PG as Recommendation Engine

instacart

Ankit Mittal and Jon Phillips

Topics

Summary

- About Us
- Postgres as a recommendation engine
- Ingestion to support embeddings
- How to maintain stability with high ingestion volume

instacart

Instacart is the leading grocery technology company in North America

1400+
Retail partners across the US and Canada

600K+
Instacart shoppers

95%+
Household coverage in US, CA

Grocery
and Beyond

Product Retrieval Platform Team

- Ownership of infrastructure that powers all search and product retrieval
- Operations, uptime and reliability of ~250 self hosted PG hosts
- Building of product retrieval read client and ingestion system

instacart

The Origins

PGCon2012 - Fin

PGCon 2012
The PostgreSQL Conference

Finding Similar

Effective similarity search in database

Finding similar objects is an ubiquitous task in day-to-day activity of developers of informational services. We present PostgreSQL extension, which provides an effective way to find similar objects in database, as well as several usage examples. The extension provides several methods to calculate sets similarity and similarity operator with indexing support on the base of GiST and GIN frameworks.

Similarity search in large databases is an important issue in nowadays informational services, such as recommender systems. Naive implementation is slow and resource consuming. We developed PostgreSQL extension, called smlar, which provides several methods to calculate sets similarity (all built-in data types supported), similarity operator with indexing support on the base of GiST and GIN frameworks. Sets similarity means, that smlar isn't about content similarity (it doesn't interested in the nature of objects), but it's about similarity of sets. One example is a recommender system, which produces a list of recommendations based on collaborative and/or content filtering (Amazon is one of the most popular electronic commerce company, which provides recommendations, based on item-item similarity). Content filtering utilizes a set of discrete metadata of an object to build recommendation list of additional objects with similar properties, while collaborative filtering uses information about user's past behaviour and similar decisions made by other users, to predict objects that the user may have interest in. Smlar extension was developed in mind with collaborative filtering. It provides several methods to compute similarity between sets: jaccard, cosine and tfidf. Experiments with generated and real data sets show considerable advantage of using smlar extension in compare with brute-force approach.

Attached files

- Effective similarity search in PostgreSQL (application/pdf - 482.5 KB)

SPEAKERS	
	Oleg Bartunov
	Teodor Sigaev
SCHEDULE	
Day	**Talks - 1 Thursday 2012-05**
Room	**MRT 219**
Start time	**13:00**
Duration	**01:00**
INFO	
ID	**443**
Event type	**Lecture**
Track	**Hacking**
Language used for presentation	**English**

Embeddings

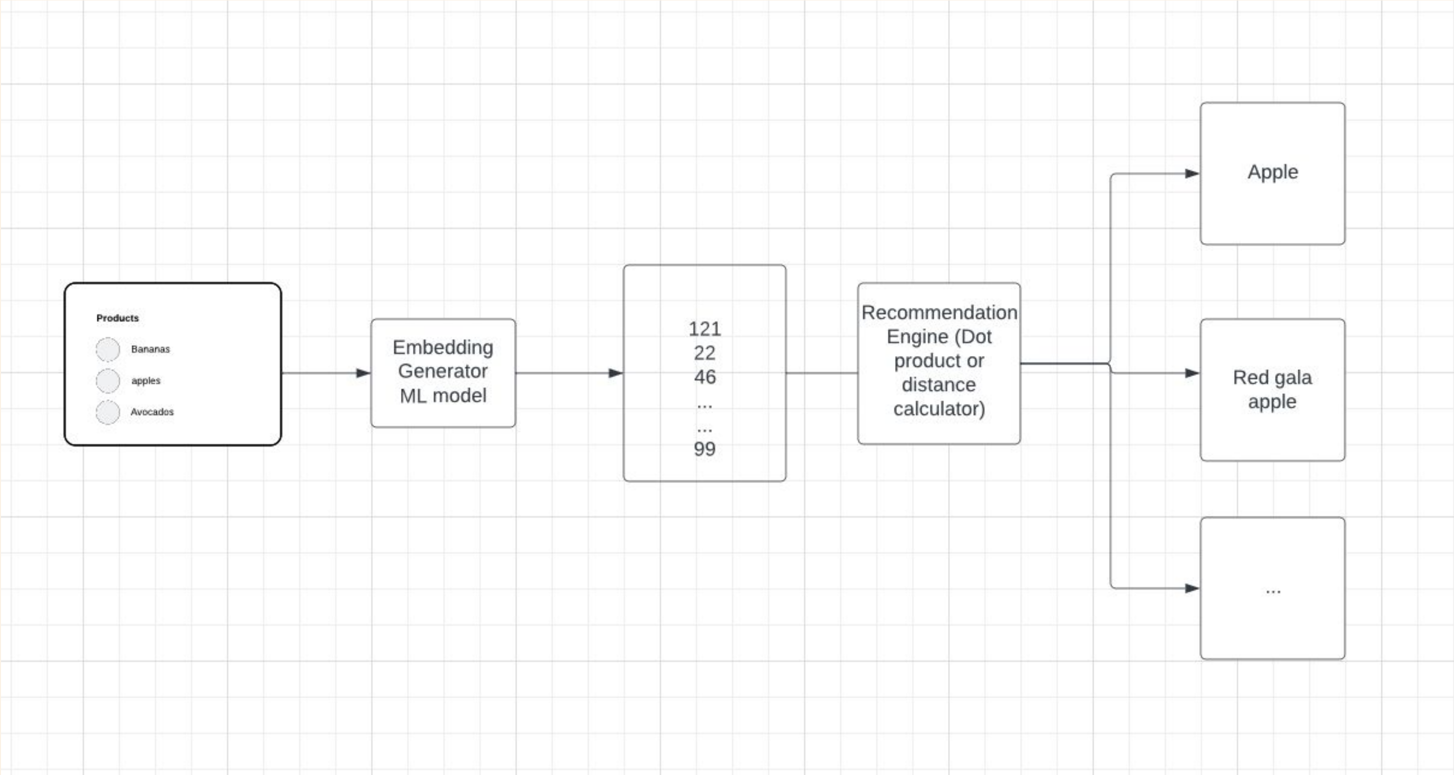
An embedding is vector (of floats and ints) representation of any real world object. These could of embeddings of words, phrases, items, songs, videos etc.

A machine learning model converts objects into embeddings.

Inference problems can be converted into a similarity search in embeddings space.

Embeddings close to each other in this hyper dimensional embeddings space are similar to each other.

Similarity Search



There and Back Again

Home Posts Talks About

VECTORS ARE THE NEW JSON IN POSTGRESQL

Mon, Jun 26, 2023 10-minute read

Vectors are the new JSON.

That in itself is an interesting statement, given vectors are a well-studied mathematical structure, and JSON is a data interchange format. And yet in the world of data storage and retrieval, both of these data representations have become the lingua franca of their domains and are either essential, or soon-to-be-essential, ingredients in modern application development. And if current trends continue (I think they will), vectors will be as crucial as JSON is for building applications.

Generative AI and all the buzz around it has caused developers to look for convenient ways to store and run queries against the outputs of these systems, with PostgreSQL being a natural choice for a lot of reasons. But even with the hype around generative AI, this is not a new data pattern. Vectors, as a mathematical concept, have been around for hundreds of years. Machine learning has over a half-century worth of research. The array — the fundamental data structure for a vector — is taught in most introductory computer science classes. Even PostgreSQL has had support for vector operations for over 20 years (more on that later)!

So, what is new? It's the *accessibility* of these AI/ML algorithms and how easy it is to represent some "real world" structure (text, images, video) as a vector and store it for some future use by an application. And again, while folks may point to the fact it's not new to store the output of these systems ("embeddings") in data storage systems, the emergent pattern is the *accessibility* of being able to query and return this data in near real-time in almost any application.

What does this have to do with PostgreSQL? Everything! Efficient storage and retrieval of a data type used in a common pattern greatly simplifies app development, lets people to keep their related data in the same place, and can work with existing tooling. We saw this with

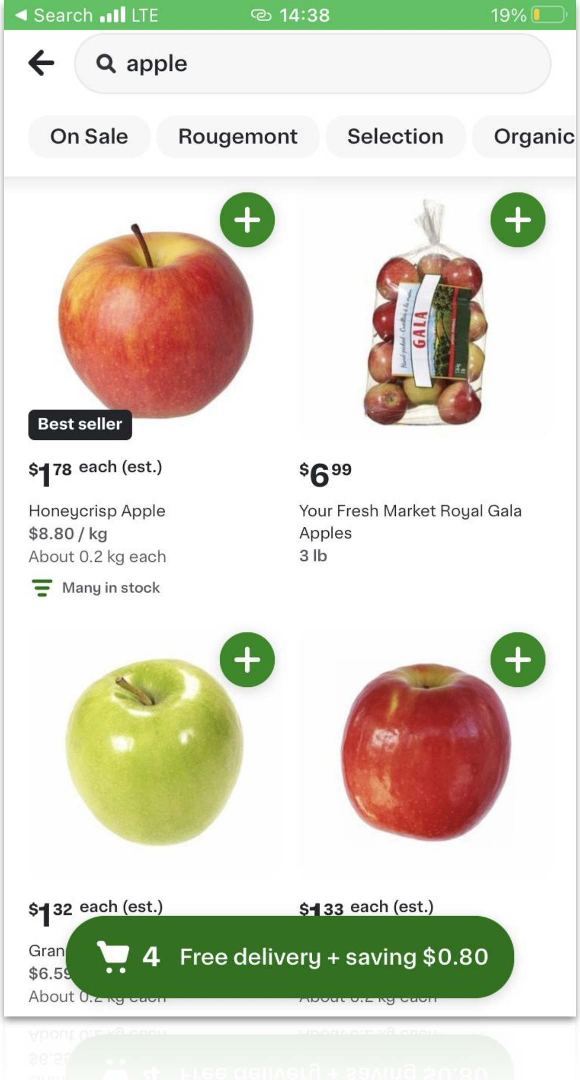
PG as Recommendation Engine

instacart

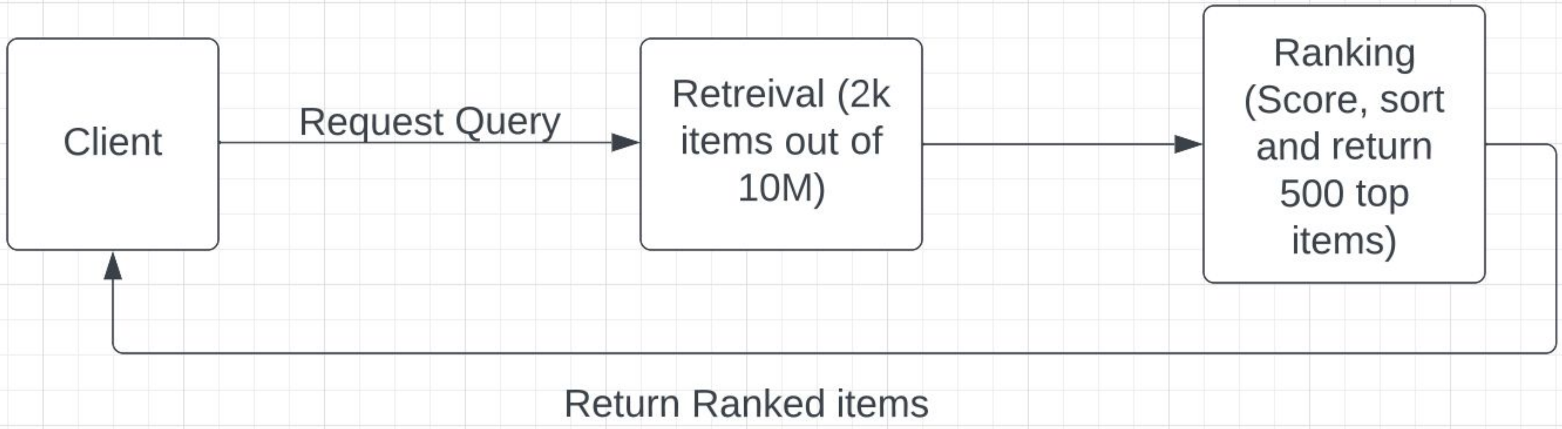
In production since 2019

Search Architecture in One Line

Whenever a search command is issued on the storefront, a single postgres query uses tsvector to perform keyword search and an embedding based personalization ranker. *



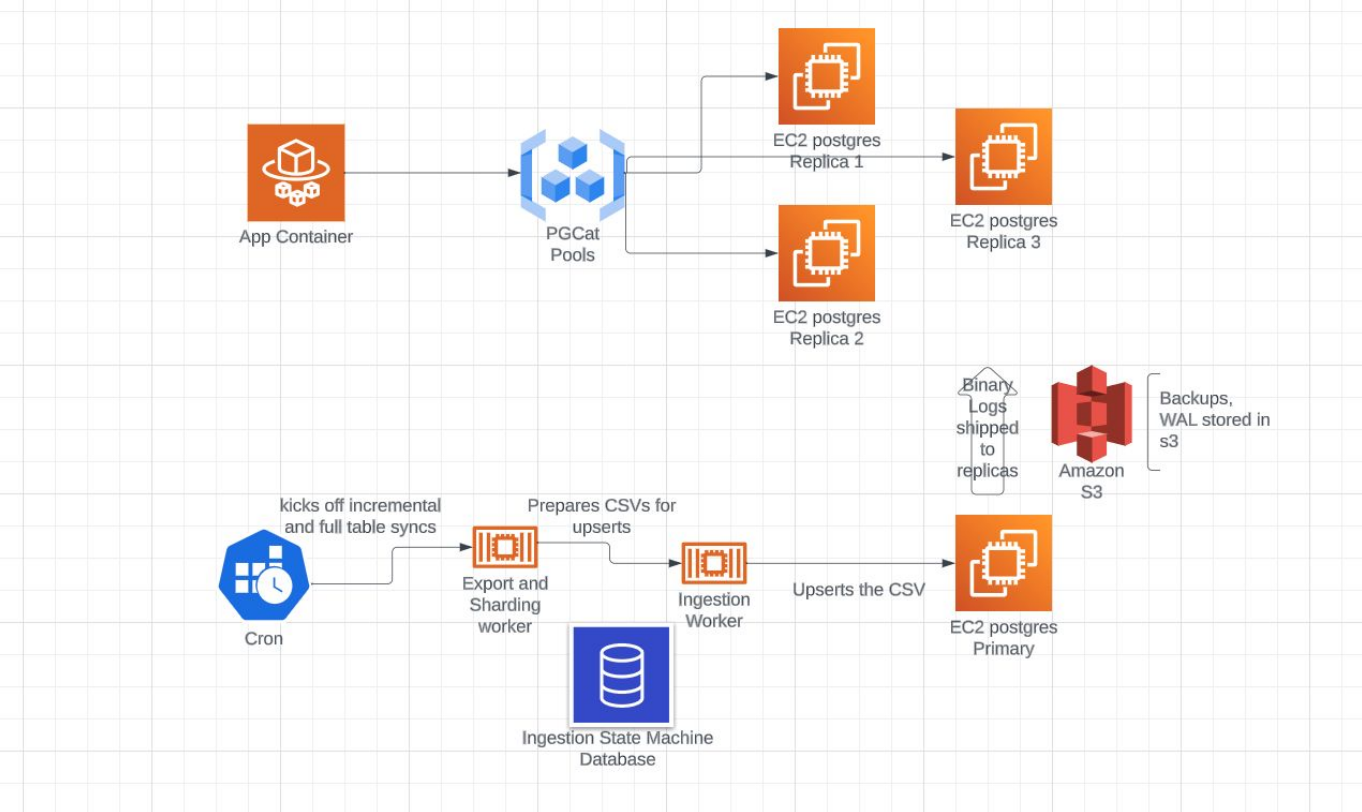
Search Architecture



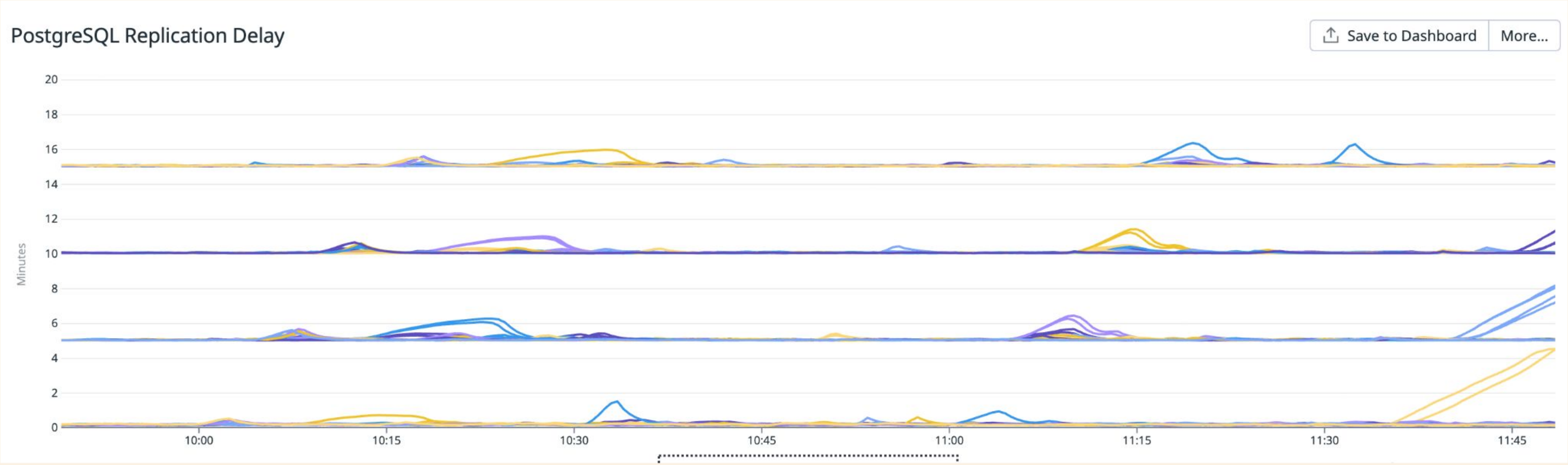
Product Retrieval Cluster

- The cluster is designed to be a write ahead cache*
  - Clients only have read only access
  - Writes are written by specific workers pipelining the row upserts from source of truth
  - Replica sets have staggered replication lag (0, 5, 10) minutes. Giving our cluster an eventually consistent flavour
  - Local NVMes as disk, high shared buffers usage
- Replica is never promoted, handles primary loss by serving stale data while primary is rebuilt

Topology Diagram



Staggered Replica Lag?



Staggered Replica Lag?

- Migrations / DDL locks
  - A mistake, bad migration that grabs locks for long
- Vacuums on certain pg-catalog tables would lock them
- recovery_min_apply_delay

Outside Postgres

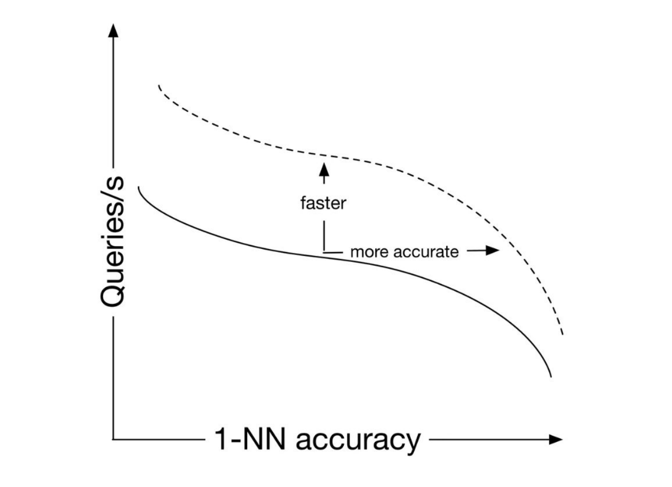
- Training
- Parameter and Hyperparameter tuning of ML models
- Combining Embedding-based retrieved candidates and keyword based candidates
- Query understanding

Within Postgres

- Indexing trained model via MERGE-like command
- TSVector keyword based search
- Dot product (KNN) of user and product embeddings
- Ranking for both Embedding-based and keyword based candidates based on dot product scores
- Joins for inventory availability, CTRs and many other ML-generated scorings for ranking

K–nearest neighbour vs Approximate Nearest Neighbor

- ANN is an approximate algorithm that trades offs accuracy for speed
- ANN latency grows slowly, needed for similarity search >1k records
- Consequently ranking already retrieved search (100–500 items) set according to personalization embeddings can be done by KNN



KNN Extension

```
Datum dot_product_c(PG_FUNCTION_ARGS)
{
    ArrayType *input1, *input2;
    float4 *a1, *a2;
    int len1, len2, len;
    float4 result = 0.0;

    input1 = PG_GETARG_ARRAYTYPE_P(0);
    input2 = PG_GETARG_ARRAYTYPE_P(1);
    a1 = (float4 *) ARR_DATA_PTR(input1);
    a2 = (float4 *) ARR_DATA_PTR(input2);
    len1 = ARR_DIMS(input1)[0];
    len2 = ARR_DIMS(input2)[0];
    len = len1 < len2 ? len1 : len2;

    for (int i = 0; i < len; i++) {
        result += a1[i] * a2[i];
    }

    PG_RETURN_FLOAT4(result);
}
```

Why

Model Update Speed

Development Velocity

Minimal Data Transfers

CTRs and Continuous Improvement

More Flexibility

Availability Machine

Why

- Five nines reliability
- Faster and more reliable data pipeline for retailer information
- Much better p99 latency than our previous architecture
    - 80% reduction per API call
    - Reduced API calls due to availability joins

Dealing with dead tuples and herding cats

Ingestion

Some Numbers

15 Billion Writes Per Day

How do we ingest 15b records a day?

Two strategies

1. Shard the data
2. Copy + on-conflict bulk upserts

Sharding Strategy

Store Front Sharding
Region Sharding
Omni Sharding

Sharding Strategy

Each strategy allows us to isolate primaries and group data according to query patterns

Store Front
Prices
Availability
Sale information

Region
Cross-retailer search
Aggregate searches

Omni
Isolated non-joined tables
Shard key and mapping lookups

Defining Sharding Strategies

Model
MetaData

Sharding
Rules

```
models[Item] = &CatalogStoreModel{
  tableName:   "items",
  description: "items table for housing location specific data including price and availability",
  owner:       "catalog",
  opsgeniePoc: "catalog",
  shardRouting: map[ClusterType]ShardingStrategy{
    ItemCluster:   retailerClusteredV1ByInventoryAreaIdSharding,
    LegacyCluster: legacyUniformSharding,
  },
}
```

Sharding Strategies Continued

- **Sharding strategies are immutable per cluster**
- **Shards must receive enough traffic to keep buffers warm**

How do we write sharded data?

1. Teams write CSVs to S3
2. Each file is streamed and split based on the sharding strategies defined for that model
3. Then each split file is written to a postgres instance

Copy + On Conflict Upserts

- Check if unlogged_table exists (create if it doesn't exist)
- Stream contents of s3 csv file to unlogged_tabe
- Insert contents of unlogged_table to actual table
- Delete rows in unlogged_table
- On errors, individually upsert directly to table row by row

```
COPY #{unlogged_table_name}
(#{columns.join(', ')})
          FROM STDIN
           WITH (FORMAT csv,
 HEADER false, NULL '\\N',
 FORCE_NULL (#{columns.join(',
 ')}));
```

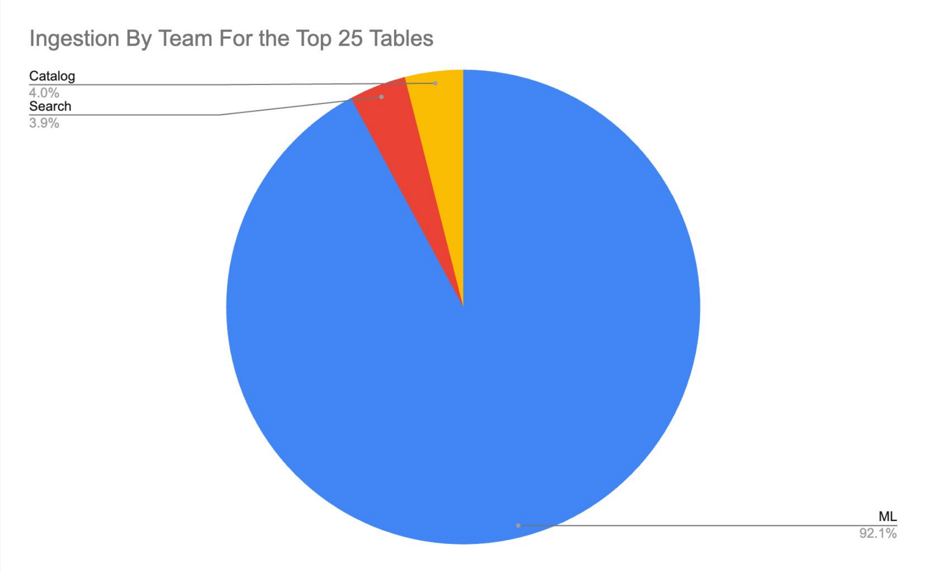
```
    INSERT INTO #{table_name}
(#{columns.join(',')})
              SELECT DISTINCT ON
(#{import_key})
columns.join(',')}
              FROM
#{unlogged_table}

#{order_by_incremental(model)}
              ON CONFLICT
(#{import_key}) DO UPDATE SET
              #{columns}
              WHERE
(#{columns.map { |c|
"#{table_name}.#{c} IS DISTINCT
FROM excluded.#{c}" }.join(' OR
')})

 #{where_incremental_is_newer}
```

Upsert Table Gotchas With High Volume

Preserving MVCC
Unlogged tables and dead tuples

Who owns these writes?

Catalog 4%
ML 96%



Why does this matter?

Why does this matter?

- Teams have different ingestion requirements
- Ingestion needs to be prioritized by team based and the importance of the data
- Batches have drastically different load characteristics

They all have one thing in common

Stability of the front end services must be maintained

Dead Tuples

Table Bloat and DB performance

instacart

If you stack two lasagnas, you have one tall lasagna

Dead tuple risks

- Disk usage
- Autovacuum can not keep up with new writes
- Ingestion throughput goes down
- Slows down sequential scans
- Causes poor query plans and slows down queries

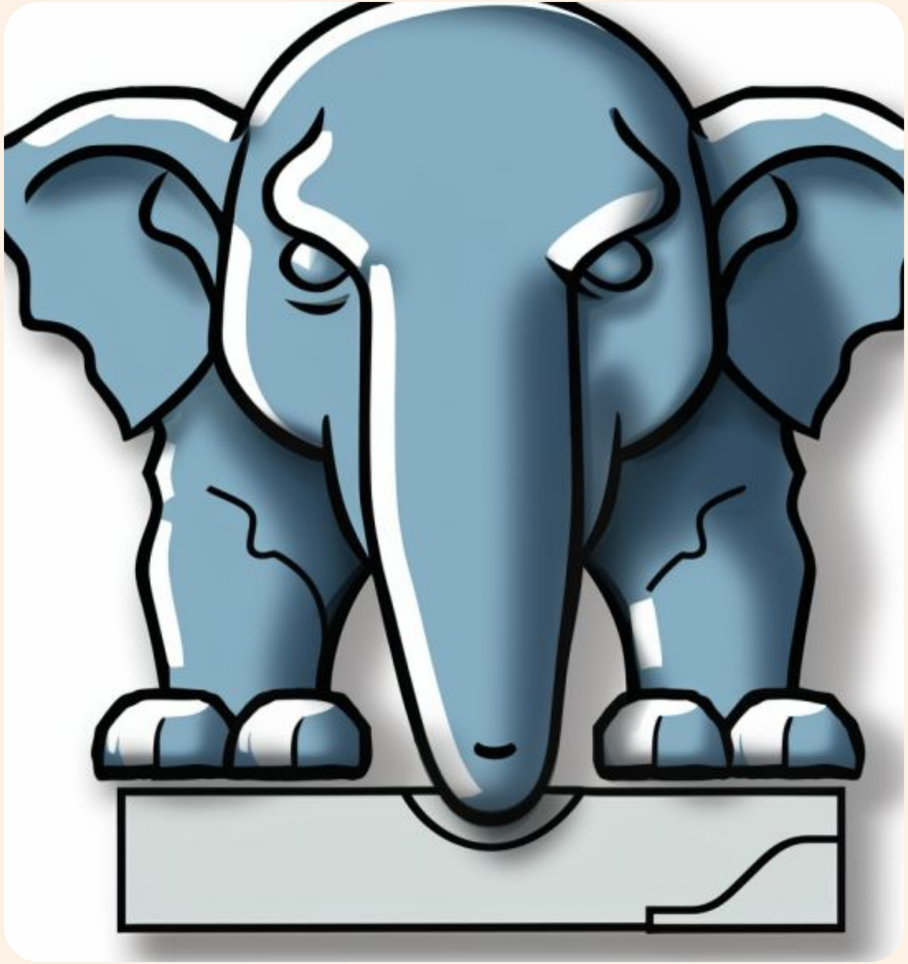
Addressing Bloat

Autovacuum
pg_repack

instacart

Power Repack

How we tune dead tuple cleanup



instacart

What is power repack?

Power repack is our pg_repack orchestrator. It keeps tabs on dead tuples and kicks off pg_repacks. It is aware of table-specific overrides.

Power Repack

Teams define rules that selectively prune stale content.

Power Repack’s extended capabilities

```yaml
where_clause:
  items_availabilities: "updated_at > now() - Interval '14 days'"
  items: "retailer_product_experiment_variant_id = -1"
  retailer_products_cpgs: "has_deal = true OR updated_at >= NOW() - INTERVAL '14 days'"
```

Issues with repacking

Things to keep in mind

- There must be enough headroom at all times to run repack (based on the largest table and its indices)
- If possible, pause ingestion to the repacking table. High throughput ingestion can delay repacks by several hours because data is effectively written twice. Pausing ingestion reduces the time to minutes.

Postgres as complex objects engine

Postgres as a place to

- Store complex objects
- Compare complex objects

Michael Stonebaker’s original postgres theis. This was one of the fundamental design goals of Postgres

Abstract

This paper presents the preliminary design of a new database management system, called POSTGRES, that is the successor to the INGRES relational database system The main design goals of the new system are to

1) provide better support for complex objects,
2) provide user extendibility for data types, operators and access methods,
3) provide facilities for active databases (i e , alerters and triggers) and inferencing including forward- and backward-chaining,
4) simplify the DBMS code for crash recovery,
5) produce a design that can take advantage of optical disks, workstations composed of multiple tightly-coupled processors, and custom designed VLSI chips, and
6) make as few changes as possible (preferably none) to the relational model

The second goal for POSTGRES is to make it easier to extend the DBMS so that it can be used in new application domains A conventional DBMS has a small set of built-in data types and access methods Many applications require specialized data types (e g , geometic data types for CAD/CAM or a latitude and longitude position data type for mapping applications) While these data types can be simulated on the built-in data types, the resulting queries are verbose and confusing and the performance can be poor A simple example using boxes is presented elsewhere [STON86] Such applications would be best served by the ability to add new data types and

Thank you!

Oct 3 2023