Стратегия работы с jsonb, массивами и комплексными типами PostgreSQL для JPA

Антон Ромза
ITFB Group

Пролог

Что же включает в себя определение «нетривиальные типы» ?

Нетривиальные типы — это типы данных, которые состоят из фундаментальных (базовых) типов данных и инкапсулирующие специфичные только для данного типа функции

Пролог

В данном докладе освещены следующие типы данных:

- ✓ **composite types**
- ✓ **jsonb**
- ✓ **tsvector и tsquery**
- ✓ **enumerated types**
- ✓ **array**

Пролог

Что включает в себя стратегия работы с нетривиальным типами?

- Binding типов базы данных и приложения, использующего JPA
- Подходы к использованию нативных функций базы данных на уровне приложения
- DDL и DML операции из слоя приложения в БД
- Целесообразность использования того или иного типа исходя из функциональных потребностей и возможностей приложения
- Ограничения и аспекты, которые следует учитывать при проектировании модели хранения данных

Из Java в DML и обратно

Какие подходы мы можем реализовать для работы с нетривиальными типами:

1. Сбор SQL-строки в рантайме или константное определение
2. Создать определение хранимых функций с использованием инструментов управления схемой БД
3. Использовать встроенные инструментарии Hibernate/JPA

Из Java в DML и обратно

Сбор SLQ-строки запроса в runtime ведет к:

```
if (nonNull(spec.getAttributeDefinitionId())) {
    if (nonNull(spec.getDictionaryValueId())) {
        sql.append(" AND exists (select 1 from document_attribute_value where document_version_id = v.id \n" +
            .getDictionaryValueId() + ")\n");
    } else {
        sql.append(" AND exists (select 1 from document_attribute_value where document_version_id = v.id \n" +
            spec) + ")\n");
    }
} else if ("APP_USER".equalsIgnoreCase(spec.getExternalTable())) {
        if (nonNull(spec.getExternalId())) {
            sql.append(" AND v." + spec.getExternalField() + " = " + spec.getExternalId());
         }
} else if ("DOCUMENT_TYPE".equalsIgnoreCase(spec.getExternalTable())) {
        if (nonNull(spec.getExternalId())) {
            sql.append(" AND t.id = " + spec.getExternalId());
         }
}
```

Мотивация

Почему возникла тема данного доклада?

- Мало структурированной информации
- Небольшое количество проектов в Open Source

Что хотелось бы дать?

- Проверенный подход к работе с типами данных БД в JPA
- Cheatsheet для работы с данными нетривиальных типов

Нетривиальные типы в Open Source проектах

[Vlad Mihalcea library with general purpose utilities]

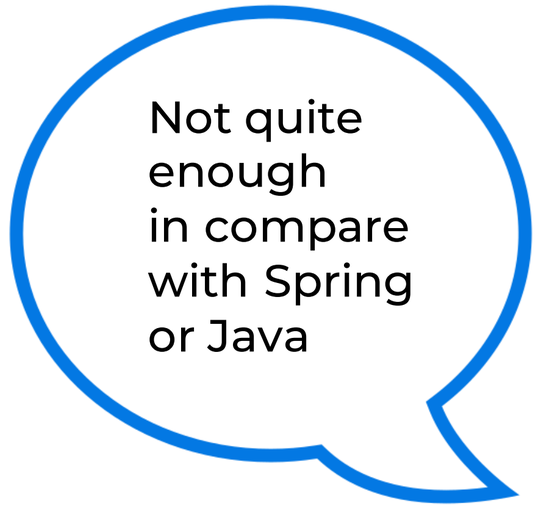
https://github.com/vladmihalcea/hypersistence-utils/tree/master

[Thorben Janssen blog]

https://thorben-janssen.com/

[Szymon Tarnowski]

https://github.com/starnowski/posjsonhelper



Стек

Рекомендуемый стек:

- **Spring 6.x**
- **Hibernate 6.x**
- **Spring Data JPA**
- **PostgreSQL 15.x**

Работа с типами из коробки Hibernate

Компоненты Hibernate

Для работы с нетривальными типами рекомендуется использовать следующие интерфейсы:

- JavaType и JdbcType интерфейсы и аннотации
- UserType и CompositeUserType
- UserCollectionType + PersistentCollection + CollectionType, если есть потребность кастомизировать поведение коллекций типов
- CompositeUserType

Не забываем про регистрацию типов, как локальную, так и глобальную.

Работа с функциями из коробки в JPA

На уровне Criteria API:

```
Predicate inJsonNumbers = cb
        .function("jsonb_extract_path_text",
                String.class,
                root.get("json"),
                cb.literal("number"))
        .in(filter.getJsonNumbers())

Predicate inJsonNames = cb
        .function("jsonb_extract_path_text",
                String.class,
                root.get("json"),
                cb.literal("name"))
        .in(filter.getJsonNames())
```

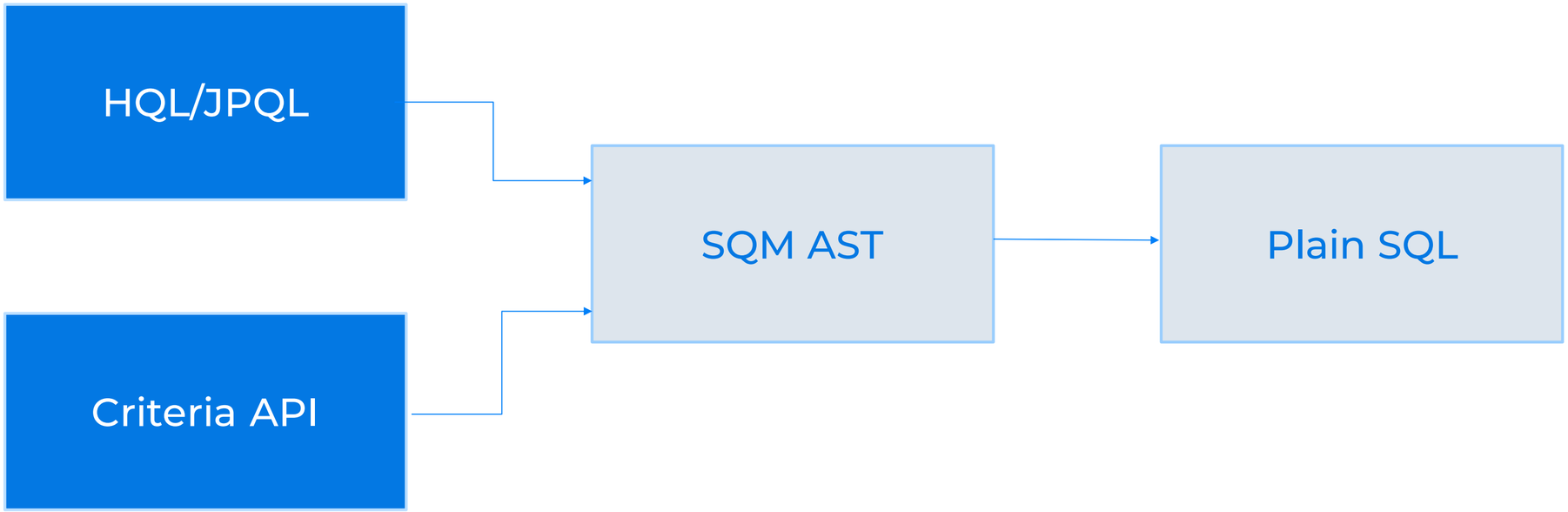
Работа с функциями в HQL/JPQL

На уровне JPQL-запроса:

```java
public interface EmailRepository extends JpaRepository<Email, Long> {
    @Query("SELECT e FROM Email e WHERE ts_query_perform(e.lexemes, ?1)")
    List<Email> fullTextQuery(String query);
}
```

База: Hibernate SQM

The Semantic Query Model (SQM) is Hibernate's representation of an HQL or Criteria query's semantic (meaning). This representation is modeled as an «abstract syntax tree» (AST) — meaning it is a structured tree of nodes where each node represents an atomic piece of the query.



База: диалект Hibernate

Диалект знает, как перевести HQL и SQM в запрос к конкретному типу БД:

```java
public class PostgreSQLDialectCustomized extends PostgreSQLDialect {
    @Override
    public void initializeFunctionRegistry(FunctionContributions functionContributions) {
        super.initializeFunctionRegistry(functionContributions);
        BasicTypeRegistry basicTypeRegistry = functionContributions.getTypeConfiguration().getBasicTypeRegistry
        var typeConfiguration = functionContributions.getTypeConfiguration();
        // ...code logic come next
    }
}
```

База: регистрация диалекта

Базовая регистрация диалекта через свойства Hibernate:

```yaml
spring:
  application:
    name: spring-app
  datasource:
    url: jdbc:postgresql://**********:5432/mydb
    username: **********
    password: **********
  jpa:
    show-sql: true
    hibernate:
      ddl-auto: create-drop
    properties:
      hibernate:
        format_sql: true
        dialect: jpa.pgsql.report.dialect.PostgreSQLDialectCustomized
```

Композитные типы

Базовая регистрация диалекта через свойства Hibernate:

1. Инкапсуляция ограничений и логики работы с типом
2. Выделение субдоменов на уровне таблицы
3. Database first как часть дизайна приложения

Composite types

Enumerated types

Domain types

```
create type transaction_record_t as (
    amount NUMERIC(20, 2),
    senders varchar(512)[],
    reciviers varchar(512)[]
);

create domain transaction_record_d as transaction_record_t
check ( (value).amount > 0 and cardinality((value).senders) > 0 and cardinality((value).reciviers) > 0)
```

Композитные типы: модели JPA

Типовой подход: @Struct + @Embeddable + @Embedded

```java
@Data
@AllArgsConstructor
@NoArgsConstructor
@Embeddable
@Struct(name = "description", attributes = {"id", "amount", "sender", "receiver"})
public class StockRecordDescription implements Serializable {
    private Long id;
    private Double amount;
    private String sender;
    private String receiver;
}
```

Композитные типы: модели JPA

```java
@Getter
@Setter
@Entity
@NoArgsConstructor
@AllArgsConstructor
@Table(name = "stock_record")
public class StockRecord {

    @Id
    @GeneratedValue(strategy = GenerationType.UUID)
    private UUID uuid;

    @Embedded
    private StockRecordDescription description;
}
```

Композитные типы: модели JPA

stock_record | Enter a SQL expression to filter results (use Ctrl+Space)

Grid | Text

	uuid	description				status
		id	amount	sender	receiver	
1	38f0c76a-0ef5-4b80-84ae-ce9dfe284fa9	1	200	Cell 20	Cell 1	COMPLETED

Больше контроля над трансфером типов

```
public class StockRecordCompositeType implements CompositeUserType<StockRecordDescription> {
    @Override
    public Object getPropertyValue(StockRecordDescription component, int property) throws HibernateException {
        switch (property) {
            case 1:
                return component.getId();
            case 0:
                return component.getAmount();
            default:
                throw new IllegalArgumentException(property +
                        " is an invalid property index for class type " + component.getClass()
                        .getName());
        }
    }

    @Override
    public StockRecordDescription instantiate(ValueAccess values, SessionFactoryImplementor sessionFactory) {
        return new StockRecordDescription(values.getValue(1, Long.class),values.getValue(0, Double.class),
                values.getValue(2, String.class), values.getValue(3, String.class));
    }

    // ... omitted methods and logic
}
```

Enumerated types

Зачем нам перечисления в БД?

1. Несколько сервисов используют одну базу данных
2. Контроль над значениями перечислений на уровне БД
3. Использование БД для интеграции между приложениями

Enumerated types

Было бы хорошо, если не так много кода

```
create type stock_record_status as ENUM('COMPLETED', 'STARTED', 'FAILED');

CREATE CAST (varchar AS stock_record_status) WITH INOUT AS IMPLICIT;
```

Enumerated types

Было бы хорошо, если не так много кода

```
1  public class StockRecordStatusType implements UserType<StockRecordStatus> {
2
3      @Override
4      public StockRecordStatus nullSafeGet(ResultSet rs, int position, SharedSessionContractImplementor session,
5          return StockRecordStatus.valueOf(rs.getString(position));
6      }
7
8      @Override
9      public void nullSafeSet(PreparedStatement st, StockRecordStatus value, int index, SharedSessionContractImpl
10         st.setObject(index, value, SqlTypes.OTHER);
11     }
12
13     // ... some methods was omitted
14 }
```

JSONB для JPA: вступление

Зачем нам json в реляционной СУБД:

1. Модель данных неопределена или подвержена изменениям
2. Необходимо провести миграцию данных между СУБД
3. Повышенная скорость доступа к вложенным структурам данных
4. Большое разнообразие нативных функций в СУБД для работы с jsonb
5. Оставляем UI-логику на фронте
6. В некоторых случаях скорость вставки выше, чем у реляционных данных

JSONB для JPA: downsides

Какие недостатки мы встретили:

1. Повышенные требования к объему точек монитрования pg data и pg walarchive
2. GIN-индекс может занимать до 40% от размера таблицы
3. Агрегирующие запросы с использованием jsonb — долго и дорого
4. UPDATE jsonb данных из java-приложений — долго и неудобно

Необходимо выделять поля в отдельные колонки, по которым ведется поиск и индексация.

JSONB для JPA: Types Binding

Что рекомендуется использовать:

- Из коробки JdbcTypeCode + SqlTypes.JSON для простых случаев
- hypersistence-utils от Vlada Mihalcea для случаев посложнее
- PGObject из JDBC и ObjectMapper от Jackson Project для любителей DIY

JSONB binding out the box

Binding типа jsonb из коробки доступен с версии Hibernate 6.0

```java
@Entity
public class UserFiler {

    @Id
    @GeneratedValue
    private Long id;

    @JdbcTypeCode(SqlTypes.JSON)
    private Filter content;

    ...
}
```

JSONB binding by hypersistence-utils

Open Source проект hypersistence-utils позволяет проводить binding с использованием Hibernate аннотации @Type

```
@Entity
@NoArgsConstructor
@AllArgsConstructor
public class DocumentAttribute {

    ...

    @Type(JsonType.class)
    @Column(name = "attributes_value", columnDefinition = "jsonb")
    private DocumentAttributesValue attributesValue;
}
```

JSONB functions and operators binding

Для регистрации функций и паттернов с операторами следует использовать FunctionContributions, которые доступен из переопределенного метода initializeFunctionRegistry класса Dialect.

Доступны два способа регистрации:

Pattern based для render одного и того же паттерна в различных СУБД

Function descriptor использует прямое описание SQM-функции с валидацией и resolve аргументов и возвращаемого значения

JSONB functions and operators binding

Как выглядят расширения диалекта в контексте определения функций и операторов:

```java
public class PostgreSQLDialectCustomized extends PostgreSQLDialect {
    @Override
    public void initializeFunctionRegistry(FunctionContributions functionContributions) {
        super.initializeFunctionRegistry(functionContributions);
        BasicTypeRegistry basicTypeRegistry = functionContributions.getTypeConfiguration().getBasicTypeRegistry
        var typeConfiguration = functionContributions.getTypeConfiguration();
        functionContributions.getFunctionRegistry().registerPattern("jsonContains", "(?1::jsonb @> ?2::jsonb)",
                basicTypeRegistry.resolve( StandardBasicTypes.BOOLEAN ));
    }
}
```

JSONB functions and operators binding

```java
public class PostgreSQLDialectCustomized extends PostgreSQLDialect {
    @Override
    public void initializeFunctionRegistry(FunctionContributions functionContributions) {
        super.initializeFunctionRegistry(functionContributions);
        BasicTypeRegistry basicTypeRegistry = functionContributions.getTypeConfiguration().getBasicTypeRegistry
        var typeConfiguration = functionContributions.getTypeConfiguration();
        functionContributions.getFunctionRegistry().register("jsonContainsTrue",
                new JsonbContainsSQM(
                        "jsonContainsTrue",
                        new ArgumentTypesValidator(
                                StandardArgumentsValidators.exactly(2),
                                FunctionParameterType.ANY, FunctionParameterType.ANY
                        ),
                        StandardFunctionReturnTypeResolvers.invariant(
                                typeConfiguration.getBasicTypeRegistry().resolve( StandardBasicTypes.BOOLEAN )
                        ),
                        StandardFunctionArgumentTypeResolvers.invariant( typeConfiguration, ANY, ANY )
                )
        );
    }
}
```

Вывод данных из JSONB

Когда следует хранить значения вне jsonb колонки:

1. Всегда храним primary key отдельно от самого значения и, желательно, ключ содержит данные из jsonb
2. Предполагается, что будут использоваться группировки и агрегирующие функции — min, max, sum и т. д.
3. Значение чисел на <, >, = и т. д.

Вывод данных из JSONB on the fly

Используем Generated колонки:

```
1  public class DocumentAttribute {
2      @Id
3      @Generated(sql = "attributes_value -> 'id'", writable = true)
4      @Column(columnDefinition = " varchar(512) GENERATED ALWAYS AS ((attributes_value -> 'id')::text::varchar) "
5                                 + "stored not null")
6      private String id;
7
8      @Type(JsonType.class)
9      @Column(name = "attributes_value", columnDefinition = "jsonb")
10     private DocumentAttributesValue attributesValue;
11 }
```

JSONB когда нет json-схемы

Схема DDL недоступна или требуется хранить любой валидный json:

```java
public class DocumentAttributeAny {
    @Id
    @Generated(sql = "attributes_value -> 'id'", writable = true)
    @Column(columnDefinition = " varchar(512) GENERATED ALWAYS AS ((attributes_value -> 'id')::text::varchar) "
        + " stored not null")
    private String id;

    @Type(JsonbObjectMapSQLType.class)
    @Column(name = "attributes_value", columnDefinition = "jsonb")
    private Map<String, Object> attributesValue;
}
```

JSONB когда нет json схемы

```java
@SuppressWarnings("rawtypes")
public class JsonbObjectMapSQLType implements UserType<Map> {
    private final static ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public Map nullSafeGet(ResultSet rs, int position,
                           SharedSessionContractImplementor session, Object owner) throws SQLException {
        try {
            return objectMapper.readValue(rs.getString(position), Map.class);
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
    }

    ...
}
```

JSONB когда нет json схемы

```java
@SuppressWarnings("rawtypes")
public class JsonbObjectMapSQLType implements UserType<Map> {
    private final static ObjectMapper objectMapper = new ObjectMapper();

    @Override
    public void nullSafeSet(PreparedStatement st, Map value, int index, SharedSessionContractImplementor sessio
        PGobject pGobject = new PGobject();
        pGobject.setType("jsonb");
        try {
            pGobject.setValue(objectMapper.writeValueAsString(value));
        } catch (JsonProcessingException e) {
            throw new RuntimeException(e);
        }
        st.setObject(index, pGobject);
    }

    ...
}
```

Массивы PostgreSQL для JPA

@ElementCollection

- Создает отдельную таблицу для хранения значений
- Хранит только базовые типы
- Lazy loading из коробки

Определенные пользователем массивы PostgreSQL в JPA

- Доступны для хранения любого значения
- Операторы GIN
- Только Eager loading, проекции и дополнительные конструкторы для Lazy loading

Где массивы пригодятся

Хранение однотипных базовых типов
(телефоны, почтовые адреса и т. д.)

Серия значения с гарантированным порядком
(tuple или range представленный массивом)

Массивы jsonb для хранения UI данных
(сохраненные фильтры, настройки)

Массивы PostgreSQL для JPA

Объявление поля с типом массив:

```java
@Entity
public class Employee {
    @Id
    private Long id;

    private String fullName;

    @ElementCollection(fetch = FetchType.EAGER)
    private List<String> phoneNumber;

    @Type(StringArraySQLType.class)
    @Column(columnDefinition = "varchar(256)[]")
    private String[] email;
}
```

Массивы PostgreSQL для JPA

Определение типа поля:

```
1  public class StringArraySQLType implements UserType<String[]> {
2      @Override
3      public String[] nullSafeGet(ResultSet rs, int position, SharedSessionContractImplementor session,
4                                  Object owner) throws SQLException {
5          Array array = rs.getArray(position);
6          return array ≠ null ? (String[]) array.getArray() : null;
7      }
8
9      @Override
10     public void nullSafeSet(PreparedStatement st, String[] value, int index,
11                             SharedSessionContractImplementor session) throws SQLException {
12         st.setArray(index, st.getConnection().createArrayOf("text", value));
13     }
14
15     ...
16 }
```

Полнотекстовый поиск

Зачем, когда есть Pattern Matching?

1. Скорость поиска
2. Лингвистическая поддержка (поиск слов по корню и т. д.)
3. Ranking результатов поиска
4. Hint по результатам анализа документов

Полнотекстовый поиск

Внедрение функционала tsquery в JPA:

```java
public interface EmailRepository extends JpaRepository<Email, Long> {
    @Query("SELECT e FROM Email e WHERE ts_query_perform(e.lexemes, ?1)")
    List<Email> fullTextQuery(String query);
}

public class PostgreSQLDialectCustomized extends PostgreSQLDialect {
    @Override
    public void initializeFunctionRegistry(FunctionContributions functionContributions) {
        super.initializeFunctionRegistry(functionContributions);
        BasicTypeRegistry basicTypeRegistry = functionContributions.getTypeConfiguration()
                                                                   .getBasicTypeRegistry();
        var typeConfiguration = functionContributions.getTypeConfiguration();

        functionContributions.getFunctionRegistry().registerPattern(
                "ts_query_perform", "?1 @@ ?2::tsquery",
                basicTypeRegistry.resolve( StandardBasicTypes.BOOLEAN )
        );
    }
}
```

ITFB Group

+7 (495) 234 61 42

mail@itfbgroup.ru

Холодильный пер. 3, г. Москва,
Россия, 115191