# Couchbase server

...

G03

# Overview of the technology

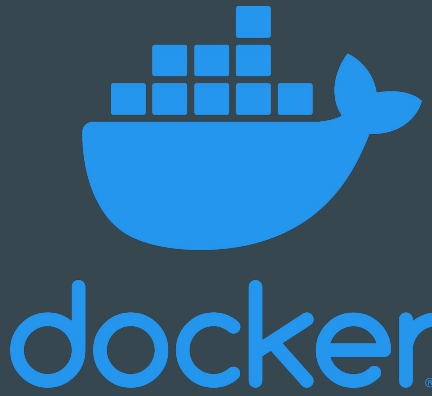
# Technology

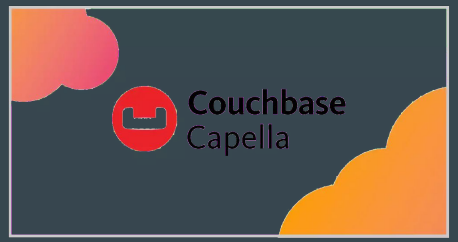
- Document based.
- Simultaneous Key-value architecture.
- Multiple data patterns (flexibility).
- Data storage hierarchy and diversity.
- Focus on query optimization and overall speed.
- Uses an SQL like language, N1QL.
- Facilitates data replication through node creation (Database Change Protocol and XDCR).
- Satisfies Consistency and Partitioning.
- Security and encryption.

# Installation and administration overview

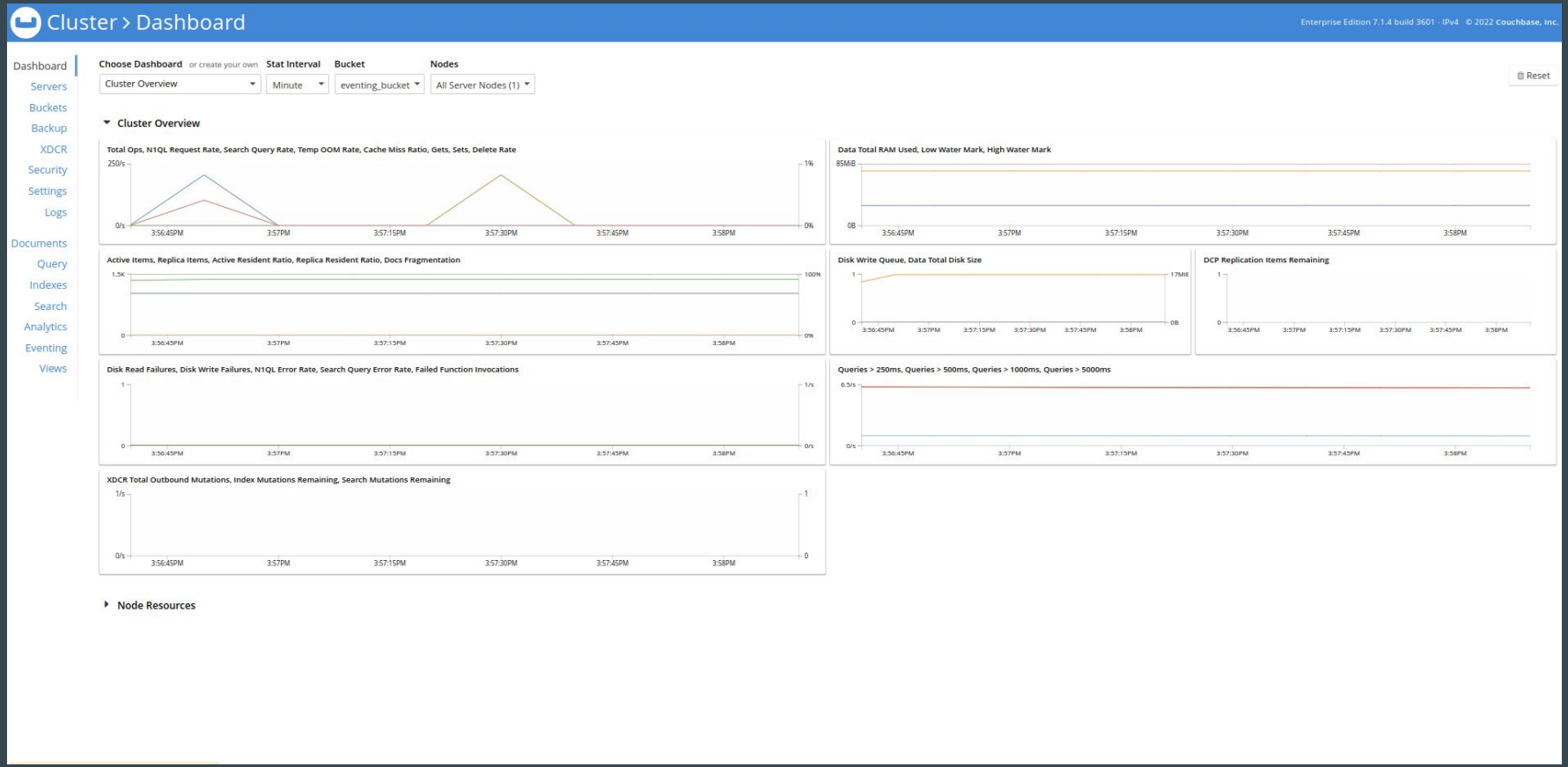
# Installation and administration overview

- Enterprise and Community edition.
- Manual installation on local machines (Linux, Windows, Mac).
- Official docker image.
- Online Capella service (paid).
- Interaction through the Web UI dashboard.
- cURL requests.
- User authentication (credentials or TLS certificates).
- Special RBAC roles.



# Demonstration



# Data model and data operations

# Data model

- Data stored in cluster nodes.
- Data hierarchy: Buckets, Scopes, Collections.
- Documents are JSON-like objects with unique key, that may have other JSON-like objects nested in them.
- Documents of a collection can have different structures.
- *vBuckets* data shards/partitions.

# SQL++ queries

- Follow the SQL syntax as closely as possible.
- Retrieval queries select information from a given collection and can specify a set of conditions.
- Common expected keywords: *SELECT, UPDATE/UPSERT, DELETE, DROP*…
- *JOINs* are supported, as well as *EXECUTE* queries/functions.
- Subqueries are valid.
- Some additions are made in order to switch to a NoSQL context.
- *UNNEST/NEST* features, to uncover nested objects inside documents.
- Array operations to mutate list fields (*FLATTEN, DISTINCT, AGGR, MAX, INTERSECT…)*
- Aggregate operations: *GROUPBY, COUNT, AVG, VARIANCE, STDDEV*…

# Key-Value alternative

- Takes advantage of the required ID in each document.
- Operations: get, insert, remove.
- Bulk operations are language dependent or have a multi implementation.
- Handled by the KV engine (remnants of Memcached).
- Uses multithreading and an append-only disk storage.
- Has in memory caching that provides low latency R/W, and streams through the DCP mechanism.
- Subdocument queries: refer to specific fields inside a document. Useful for partial modifications/reads since only relevant parts are retrieved or sent.
- e.g *mutateIn, arrayAppend, increment, lookupInSpec, get, remove, upsert*…

# Eventing

- Resembles triggers and procedures from RDB.
- Mechanisms that activates javascript callbacks, while listening to database region.
- Triggers upon delete (*onDelete*) or update/creation (*onUpdate*).
- Completes smaller tasks, or more complex tasks (cURL requests, issuing timers).
- May have a function scope and a log storage bucket.
- Operations with events include: deploy/undeploy and pause/resume.

# Transactions

- Ensure ACID properties across multiple operations that can affect multiple documents.
- Operations all succeed or all fail because changes are all committed at the same time. Changes are not visible until transactions are committed.
- Does not enforce referential integrity since it is a NoSQL Document-oriented database.
- There is a possibility to configure if the transaction is accepted when the majority of replicas reply positively to it or if data needs to be stored on disk of the main Bucket or even in the disk of all replicas (durability settings).

# Analytics

- The Analytics service allows performing complex queries directly on the data (no need for ETL).
- Uses MPP - massive parallel processing.
- Can perform analytics in near real time.
- Uses a copy of the data that is automatically updated.
- Appears to be Couchbase’s replacement of for their MapReduceView.

# Features highlight

# Indexes

- Allow for powerful querying options such as the scoring of results, Fuzzy matches, regular expression searches, wildcard searches, and geospatial queries, among many others.
- Primary indexes are based on the unique key of every item in a specified collection.
- Global Secondary Indexes can be created on any fields or expressions necessary.
- Other types Full-Text-Indexes, Geospatial Indexes, Array Indexes and Analytics Indexes.

# Data rebalancing and cluster organization

- Best selling point of the product: ensuring high availability, fault tolerance and data durability.
- Intra-cluster (replicates and syncs data across cluster nodes)
  - Up to 3 replicas per node.
  - Even data distribution and conflict resolution system, upon simultaneous document access.
- Cross Data Center Replication (bucket or collections)
  - Replicate active data to geographically distributed data centers (for recovery or availability).
  - Highly customizable and has robustness and security options.
  - Is triggered by CURL or REST requests.
- Both can occur simultaneously.
- Automatic failover mechanisms that reorganize the cluster and promote replicas to active nodes (regain the healthy state of the cluster).

# Consistency features

- Replication plays an important role in maintaining data availability, preventing data loss, and ensuring the overall reliability of the system.
- Couchbase is considered a CP system that can be configured to favor AP.
- Couchbase uses an eventual consistency model by default - data updates are propagated asynchronously to all replicas within the cluster, and therefore replicas may temporarily have different versions of the data until the updates are fully propagated and replicated.
- Couchbase allows the user to configure specific groups of nodes within a cluster to provide strong consistency, ensuring that read and write operations are fully synchronized within a said group.
- The AS OF SYSTEM TIME clause allows for stale reads if performance over consistency is desired for a particular query.

# The Backup Service

- Selection of which data is stored
- Configuration of  how often backups are performed.
- Choice of which buckets have backups and if the backups are done incrementally or not.

# APIs and libraries

# APIs and libraries

- SDK support for multiple languages: .NET, C(++), Java, Kotlin, PHP, Ruby…
- Convenient adaptations.
- Interoperability with ElasticSearch, Tableau and Kafka.
- REST API, that supports cURL requests.

```
const product = cluster.bucket("server").scope("store").collection("products").get("0307141985");
```

```
        await db.getCluster().transactions().run(async (ctx) => {
            const user = await ctx.get(user_collection, user_id);

            if (!user) throw new Error("User not found");

            const already_reviewed = user.content.products_reviews_pairs.some(
                (pair) => pair.product_id === productId);

            if (already_reviewed) throw new Error("User already reviewed the product");

            const new_user = {
                ...user.content,
                products_reviews_pairs : user.content.products_reviews_pairs.concat(
                    [{ product_id: productId, review_id: new_review_id }])
            }
            await ctx.replace(user, new_user);

        }).then((result) => {...}).catch((err) => {...});
```

```
const query = "SELECT s.store_id AS store_id,
               s.name AS store_name,
               s.location AS store_location,
               s.contact AS store_contact
               FROM server.store.stores AS s
               WHERE ANY item IN store_items
                 SATISFIES item.product_id = product_id";

const scope =  cluster.bucket("server").scope("store");

await scope.query(query,
          (err, result) => err ? err : result)
  .then((result) => {
     ...
  })
    .catch((err) => {
     ...
  });
```

```
 const qp = couchbase.SearchQuery.disjuncts(
            couchbase.SearchQuery.match(search_query).field("product_category"),
            couchbase.SearchQuery.match(search_query).field("product_title")
         );
const results = await clu.searchQuery(indexName, qp).then(...).catch(...);
```

# Overview of the prototype

# Overview of the prototype

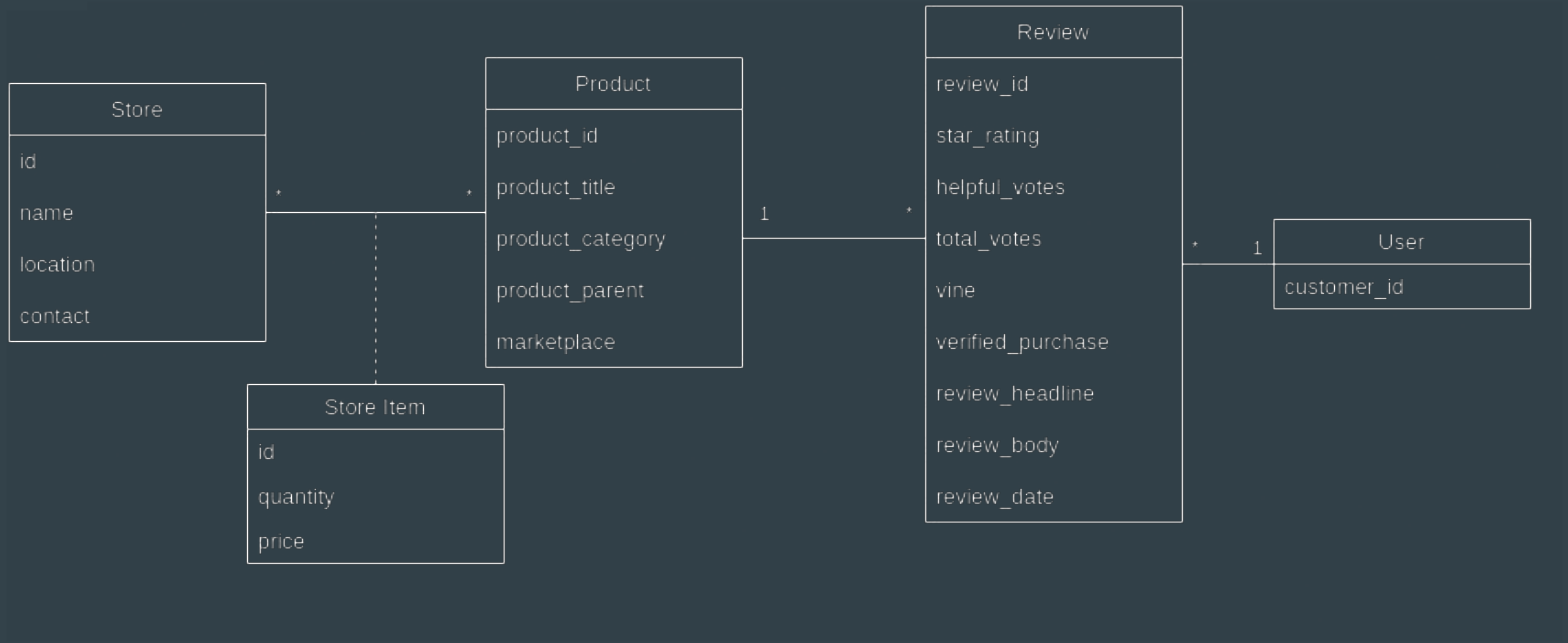
Objective: Create a store management system, that allows a customer to easily find products that might be of interest, upon selecting a variety of filters and evaluating their reviews. A store owner could also checkup up on inventory and apply discounts.

Main pages:

- Product, Review and Store listing.
- Product.
- User profile.

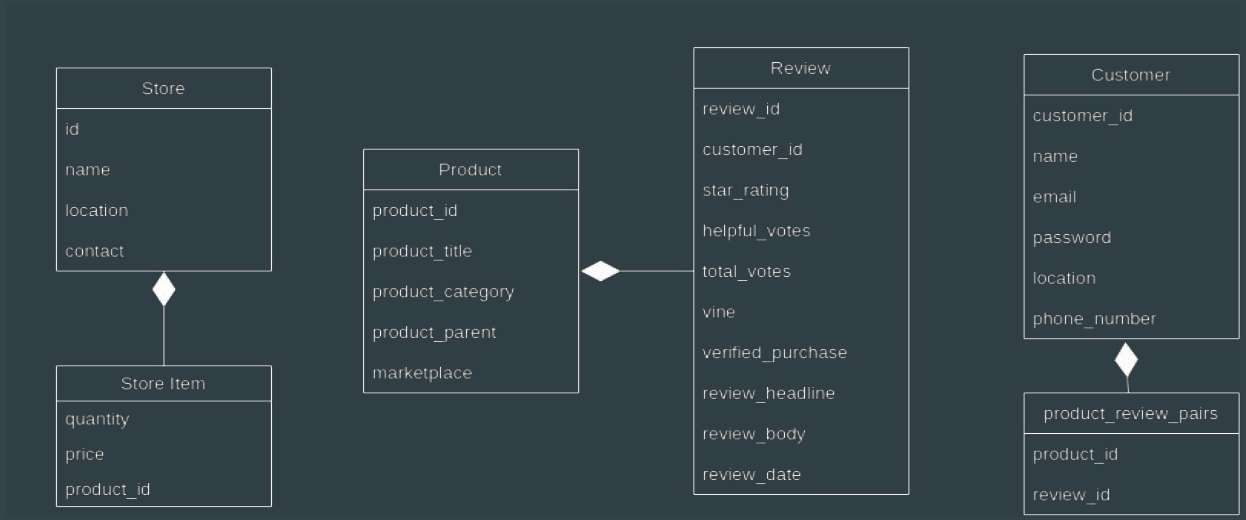
# Models of the prototype

# Conceptual data model of the prototype



# Physical data model of the prototype

A single node/cluster. Server bucket, in which there exists a store scope that keeps the collections products, users and stores.



# Implemented Features

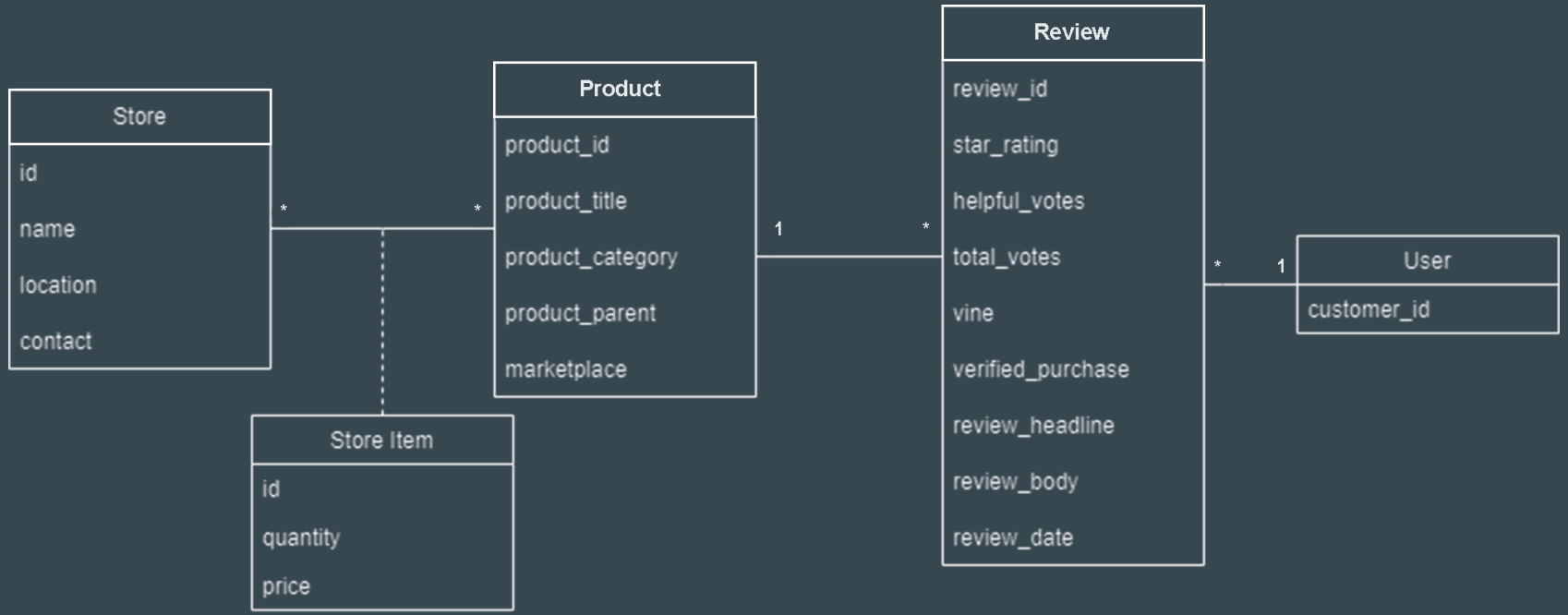
- Product and store listings.
- Complete Review CRUD, using KV operations and normal SQL++ (ARRAY utilities).
- FTS search.
- Eventing trigger for rating calculation.
- Primary and secondary indexing on recurring fields.
- Aggregation queries on all document entities (e.g. avg product price, applying discounts across stores).
- Transaction for review addition/deletion.
- *Geospacial* querying for the distance radius filter.

# Live demo

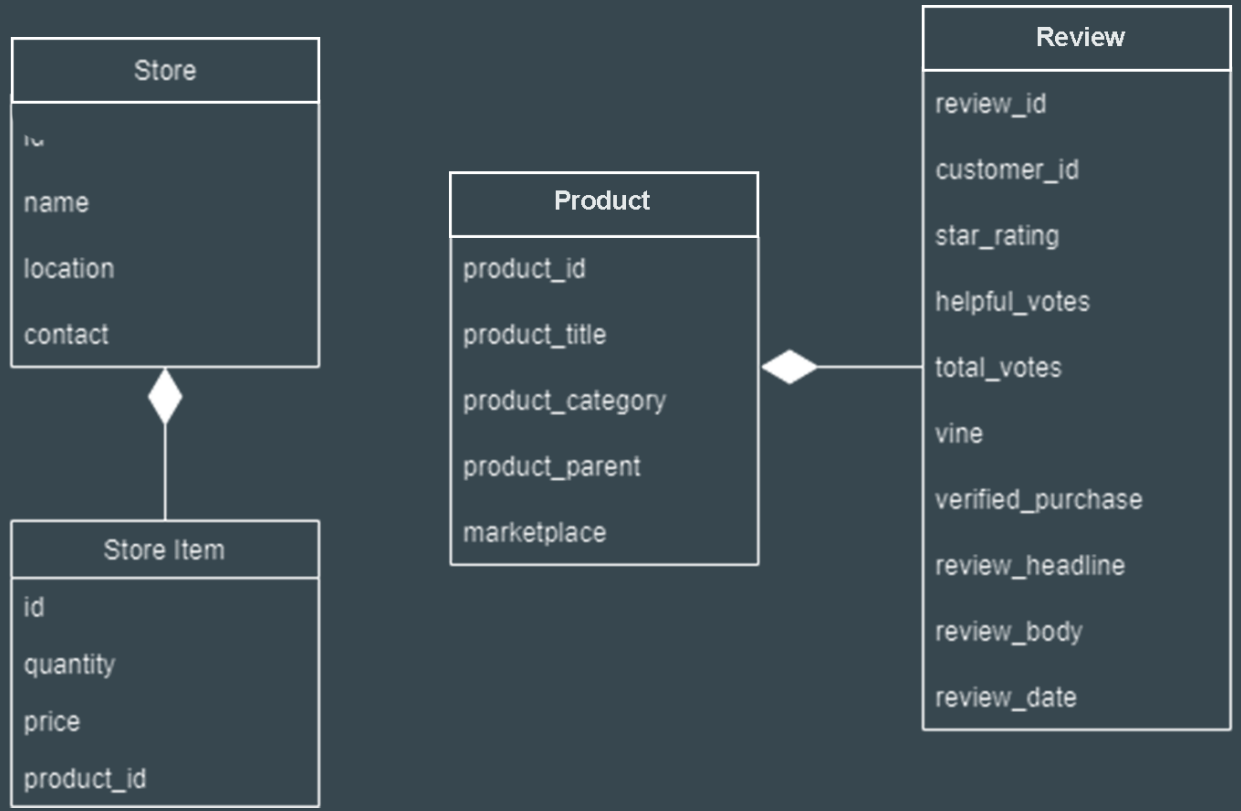
# Objective

Create a store management system, that allows a user to find the nearest store that has a certain type of product or a general store clerk/owner to keep track of its inventory.

# Conceptual Data Model



# Aggregation Data Model



# Queries

1. Product and its reviews by product id.
2. Contact of the store by store id.
3. Every store with a product in stock.
4. Products whose reviews and product title best match a full text search from user input.
5. Products of a determined category that are near the user's location (using the geospatial query feature).
6. Products by store, price, quantity, category, rating or any combination of these.
7. Reviews authored by a specific consumer.

# Queries

## Retrieval of a product and its reviews by  product id

- The user wants to view a product in a dedicated page, including its characteristics and reviews.
- Good performance and ease of implementation due to products being an aggregate (each contains reviews pertaining to it) and therefore having a primary index.
- Schemaless approach allows for products with different specifications to be stored in the same collection.

# Queries

## Contact of a store by store id

- The user wants to contact a store, usually to purchase or obtain additional information related to a product.
- Stores are aggregates and querying these by id is generally easy and with good performance.

# Queries

## Retrieve every store with a product in stock

- The user wants to view which stores sell a desired product.
- Querying all the stores containing a product won't be as efficient, and therefore creating a secondary index for this purpose might be beneficial.

# Queries

## Products whose reviews and product title best match a full text search form user input

- Users may be looking for a specific product or specific feedback from other users to help with their decisions.
- Couchbase Search Service supports the creation of specially purposed indexes for Full Text Search.

# Queries

## Products of a determined category that are near the user's location

- The user may require a certain type of product, but is bounded by transportation constraints. Therefore it's appropriate to have the list of nearby stores, that have that product in stock.
- Couchbase offers a geospatial query feature, that restricts search by distance between two locations (compares triplets of latitude, longitude, altitude), and can even map proximity with radius, rectangle or other polygon area limits.

# Queries

## Products by store, price, quantity, category, rating, or any combination of these

- When looking for a product users expect to be able to filter available options by their personal preferences, reducing the available options to those fitting their criteria.
- Couchbase's query language, N1QL, allows for easy combination of filters.

# Queries

## Reviews authored by a specific customer

- When visiting a review's author page, the user can find all reviews made by that given customer.
- The customer ids are included in the reviews' documents, thus a direct N1QL will suffice.

# Additional implementations

- Indexing on rankings and category for better retrieval performance.
- Trigger functions to calculate on change the average product rating.
- Procedures in order to alter quantity or price of a group of products.
- Add/Remove products, reviews and product info.
- Facet like interface for product filtering.
- Distributed sharding, throughout several containers/machines.

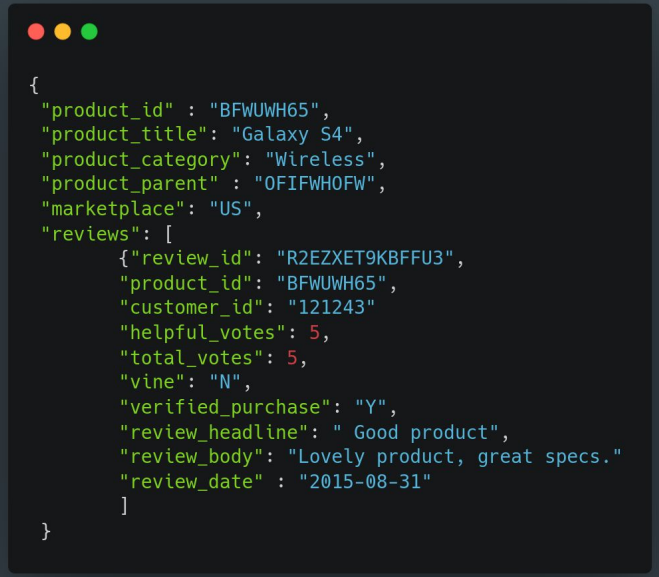
# Couchbase implementation

A single bucket since all of our information is related and will be used in the same application.

A single scope for now, more could be used if desire to display products taking into account the user's region arises.

Two collections, that follows the aggregation model: one pairing each store with the products' amount and price, and the other pairing the products (and their general info) with their reviews.

# Documents



```
{
 "product_id" : "BFWUWH65",
 "product_title": "Galaxy S4",
 "product_category": "Wireless",
 "product_parent" : "OFIFWHOFW",
 "marketplace": "US",
 "reviews": [
        {"review_id": "R2EZXET9KBFFU3",
        "product_id": "BFWUWH65",
        "customer_id": "121243"
        "helpful_votes": 5,
        "total_votes": 5,
        "vine": "N",
        "verified_purchase": "Y",
        "review_headline": " Good product",
        "review_body": "Lovely product, great specs."
        "review_date" : "2015-08-31"
        ]
 }
```

```
{
 "id" : "URA1033"
 "name": "Loja do João",
 "location": (-10.4, 6.8),
 "contact": 900000000,
 "store_items": [
        {"id": 10,
        "product_id": "BFWUWH65",
        "price": 50.12}
        ]
 }
```