Yes, absolutely! Implementing the filter functionality and fetching the data on the **backend (Java/Hibernate)** is the standard and most scalable approach, especially for large datasets. The Angular frontend simply sends the filter parameters to your backend API, and your Java application handles the database query.

The `searchProductsAcrossAutoDetectedColumns` method (using JPA Criteria API) and the `getProductsPaginated` method you already have are the building blocks. We'll combine them and adjust the backend to receive the filter values.

1. Backend DTO for Paginated & Filtered Response

First, let's define a simple DTO (Data Transfer Object) in your Java backend that will encapsulate the paginated content and the total count. This mirrors the `ProductFetchResult` interface you used in Angular.

```java
// src/main/java/com/example/dto/PaginatedResponse.java
package com.example.dto;

import java.util.List;

public class PaginatedResponse<T> {
    private List<T> content;
    private long totalElements;

    public PaginatedResponse() {}

    public PaginatedResponse(List<T> content, long totalElements) {
        this.content = content;
        this.totalElements = totalElements;
    }

    // Getters and Setters
    public List<T> getContent() {
        return content;
    }

    public void setContent(List<T> content) {
        this.content = content;
    }

    public long getTotalElements() {
        return totalElements;
    }
```

```
    public void setTotalElements(long totalElements) {
        this.totalElements = totalElements;
    }
}
```

2. Modified `ProductDAO` (Backend)

We'll add a new method to `ProductDAO` that combines pagination and dynamic filtering using the Criteria API.

```java
// src/main/java/com/example/dao/ProductDAO.java
package com.example.dao;

import com.example.model.Product;
import com.example.dto.PaginatedResponse; // Import the DTO
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import org.hibernate.metamodel.model.domain.ManagedType;
import org.hibernate.metamodel.model.domain.Attribute;

import jakarta.persistence.criteria.CriteriaBuilder;
import jakarta.persistence.criteria.CriteriaQuery;
import jakarta.persistence.criteria.Predicate;
import jakarta.persistence.criteria.Root;
import jakarta.persistence.criteria.Expression;

import java.util.ArrayList;
import java.util.List;
import java.util.Locale;
import java.util.Map; // Import Map
import java.util.Set;

public class ProductDAO {

    private static SessionFactory sessionFactory;
    private static List<String> productTextualColumnNames;

    static {
        try {
```

```
            sessionFactory = new
Configuration().configure("hibernate.cfg.xml").buildSessionFactory();
            System.out.println("SessionFactory initialized.");

            loadProductTextualColumnNames();
            System.out.println("Product textual column names loaded: " +
productTextualColumnNames);

        } catch (Throwable ex) {
            System.err.println("Initial SessionFactory creation failed." + ex);
            throw new ExceptionInInitializerError(ex);
        }
    }

    private static void loadProductTextualColumnNames() {
        productTextualColumnNames = new ArrayList<>();
        if (sessionFactory != null) {
            ManagedType<Product> productManagedType = (ManagedType<Product>)
sessionFactory.getMetamodel().managedType(Product.class);
            Set<? extends Attribute<?, ?>> attributes = productManagedType.getAttributes();

            System.out.println("--- Detecting textual columns for Product entity ---");
            for (Attribute<?, ?> attribute : attributes) {
                System.out.println("  Attribute: " + attribute.getName() + ", Java Type: " +
attribute.getJavaType().getName());
                if (attribute.getJavaType() == String.class) {
                    productTextualColumnNames.add(attribute.getName());
                    System.out.println("    -> Added as textual column for search.");
                } else {
                    System.out.println("    -> Skipped (not a String type).");
                }
            }
            System.out.println("-------------------------------------------------");
        }
    }

    public static SessionFactory getSessionFactory() {
        return sessionFactory;
    }

    public static void shutdown() {
        if (sessionFactory != null) {
            sessionFactory.close();
            System.out.println("SessionFactory closed.");
```

```
        }
    }

    // ... (Keep your existing getAllProducts, getProductsPaginated, getTotalProductCount methods here) ...
    // Note: The new method 'getProductsPaginatedAndFiltered' will supersede getProductsPaginated for most uses.

    /**
     * Retrieves a paginated and filtered list of products.
     * Applies filters based on a map of column names to keywords.
     *
     * @param pageNumber The page number (0-based index, as typically used by paginators like MatPaginator).
     * @param pageSize The number of records per page.
     * @param filters A map where keys are entity field names (e.g., "productName", "category")
     * and values are the search keywords for that column.
     * @return A PaginatedResponse containing the list of Product entities and the total count.
     */
    public PaginatedResponse<Product> getProductsPaginatedAndFiltered(
            int pageNumber,
            int pageSize,
            Map<String, String> filters,
            String sortBy, // Added for optional sorting
            String sortDirection // Added for optional sorting
    ) {
        Session session = null;
        Transaction transaction = null;
        List<Product> products = new ArrayList<>();
        long totalCount = 0;

        if (pageNumber < 0) pageNumber = 0; // Ensure 0-based page index
        if (pageSize < 1) pageSize = 10;

        try {
            session = sessionFactory.openSession();
            transaction = session.beginTransaction();

            CriteriaBuilder cb = session.getCriteriaBuilder();

            // --- 1. Build Predicates for Filtering ---
            List<Predicate> filterPredicates = new ArrayList<>();
            if (filters != null && !filters.isEmpty()) {
                for (Map.Entry<String, String> entry : filters.entrySet()) {
```

```
                String columnName = entry.getKey();
                String filterKeyword = entry.getValue();

                if (filterKeyword != null && !filterKeyword.trim().isEmpty()) {
                    String lowerKeyword = "%" + filterKeyword.toLowerCase(Locale.ROOT) + "%";

                    // Ensure the column is a string type before applying LIKE
                    // (This check relies on productTextualColumnNames being accurate)
                    if (productTextualColumnNames.contains(columnName)) {
                        try {
                            Expression<String> columnExpression =
productRoot.get(columnName).as(String.class);
                            filterPredicates.add(cb.like(cb.lower(columnExpression), lowerKeyword));
                        } catch (IllegalArgumentException | ClassCastException e) {
                            System.err.println("Warning: Skipping filter for column '" + columnName + "'
due to type mismatch or invalid path: " + e.getMessage());
                        }
                    } else {
                        // Optionally, handle non-string filters here (e.g., equals for numbers)
                        // For instance, if 'price' is a number and you want exact match:
                        // if (columnName.equals("price") && !filterKeyword.isEmpty()) {
                        //     try {
                        //         filterPredicates.add(cb.equal(productRoot.get("price"), new
BigDecimal(filterKeyword)));
                        //     } catch (NumberFormatException ignore) { /* not a valid number */ }
                        // }
                        System.out.println("Filter for column '" + columnName + "' is not a textual
column or not handled in criteria API.");
                    }
                }
            }
        }

        // --- 2. Get Total Count (with filters applied) ---
        CriteriaQuery<Long> countQuery = cb.createQuery(Long.class);
        Root<Product> countRoot = countQuery.from(Product.class);
        countQuery.select(cb.count(countRoot));
        if (!filterPredicates.isEmpty()) {
            countQuery.where(cb.and(filterPredicates.toArray(new Predicate[0]))); // Use AND for
combining column filters
        }
        totalCount = session.createQuery(countQuery).uniqueResult();
```

```
            // --- 3. Get Paginated Content (with filters and sorting) ---
            CriteriaQuery<Product> productQuery = cb.createQuery(Product.class);
            Root<Product> productRoot = productQuery.from(Product.class);

            if (!filterPredicates.isEmpty()) {
                productQuery.where(cb.and(filterPredicates.toArray(new Predicate[0]))); // Apply
same combined filters
            }

            // Apply sorting if sortBy is provided
            if (sortBy != null && !sortBy.trim().isEmpty()) {
                if ("asc".equalsIgnoreCase(sortDirection)) {
                    productQuery.orderBy(cb.asc(productRoot.get(sortBy)));
                } else {
                    productQuery.orderBy(cb.desc(productRoot.get(sortBy)));
                }
            } else {
                // Default sort order if none specified, good for consistent pagination
                productQuery.orderBy(cb.asc(productRoot.get("productId")));
            }

            // Apply pagination
            products = session.createQuery(productQuery)
                              .setFirstResult(pageNumber * pageSize)
                              .setMaxResults(pageSize)
                              .getResultList();

            transaction.commit();

        } catch (Exception e) {
            if (transaction != null) {
                transaction.rollback();
            }
            e.printStackTrace();
        } finally {
            if (session != null) {
                session.close();
            }
        }
        return new PaginatedResponse<>(products, totalCount);
    }

    public static void main(String[] args) {
        ProductDAO dao = new ProductDAO();
```

```
        // --- Example of Populating Data (if needed) ---
        // Uncomment and run this block once if your table is empty or for fresh data.
        /*
        try (Session session = sessionFactory.openSession()) {
            Transaction tx = session.beginTransaction();
            session.persist(new Product(1L, "Laptop", new BigDecimal("1200.00"), 50, "Powerful
computing machine for professionals."));
            session.persist(new Product(2L, "Mouse", new BigDecimal("25.00"), 200, "Ergonomic
computer input device for daily use."));
            session.persist(new Product(3L, "Keyboard", new BigDecimal("75.00"), 100, "Standard
office keyboard with tactile keys."));
            session.persist(new Product(4L, "Monitor", new BigDecimal("300.00"), 75,
"High-resolution display for multimedia."));
            session.persist(new Product(5L, "Webcam", new BigDecimal("50.00"), 150, "Basic
webcam for video calls."));
            session.persist(new Product(16L, "Gaming Keyboard", new BigDecimal("150.00"), 60,
"RGB mechanical keyboard for serious gamers."));
            session.persist(new Product(17L, "Desktop PC", new BigDecimal("1500.00"), 30,
"High-performance machine for complex tasks."));
            tx.commit();
            System.out.println("Sample data inserted.");
        } catch (Exception e) {
            e.printStackTrace();
        }
        */
        // --- End of Data Population Example ---

        System.out.println("\n--- Fetching Page 0 (5 items) with filter 'mouse' in any text column
---");
        // Example filter: Only filter by 'productName' and 'description'
        Map<String, String> filters1 = new java.util.HashMap<>();
        filters1.put("productName", "mouse"); // Search 'mouse' in productName
        filters1.put("description", "daily"); // Search 'daily' in description
        PaginatedResponse<Product> result1 = dao.getProductsPaginatedAndFiltered(0, 5,
filters1, "productName", "asc");
        System.out.println("Total filtered elements (mouse, daily): " + result1.getTotalElements());
        result1.getContent().forEach(System.out::println);

        System.out.println("\n--- Fetching Page 0 (5 items) with filter 'Keyboard' in any text column
---");
        Map<String, String> filters2 = new java.util.HashMap<>();
        filters2.put("productName", "keyboard"); // Search 'keyboard' in productName
```

```
        PaginatedResponse<Product> result2 = dao.getProductsPaginatedAndFiltered(0, 5,
filters2, "productName", "asc");
        System.out.println("Total filtered elements (keyboard): " + result2.getTotalElements());
        result2.getContent().forEach(System.out::println);

        System.out.println("\n--- Fetching Page 0 (5 items) with no filters ---");
        Map<String, String> filters3 = new java.util.HashMap<>(); // Empty map for no filters
        PaginatedResponse<Product> result3 = dao.getProductsPaginatedAndFiltered(0, 5,
filters3, "productId", "asc");
        System.out.println("Total filtered elements (no filters): " + result3.getTotalElements());
        result3.getContent().forEach(System.out::println);


        ProductDAO.shutdown();
    }
}
```

3. Backend Controller (e.g., Spring Boot REST Controller)

If you're using Spring Boot (which is common for REST APIs), your controller would look something like this:

```java
// src/main/java/com/example/controller/ProductController.java
package com.example.controller;

import com.example.dao.ProductDAO;
import com.example.dto.PaginatedResponse;
import com.example.model.Product;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import java.util.HashMap;
import java.util.Map;

@RestController
@RequestMapping("/api/products")
public class ProductController {

    private final ProductDAO productDAO;
```

```
    // Assuming ProductDAO is managed by Spring or you initialize it directly
    // For simple apps, you can just new ProductDAO() or get it from a static accessor if it's
stateless.
    // In a full Spring app, you'd @Autowired it.
    public ProductController() {
        this.productDAO = new ProductDAO(); // Or inject via Spring if it's a Spring bean
    }

    @GetMapping
    public ResponseEntity<PaginatedResponse<Product>> getProducts(
            @RequestParam(defaultValue = "0") int page, // Angular's MatPaginator uses 0-based
index
            @RequestParam(defaultValue = "10") int size,
            @RequestParam Map<String, String> allRequestParams // Capture all request
parameters
    ) {
        Map<String, String> filters = new HashMap<>();
        String sortBy = null;
        String sortDirection = null;

        // Extract filter parameters (e.g.,
/api/products?page=0&size=5&productName=mouse&category=electronics)
        // You would define which query parameters are considered filters.
        // For simplicity, let's assume any param that is NOT 'page', 'size', 'sortBy', 'sortDirection' is
a filter.
        for (Map.Entry<String, String> entry : allRequestParams.entrySet()) {
            String paramName = entry.getKey();
            String paramValue = entry.getValue();

            if (!"page".equals(paramName) && !"size".equals(paramName) &&
!"sortBy".equals(paramName) && !"sortDirection".equals(paramName)) {
                filters.put(paramName, paramValue);
            } else if ("sortBy".equals(paramName)) {
                sortBy = paramValue;
            } else if ("sortDirection".equals(paramName)) {
                sortDirection = paramValue;
            }
        }

        PaginatedResponse<Product> result =
productDAO.getProductsPaginatedAndFiltered(page, size, filters, sortBy, sortDirection);
        return ResponseEntity.ok(result);
    }
```

}
```

### How it works together:

1. **Frontend (Angular):**
   * When the user types in a filter input or changes the page/sort, the Angular `AppComponent` constructs a request with `page`, `size`, and the `filterValues` (e.g., `productName=mouse&category=electronics`) as query parameters.
   * It calls `productService.getProducts(params)`, which in a real app would use `HttpClient` to send this GET request to your backend (e.g., `GET /api/products?page=0&size=5&productName=mouse`).

2. **Backend (Spring Boot Controller):**
   * The `ProductController.getProducts()` method receives these query parameters.
   * `@RequestParam Map<String, String> allRequestParams` captures all query parameters into a `Map`.
   * The controller then parses this `Map` to separate pagination (`page`, `size`), sorting (`sortBy`, `sortDirection`), and `filters` (anything else).

3. **Backend (ProductDAO):**
   * The controller calls `productDAO.getProductsPaginatedAndFiltered()`, passing the extracted `page`, `size`, and `filters` `Map`.
   * Inside the DAO method:
      * It constructs a `CriteriaQuery` to count the total filtered elements.
      * It constructs *another* `CriteriaQuery` for the actual data, applying the same filters and then the pagination (`setFirstResult`, `setMaxResults`) and sorting.
      * It returns both the list of `Product` objects (`content`) and the `totalElements` count wrapped in the `PaginatedResponse` DTO.

4. **Backend Response & Frontend Update:**
   * The Spring Boot controller sends this `PaginatedResponse` as a JSON object back to the Angular frontend.
   * The Angular `ProductService` receives this, and the `AppComponent` updates `dataSource.data` with the `content` and `totalElements` with the total count, which correctly updates the `MatPaginator`.

This complete flow allows your frontend to drive the search and pagination, while your backend efficiently handles the complex query logic and database interaction.
Citations: [[1]](https://github.com/nomosofa/tiny-ecommerce),
[[2]](https://github.com/AmberInsane/SpringMvcApp),
[[3]](https://github.com/niloofarhayati/ticketReserve)
```