# ✅ Goal

You have an **Aurora PostgreSQL** database with a table that stores **vector embeddings**. You want a **Spring Boot** app that can **insert** rows and **retrieve/search** them (including **similarity search** by embedding).

This guide gives you: - SQL to prepare Aurora (pgvector table + index) - A minimal Spring Boot project (JDBC + REST) that: - inserts documents with embeddings - fetches by ID - performs **cosine similarity** search using pgvector - Two ways to bind vectors from Java: **(A)** no extra library (string literal + ::vector) and **(B)** optional **pgvector‑java** library

**Tip:** Replace the example embedding **dimension** (e.g., 768) with the dimension your model produces.

## 1) Aurora PostgreSQL setup

Run these *once* on your Aurora cluster:

-- 1) Enable pgvector (once per DB)  
CREATE EXTENSION IF NOT EXISTS vector;  
  
-- 2) Create a table for documents with an embedding  
-- NOTE: change 768 to your embedding dimension  
CREATE TABLE IF NOT EXISTS docs (  
 id UUID PRIMARY KEY,  
 content TEXT NOT NULL,  
 embedding VECTOR(768) NOT NULL,  
 metadata JSONB,  
 created\_at TIMESTAMPTZ DEFAULT now()  
);  
  
-- 3) Create an IVFFlat index for cosine distance (common for text embeddings)  
-- You can tune `lists` based on data size; start with 100–200 and benchmark  
CREATE INDEX IF NOT EXISTS docs\_embedding\_idx  
 ON docs USING ivfflat (embedding vector\_cosine\_ops)  
 WITH (lists = 100);  
  
-- 4) (Optional) Analyze to help planner  
ANALYZE docs;

If your Aurora/pgvector version supports other indexes (e.g., HNSW), you can create those instead. The SQL above is portable and safe.

## 2) Spring Boot project (JDBC + REST)

### 2.1 pom.xml

Use Spring Boot **3.x** (Java 17+). This setup uses **spring-boot-starter-jdbc** and the **PostgreSQL** driver. The vector binding uses **Approach A** (no extra lib). We include an **optional** dependency for **Approach B** (pgvector‑java) commented out.

<?xml version="1.0" encoding="UTF-8"?>  
<project xmlns="http://maven.apache.org/POM/4.0.0"  
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <modelVersion>4.0.0</modelVersion>  
  
 <parent>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-parent</artifactId>  
 <version>3.3.2</version>  
 <relativePath/>  
 </parent>  
  
 <groupId>com.example</groupId>  
 <artifactId>vecdemo</artifactId>  
 <version>0.0.1-SNAPSHOT</version>  
 <name>vecdemo</name>  
 <description>Spring Boot + Aurora Postgres (pgvector)</description>  
  
 <properties>  
 <java.version>17</java.version>  
 </properties>  
  
 <dependencies>  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-jdbc</artifactId>  
 </dependency>  
  
 <dependency>  
 <groupId>org.postgresql</groupId>  
 <artifactId>postgresql</artifactId>  
 <scope>runtime</scope>  
 </dependency>  
  
 <!-- Optional: for concise POJOs -->  
 <dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <optional>true</optional>  
 </dependency>  
  
 <!-- Optional Approach B: typed PG vector support (uncomment to use) -->  
 <!--  
 <dependency>  
 <groupId>com.pgvector</groupId>  
 <artifactId>pgvector</artifactId>  
 <version>0.1.4</version>  
 </dependency>  
 -->  
 </dependencies>  
  
 <build>  
 <plugins>  
 <plugin>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-maven-plugin</artifactId>  
 </plugin>  
 </plugins>  
 </build>  
</project>

If you prefer Gradle, you can translate the dependencies accordingly.

### 2.2 application.yml

Replace placeholders with your **Aurora cluster endpoint**, DB name, and credentials. sslmode=require is recommended.

server:  
 port: 8080  
  
spring:  
 datasource:  
 url: jdbc:postgresql://<AURORA-CLUSTER-ENDPOINT>:5432/<DB\_NAME>?sslmode=require  
 username: <DB\_USERNAME>  
 password: <DB\_PASSWORD>  
 hikari:  
 maximum-pool-size: 10  
 minimum-idle: 2  
 jackson:  
 serialization:  
 WRITE\_DATES\_AS\_TIMESTAMPS: false

**Using IAM auth or Secrets Manager?** See the appendix at the end for code snippets.

## 3) Code — Model, Utils, Repository, Service, Controller

**Package suggestion:** com.example.vecdemo

### 3.1 Model & DTOs

src/main/java/com/example/vecdemo/model/Document.java

package com.example.vecdemo.model;  
  
import lombok.\*;  
import java.time.Instant;  
import java.util.\*;  
import java.util.stream.\*;  
  
@Data  
@NoArgsConstructor  
@AllArgsConstructor  
@Builder  
public class Document {  
 private UUID id;  
 private String content;  
 private List<Double> embedding; // JSON payload uses numbers  
 private Map<String, Object> metadata;  
 private Instant createdAt; // filled by DB  
 private Double distance; // populated for search results  
}

src/main/java/com/example/vecdemo/model/SearchRequest.java

package com.example.vecdemo.model;  
  
import lombok.Data;  
import java.util.\*;  
  
@Data  
public class SearchRequest {  
 private List<Double> query; // the query embedding  
 private Integer limit = 5; // default  
}

### 3.2 Vector helpers

src/main/java/com/example/vecdemo/util/VectorUtils.java

package com.example.vecdemo.util;  
  
import java.util.\*;  
import java.util.stream.\*;  
  
public class VectorUtils {  
 // Convert Java list to pgvector literal: "[x1, x2, x3]"  
 public static String toPgVectorLiteral(List<Double> vec) {  
 if (vec == null || vec.isEmpty()) return "[]";  
 return vec.stream()  
 .map(d -> {  
 if (d == null) return "0"; // guard  
 // Ensure non-scientific where possible  
 String s = Double.toString(d);  
 return s;  
 })  
 .collect(Collectors.joining(", ", "[", "]"));  
 }  
  
 // Parse pgvector text (e.g., "[0.1, -0.2, 0.3]") back to Java List<Double>  
 public static List<Double> parsePgVector(String text) {  
 if (text == null || text.isBlank()) return Collections.emptyList();  
 String trimmed = text.trim();  
 if (trimmed.startsWith("[") && trimmed.endsWith("]")) {  
 trimmed = trimmed.substring(1, trimmed.length() - 1);  
 }  
 if (trimmed.isBlank()) return Collections.emptyList();  
 String[] parts = trimmed.split(",");  
 List<Double> out = new ArrayList<>(parts.length);  
 for (String p : parts) {  
 out.add(Double.parseDouble(p.trim()));  
 }  
 return out;  
 }  
}

### 3.3 Repository (JdbcTemplate)

src/main/java/com/example/vecdemo/repo/DocumentRepository.java

package com.example.vecdemo.repo;  
  
import com.example.vecdemo.model.Document;  
import com.example.vecdemo.util.VectorUtils;  
import com.fasterxml.jackson.core.JsonProcessingException;  
import com.fasterxml.jackson.databind.ObjectMapper;  
import lombok.RequiredArgsConstructor;  
import org.springframework.jdbc.core.JdbcTemplate;  
import org.springframework.jdbc.core.RowMapper;  
import org.springframework.stereotype.Repository;  
  
import java.sql.ResultSet;  
import java.sql.SQLException;  
import java.time.OffsetDateTime;  
import java.util.\*;  
  
@Repository  
@RequiredArgsConstructor  
public class DocumentRepository {  
 private final JdbcTemplate jdbc;  
 private final ObjectMapper mapper;  
  
 private final RowMapper<Document> baseMapper = new RowMapper<>() {  
 @Override  
 public Document mapRow(ResultSet rs, int rowNum) throws SQLException {  
 Document d = new Document();  
 d.setId((UUID) rs.getObject("id"));  
 d.setContent(rs.getString("content"));  
 d.setEmbedding(VectorUtils.parsePgVector(rs.getString("embedding")));  
 String json = rs.getString("metadata");  
 try {  
 d.setMetadata(json == null ? null : mapper.readValue(json, Map.class));  
 } catch (Exception e) {  
 d.setMetadata(null);  
 }  
 OffsetDateTime odt = rs.getObject("created\_at", OffsetDateTime.class);  
 d.setCreatedAt(odt == null ? null : odt.toInstant());  
 // distance may be absent for non-search queries  
 Object dist = rs.getObject("distance");  
 d.setDistance(dist == null ? null : ((Number) dist).doubleValue());  
 return d;  
 }  
 };  
  
 public Document insert(Document doc) {  
 UUID id = doc.getId() != null ? doc.getId() : UUID.randomUUID();  
 String vec = VectorUtils.toPgVectorLiteral(doc.getEmbedding());  
 String metadataJson = null;  
 try {  
 metadataJson = doc.getMetadata() == null ? null : mapper.writeValueAsString(doc.getMetadata());  
 } catch (JsonProcessingException e) {  
 throw new RuntimeException("Failed to serialize metadata", e);  
 }  
  
 jdbc.update(  
 "INSERT INTO docs (id, content, embedding, metadata) VALUES (?, ?, ?::vector, ?::jsonb)",  
 id, doc.getContent(), vec, metadataJson  
 );  
  
 return findById(id).orElseThrow();  
 }  
  
 public Optional<Document> findById(UUID id) {  
 List<Document> list = jdbc.query(  
 "SELECT id, content, embedding, metadata, created\_at FROM docs WHERE id = ?",  
 baseMapper,  
 id  
 );  
 return list.stream().findFirst();  
 }  
  
 public List<Document> searchByCosine(List<Double> queryVec, int limit) {  
 String vec = VectorUtils.toPgVectorLiteral(queryVec);  
 // Use a subselect so we bind the vector once  
 String sql = """  
 SELECT id, content, embedding, metadata, created\_at,  
 (embedding <=> q.vec) AS distance  
 FROM docs, (SELECT ?::vector AS vec) AS q  
 ORDER BY embedding <=> q.vec  
 LIMIT ?  
 """;  
 return jdbc.query(sql, baseMapper, vec, limit);  
 }  
}

### 3.4 Service

src/main/java/com/example/vecdemo/service/DocumentService.java

package com.example.vecdemo.service;  
  
import com.example.vecdemo.model.Document;  
import com.example.vecdemo.repo.DocumentRepository;  
import lombok.RequiredArgsConstructor;  
import org.springframework.stereotype.Service;  
  
import java.util.\*;  
  
@Service  
@RequiredArgsConstructor  
public class DocumentService {  
 private final DocumentRepository repo;  
  
 public Document create(Document d) {  
 return repo.insert(d);  
 }  
  
 public Optional<Document> get(UUID id) {  
 return repo.findById(id);  
 }  
  
 public List<Document> search(List<Double> query, int limit) {  
 return repo.searchByCosine(query, limit);  
 }  
}

### 3.5 REST Controller

src/main/java/com/example/vecdemo/web/DocumentController.java

package com.example.vecdemo.web;  
  
import com.example.vecdemo.model.Document;  
import com.example.vecdemo.model.SearchRequest;  
import com.example.vecdemo.service.DocumentService;  
import lombok.RequiredArgsConstructor;  
import org.springframework.http.ResponseEntity;  
import org.springframework.web.bind.annotation.\*;  
  
import java.util.\*;  
  
@RestController  
@RequestMapping("/api/docs")  
@RequiredArgsConstructor  
public class DocumentController {  
 private final DocumentService service;  
  
 @PostMapping  
 public ResponseEntity<Document> create(@RequestBody Document body) {  
 Document saved = service.create(body);  
 return ResponseEntity.ok(saved);  
 }  
  
 @GetMapping("/{id}")  
 public ResponseEntity<Document> get(@PathVariable("id") UUID id) {  
 return service.get(id)  
 .map(ResponseEntity::ok)  
 .orElse(ResponseEntity.notFound().build());  
 }  
  
 @PostMapping("/search")  
 public List<Document> search(@RequestBody SearchRequest req) {  
 int limit = (req.getLimit() == null || req.getLimit() <= 0) ? 5 : req.getLimit();  
 return service.search(req.getQuery(), limit);  
 }  
}

### 3.6 Boot app class

src/main/java/com/example/vecdemo/VecdemoApplication.java

package com.example.vecdemo;  
  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;  
  
@SpringBootApplication  
public class VecdemoApplication {  
 public static void main(String[] args) {  
 SpringApplication.run(VecdemoApplication.class, args);  
 }  
}

## 4) Try it out (curl examples)

Start the app, then:

**Insert** a document (short vector shown for brevity—use your full dimension!):

curl -X POST http://localhost:8080/api/docs \  
 -H 'Content-Type: application/json' \  
 -d '{  
 "content": "Apple launches new product.",  
 "embedding": [0.12, -0.34, 0.56, 0.78],  
 "metadata": {"source": "news", "lang": "en"}  
 }'

Response includes the generated id and createdAt.

**Get by ID**:

curl http://localhost:8080/api/docs/<UUID>

**Similarity search (cosine)**:

curl -X POST http://localhost:8080/api/docs/search \  
 -H 'Content-Type: application/json' \  
 -d '{  
 "query": [0.10, -0.30, 0.60, 0.80],  
 "limit": 3  
 }'

Each result returns distance (lower is more similar for cosine distance).

## 5) Notes & gotchas

* **Dimensions must match**: VECTOR(768) means your arrays must be length 768. Otherwise inserts will fail.
* **Indexing**: For cosine search use vector\_cosine\_ops. For L2 use vector\_l2\_ops. Choose based on your embedding space.
* **Batch inserts**: You can batch multiple INSERTs with JdbcTemplate#batchUpdate for speed.
* **Migrations**: Use **Flyway** or **Liquibase** to manage the SQL in src/main/resources/db/migration.
* **Performance**: For IVFFlat, run SET ivfflat.probes = <k>; (per session) to trade accuracy for speed. You can set a default at the DB level.
* **SSL**: Prefer sslmode=require when connecting to Aurora from outside AWS.

## 6) Approach B (optional): typed vectors via pgvector‑java

If you prefer a strong type instead of string literals, add the dependency and use PGvector.

**pom.xml** (uncomment):

<dependency>  
 <groupId>com.pgvector</groupId>  
 <artifactId>pgvector</artifactId>  
 <version>0.1.4</version>  
</dependency>

**Example usage** (inside a JdbcTemplate callback or plain JDBC):

import com.pgvector.PGvector; // class name from the library  
  
// After obtaining a Connection (e.g., via DataSourceUtils.getConnection(ds))  
PGvector.addVectorType(conn);  
  
// Insert  
try (PreparedStatement ps = conn.prepareStatement(  
 "INSERT INTO docs (id, content, embedding, metadata) VALUES (?, ?, ?, ?::jsonb)")) {  
 ps.setObject(1, UUID.randomUUID());  
 ps.setString(2, "Hello world");  
 ps.setObject(3, new PGvector(new float[]{0.12f, -0.34f, 0.56f}));  
 ps.setString(4, "{\"k\":\"v\"}");  
 ps.executeUpdate();  
}  
  
// Query (similarity)  
try (PreparedStatement ps = conn.prepareStatement(  
 "SELECT id, content, embedding, metadata, created\_at, embedding <=> ? AS distance FROM docs ORDER BY embedding <=> ? LIMIT ?")) {  
 PGvector q = new PGvector(new float[]{0.1f, -0.3f, 0.6f});  
 ps.setObject(1, q);  
 ps.setObject(2, q);  
 ps.setInt(3, 5);  
 try (ResultSet rs = ps.executeQuery()) {  
 while (rs.next()) {  
 PGvector embedding = (PGvector) rs.getObject("embedding");  
 float[] arr = embedding.toArray();  
 // ...  
 }  
 }  
}

## 7) (Optional) IAM auth / Secrets Manager

### A) Using **Secrets Manager** for credentials

Store a secret with your DB username/password and load it at startup.

// @Configuration  
// Fetch JSON secret {"username":"...","password":"..."} and create DataSource

High-level steps: 1. Put creds in Secrets Manager. 2. Grant the instance/role permission secretsmanager:GetSecretValue. 3. On startup, read the secret and populate spring.datasource.username / password (e.g., via an EnvironmentPostProcessor or a small bootstrap bean).

### B) Using **IAM database authentication** (token based)

Generate a token with AWS SDK and use it as the JDBC password; require SSL.

// Example bean to create a DataSource with IAM auth  
  
import org.springframework.context.annotation.\*;  
import javax.sql.DataSource;  
import com.zaxxer.hikari.HikariDataSource;  
import software.amazon.awssdk.auth.credentials.DefaultCredentialsProvider;  
import software.amazon.awssdk.regions.Region;  
import software.amazon.awssdk.services.rds.RdsUtilities;  
import software.amazon.awssdk.services.rds.RdsClient;  
  
@Configuration  
public class DataSourceConfig {  
 @Bean  
 public DataSource dataSource() {  
 String host = "<AURORA-CLUSTER-ENDPOINT>";  
 int port = 5432;  
 String db = "<DB\_NAME>";  
 String user = "<DB\_USER\_WITH\_IAM\_AUTH>";  
  
 RdsClient rds = RdsClient.builder()  
 .region(Region.<YOUR\_REGION>)  
 .credentialsProvider(DefaultCredentialsProvider.create())  
 .build();  
  
 RdsUtilities utils = rds.utilities();  
 String token = utils.generateAuthenticationToken(b -> b  
 .endpoint(host)  
 .port(port)  
 .username(user)  
 );  
  
 HikariDataSource ds = new HikariDataSource();  
 ds.setJdbcUrl("jdbc:postgresql://" + host + ":" + port + "/" + db + "?sslmode=require");  
 ds.setUsername(user);  
 ds.setPassword(token); // token is the password  
 ds.addDataSourceProperty("ssl", "true");  
 return ds;  
 }  
}

Ensure the DB user has rds\_iam role and your runtime role can call rds-db:connect.

## 8) Troubleshooting

* **column is of type vector but expression is of type text** → ensure you cast parameter: ?::vector in SQL when using string literal approach.
* **dimension mismatch** → your JSON array length must equal the table dimension.
* **permission denied to create extension "vector"** → run CREATE EXTENSION as a superuser/cluster admin.
* **Slow search** → create IVFFlat index, increase lists, and set SET ivfflat.probes = 10 (start small, benchmark).
* **SSL errors** → use ?sslmode=require and ensure CA trust if needed.

## 9) What to change for your project

* Replace VECTOR(768) with your true dimension
* Swap content/metadata with your actual columns
* Pick distance metric + operator class to match your embeddings (cosine is common)

**You now have a working template** to insert and retrieve vectorized data from **Aurora PostgreSQL** using **Spring Boot**. Paste these files into a new project and run! If you want the same but with **JPA/Hibernate** custom types instead of JdbcTemplate, we can add that next.