowth
2. Architecture
3. Offline Storage
4. Redis + Elasticache and It's Limits
5. CockroachDB as an Online Store

What Drove Our Growth?

Make Things Easy

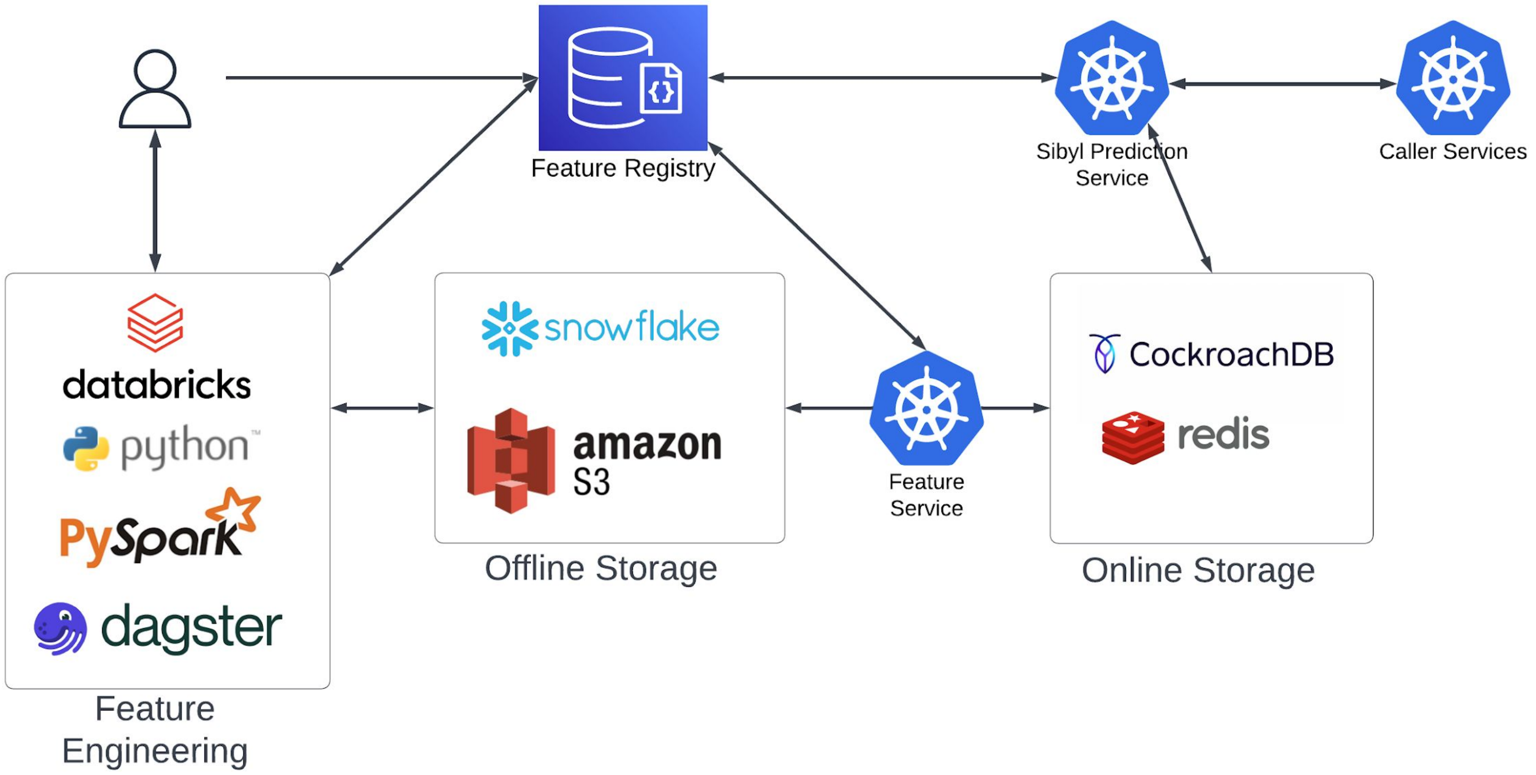
- ➔ Simplified Feature Engineering
- ➔ Guardrails that protect against failure
- ➔ Abstraction of storage concepts

Expansion of Scope

- ➔ Expansion into different verticals: Groceries, Convenience, Alcohol
- ➔ More verticals means more features and a much larger keyspace

Industry Advancements

- ➔ More data available
- ➔ Much simpler to process large quantities of data
- ➔ Large models are easier to train


Feature Registry
Sibyl Prediction Service
Caller Services
databricks
python
PySpark
dagster
Feature Engineering
snowflake
amazon S3
Offline Storage
Feature Service
CockroachDB
redis
Online Storage

Fabricator: Feature Engineering + Storage

Fabricator

A Declarative Framework That Handles

- ❏ Feature Engineering
- ❏ Feature Registry
- ❏ Online / Offline Storage

Fabricator: Simplify Feature Creation

Simple SQL

```yaml
name: store_eta_avg
storage_spec:
  table_name: store_eta_avg
compute_spec:
  upstreams:
    - fact_store_eta
compute_spec:
  sql:
    - >-
      SELECT
      store_id
      , avg(eta) eta_last_20min
      , avg(eta_last_week) eta_last_week
      FROM fact_store_eta
      WHERE
      time = {datetime}
```

PySpark Scripts

```yaml
name: store_eta_avg
storage_spec:
  table_name: store_eta_avg
compute_spec:
  upstreams:
    - fact_store_eta
  databricks_spec:
    filepath: eta_calculation_script.py
```

Simplify Deployment

```yaml
feature_groups:
  - source: store_eta_avg
    features:
      - eta_last_20min
      - eta_last_week
    materialize_spec:
      sinks:
        - some_redis_cluster
```

Read more at: https://doordash.engineering/2022/01/11/introducing-fabricator-a-declarative-feature-engineering-framework/

Fabricator: Offline Feature Storage + Fetching

```
features = get_features(
    ['eta_last_week',
    'eta_last_20_min',...])
```

→ Retrievies Protobuff Messages Containing The Storage + Entity Relations

→ amazon S3 — Fetches from S3

→ PySpark — Automatically Joins fetched features based on retrieved dates and entity relationships

Online Storage: Redis → CRDB

Guiding Principles For Online Storage

1. Availability + Resilience to Failure
2. Low latency on retrieval of 1000s of keys
3. Cost Effectiveness

Redis + Elasticache

Why Use Redis

- Extremely Fast, can support a much higher throughput than other solutions
- Can scale horizontally and vertically
- Resilient to failure
- Can support an extremely high QPS

DB	Write latency	Read heavy latency (95% batch read, 5% update)			Read only latency (100% batch reads)		
	10k rows (s)	Avg (ms)	P95 (ms)	P99 (ms)	Avg (ms)	P95 (ms)	P99 (ms)
Redis (3 masters)	5	1.9	2.4	4.0	1.9	2.3	4.3
CockroachDB (3 nodes behind a lb)	1.127	4.7	6.1	8.8	5.9	7.8	10.8
ScyllaDB (3 nodes)	10.8	16.9	22	28.5	17	22	28
Cassandra (3 nodes)	18.8	23.5	30	38	23.6	32	43.5
YugabyteDB (3 nodes)	25.7	43.2	50.3	54.2	33.4	38.3	41.5

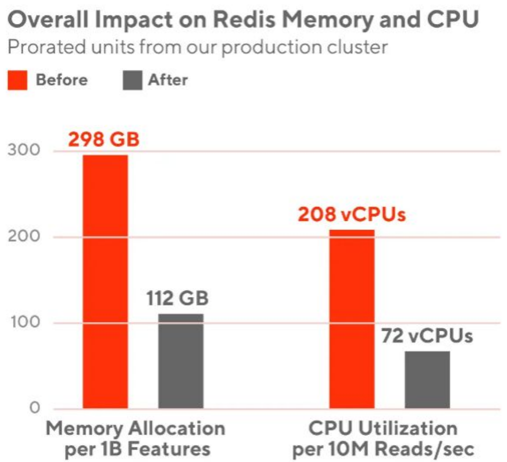
Redis + Elasticache

Going from KV to Hashmap Storage

store_id	eta_last_20min	eta_last_week
1	5.00	7.00
2	20.00	21.00

```
> SET(CONCAT(id + "eta_last_20_min"), 5.00)
> EXPIRE(CONCAT(id + "eta_last_20_min"), 3600)
> SET(CONCAT(id, "eta_last_week", 7.00)
> EXPIRE(CONCAT(id, "eta_last_week"), 3600)
```

```
> HSET(id, "eta_last_20_min", 5.00)
> HSET(id, "eta_last_week", 7.00)
> EXPIRE(id, 3600)
```



Difficult to Maintain at Scale

Scaling Operations are Time Consuming

Scaling using Elasticache functions is not practical

1. Scaling up a cluster can take upwards of 12 hours depending on the size and utilization
2. Scaling down can take even longer
3. Drops in performance and availability are significant during scaling operations and behavior is largely dependent on the redis client

No Downtime Scaling Operation (~3-10 hours)

1. Spin up a cluster from backup
2. Backfill all features uploaded since last backup
3. Redirect traffic from services to new cluster
4. Remove old cluster

Difficult to Maintain at Scale

Storage Is Expensive

- ➔ Hash keys are expensive to create and will increase storage usage at a much higher rate
- ➔ Embeddings are exceptionally expensive to store
- ➔ Provisioning storage predictably is difficult
- ➔ Eviction behavior needs to be aggressive to keep costs down
- ➔ Can lose features if upstream tables on not reliably maintained

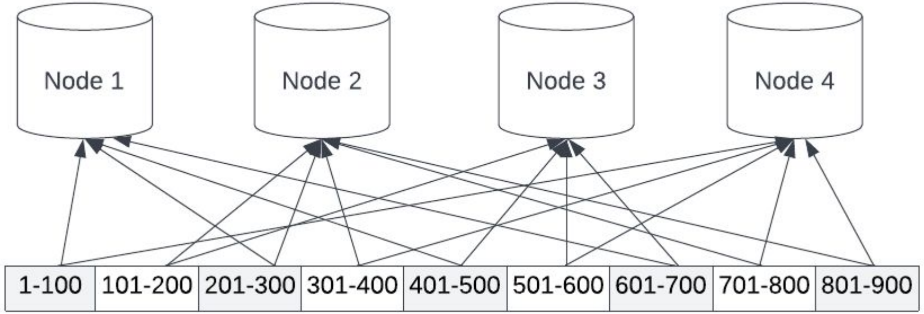
Cockroach DB As An Alternative

Why Cockroach DB?

1. Best performer in last round of benchmarks
2. Maintenance operations are virtually seamless
3. Can be utilized as multi-region
4. Can autoscale and automatically respond to skewed usage
5. Near instantaneous recovery from node failures
6. Supports a wide range of indexing schemes
7. Uses PostgresQL

What Makes it Different

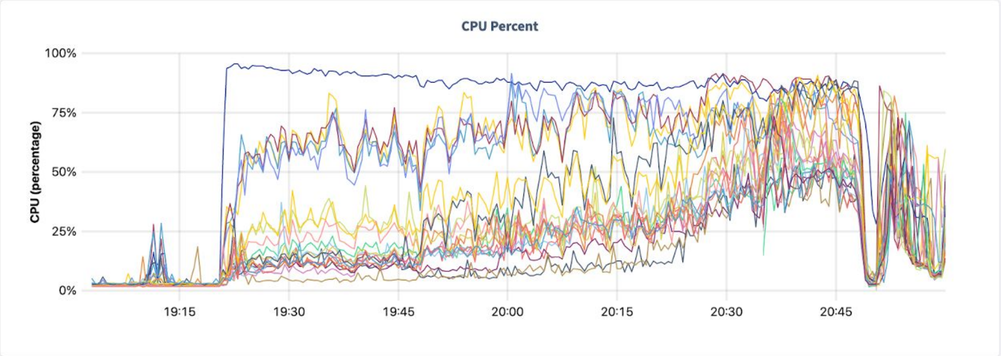
id (PK)	feature_name (PK)	Value	range_id	node_id
1	feat_1		1	5
1	feat_2		1	5
...	...	...	...	...
30	feat_1		2	3
30	feat_2		2	3
30	feat_3		2	3
....	...	...	...	...
10001	feat_3		100	2
...	...	...	...	...
21412	feat_2		500	1



Sequential keys are stored in blocks called ranges that allow similar values to be colocated on the same node

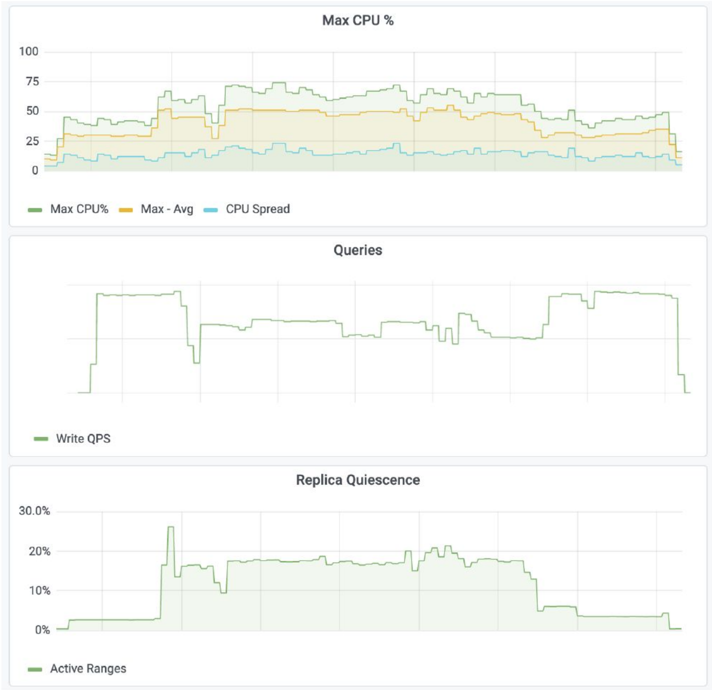
Growing Pains

Tables All Start on a Single Node



FEATURE STORE
SUMMIT
2022

More Keys == More CPU



Upload Throughput Optimization

1. Sort data before uploading
2. Batch rows into groups
3. Shuffle Batches
4. Update the entire row, not a subset

FEATURE STORE SUMMIT 2022

Improve Read / Write By Optimizing For Key Density

ETL Tables

store_id	eta_feat_1	eta_feature_2
5	3.0	2.2
7	10.1	20.1

store_id	x_feat_1
5	3
7	8

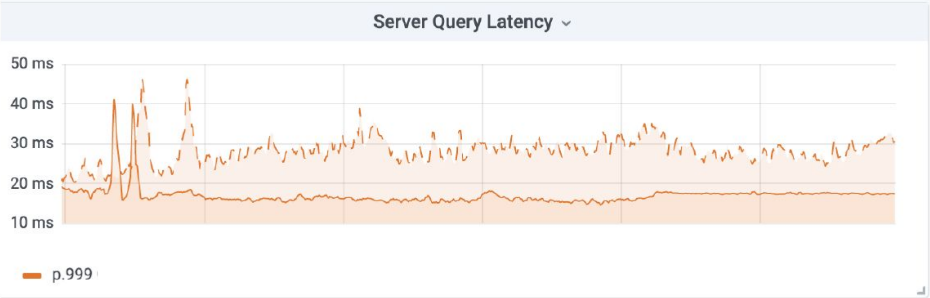
KV Format

id	feature_name	feature_value
store_id=5	eta_feat_1	3.0
store_id=5	eta_feat_2	2.2
store_id=5	x_feat_1	3
store_id=7	eta_feat_1	10.1
store_id=7	eta_feat_2	20.1
store_id=7	x_feat_1	8

Dense Format

id	source	feature_value_map
store_id=5	eta_features	{eta_feat_1: 2.0, eta_feat_2: 2.2}
store_id=5	x_features	{x_feat_1: 3}
store_id=7	eta_features	{eta_feat_1: 10.1, eta_feat_2: 20.1}
store_id=7	x_features	{x_feat_1: 8}

50% Improvement in p999s



FEATURE STORE SUMMIT 2022

Looking Forward

1. Creating Features on Demand
2. Getting smarter about the values that should be updated
3. Optimizing storage formats for a given call pattern

Questions?

Thank You!