SAP Customer Experience

Flexible Search

SAP Commerce Cloud Developer Training

THE BEST RUN SAP

The Context

SAP Commerce Cloud comes with a FlexibleSearch, a built-in **query language using an SQL-based syntax**. It enables searching for items in SAP Commerce Cloud.

Overview

Overview

- SQL-like syntax
- Abstracts a database query into a Commerce Item query
- Returns a list of objects (SAP Commerce items)
- Makes attributes of SAP Commerce items easily queryable
- Is translated into native SQL statements on execution
- Allows nearly every feature of SQL SELECT statements
- Queries go through cache

Syntax

Syntax

- Basic Syntax:

```
SELECT {attribute1}, {attribute2}, ... {attributeN} FROM {types} (where <conditions>)?
   (ORDER BY <order>)?
```

- Mandatory:

```
SELECT {attribute1}, {attribute2}, ... {attributeN}
FROM {types}
```

- Optional:

```
where <conditions>
ORDER BY <order>
```

- SQL Command / Keywords:

  ASC, DESC, DISTINCT, AND, OR, LIKE, LEFT JOIN, CONCAT, ...

Query examples

- Basic query
  - Special case: returns Car object instead of PK value

```
SELECT {PK} FROM {Car}
```

- Simple queries

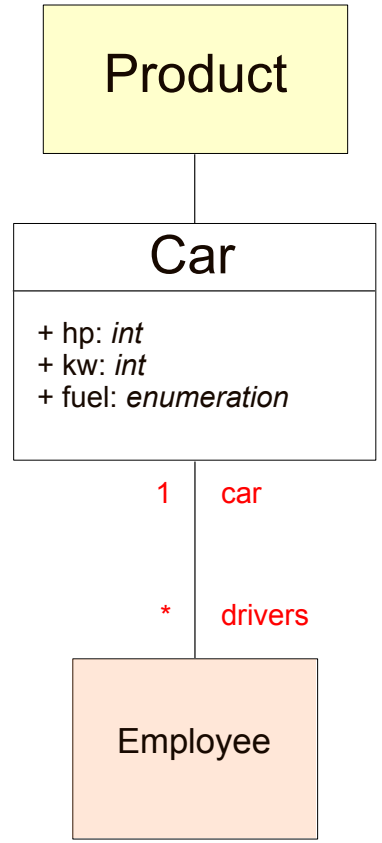
```
SELECT {code}, {hp} FROM {Car}
```

- Single type queries
  - Returns only Product items, not subtypes

```
SELECT {code} FROM {Product!}
```

- Joins

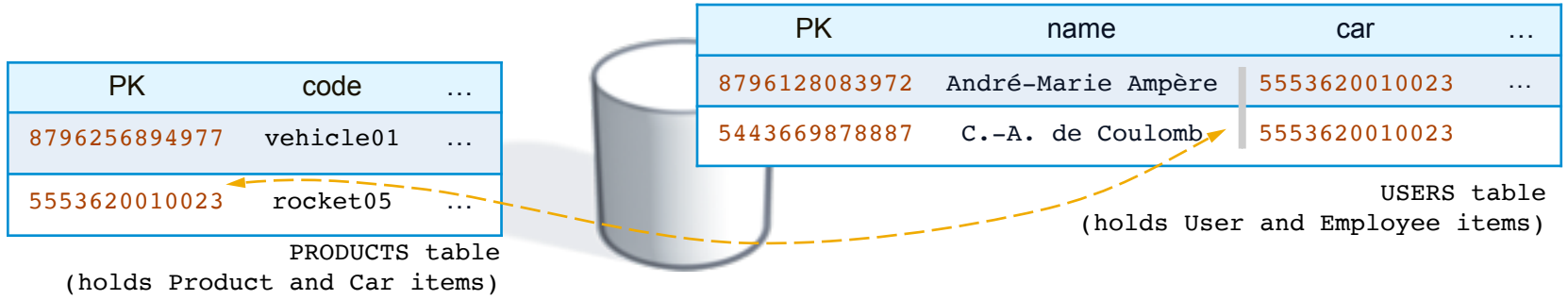
```
SELECT {c.code},{e.uid} FROM {
      Car as c JOIN Employee as e
                ON {c.pk} = {e.car}
  } WHERE {e.uid} LIKE '%Columbo'
```



Joins for One-to-Many Relations

- Recall the following one-to-many relation from a previous chapter:

```
<relation code="Car2DriversRelation" generate="true" autocreate="true" localized="false" >
      <sourceElement qualifier="car" type="Car" cardinality="one" />
      <targetElement qualifier="drivers" type="Employee" cardinality="many"/>
</relation>
```



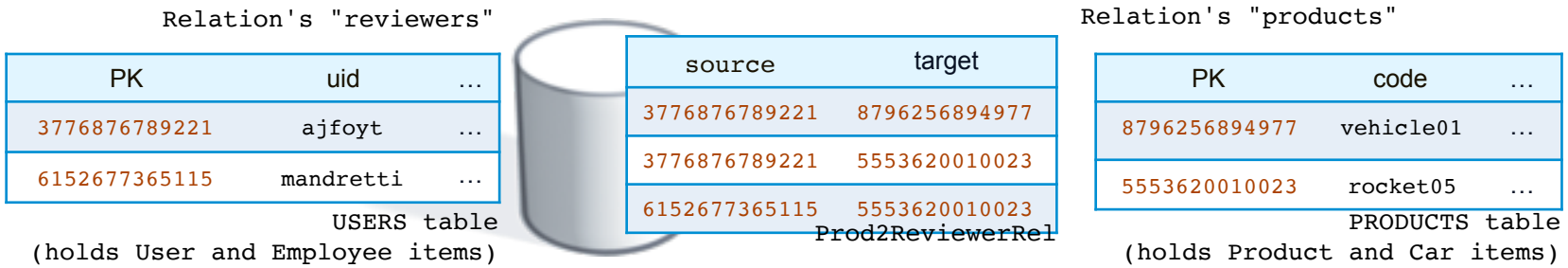
- Based on the table deployment of this kind of relation, the JOIN statement would be:
  (note the use of SAP Commerce types instead of table names)

```
SELECT {c.code} as "Car", {e.name} as "Driver"
FROM {  Employee as e JOIN Car as c
                      ON {e.car}={c.PK} }
```

Joins for Many-to-Many Relations

- Recall the following many-to-many relation from a previous chapter:

```
<relation code="Product2ReviewerRelation" autocreate="true" generate="true" localized="false">
    <deployment table="Prod2ReviewerRel" typecode="20123"/>
    <sourceElement qualifier="reviewers" type="Employee" cardinality="many" />
    <targetElement qualifier="products" type="Product" cardinality="many" />
</relation>
```



- Based on the table deployment of this kind of relation, the JOIN statement would be:
  (note the use of SAP Commerce types instead of table names)

```
select {p.code} as "Product", {u.uid} as "Reviewer"
from { Product as p JOIN Prod2ReviewerRelation as p2r
                    ON {p2r.target}={p.PK}
                    JOIN User as u
                    ON {p2r.source}={u.PK}   }
```

More Query Examples

- Inner queries:

```
SELECT {c.code} FROM {Car as c}
   WHERE {c.mechanic} IN
   (({
      SELECT {PK} FROM {Employee}
         WHERE {uid} LIKE '%Tesla'
    }))
```

- Group functions:
  (notice how only attribute names and SAP Commerce types are enclosed in curly braces **{ }**)

```
SELECT count(*) FROM {Car}
```

Using Dates (When In HAC Flexible Search Console)

- Date literals (specific to underlying DB):

```
SELECT {c.code} FROM {Car as c}
   WHERE {Car.warrantyExpiry} < '2020-12-01 0:00:00.0'
```

- Date functions (specific to underlying DB):
  - E.g., in HSQLDB: CURDATE and TODAY are aliases for CURRENT_DATE (SYSDATE also works)

```
SELECT {c.code} FROM {Car as c}
  WHERE {Car.warrantyExpiry} < TODAY
```

API Examples

Querying for SAP Commerce items

In Java, we normally query for the entire item, rather than single properties

- Java domain model objects are returned by `FlexibleSearchService.search()` when you select the `{PK}` property (and nothing else)

```
"SELECT {PK} FROM {MyCommerceType}"
```

- Transform the returned `SearchResult` list into a `List<MyCommerceTypeModel>`:

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getAllCars() {
   String queryStr = "SELECT {PK} FROM {Car}";
   FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
   SearchResult<CarModel> result = getFlexibleSearchService().search( fsq );
   List<CarModel> cars = result.getResult();
   return cars;
}
```

  - Or combine the last three statements, using generics:

```
return getFlexibleSearchService().<CarModel>search( fsq ).getResult();
```

Querying For Atomic-Type Parameters

- Parameter-value bindings reference map keys using *?key*
  - For **atomic types**, values map conveniently to Java wrapper-class objects, etc.

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getCarsByHpRange( Integer minHp, Integer maxHp ) {
     String queryStr = "SELECT {PK} FROM {Car} WHERE {hp} >= ?hpMin AND {hp} <= ?hpMax";
     FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
     fsq.addQueryParameter("hpMin", minHp);
     fsq.addQueryParameter("hpMax", maxHp);
     SearchResult<CarModel> result = getFlexibleSearchService().search( fsq );
     List<CarModel> cars = result.getResult();
     return cars;
}
```

Querying For Non-Atomic Parameters

- Parameter-value bindings reference map keys using *?key*
  - For **non-atomic SAP Commerce types**, item references (PKs) map conveniently to Java object references

```
import de.hybris.platform.servicelayer.search.SearchResult;
. . .
public List<CarModel> getCarsByMechanic( EmployeeModel mechanic ) {
     String queryStr = "SELECT {PK} FROM {Car} WHERE {mechanic} = ?mechanic";
     FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
     fsq.addQueryParameter("mechanic", mechanic);
     SearchResult<CarModel> result =
              getFlexibleSearchService().search( fsq );
     List<CarModel> cars = result.getResult();
     return cars;
}
```

Querying for non-Model objects (1-column queries)

- The data type for the single column must be mapped in the form of a 1-element `List<Class>`
  - For a 1-column query, `result.getResult()` still returns a `List<Object>`
  - The `List` is `ArrayList<valueClass>` where each entry is the sole value object representing a result "row"

```
String queryStr = "SELECT {vin} FROM {Car}";
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Class[] resultTypesArray = { String.class };
fsq.setResultClassList( Arrays.asList( resultTypesArray ) );
SearchResult<String> result =  getFlexibleSearchService().search( fsq );
List<String> vinList = result.getResult();
for( String carVin : vinList ) {
     logger.info( "VIN: " + carVin );
}
```

Try to use Models whenever it makes sense

Querying for non-Model objects (2+-column queries)

- The column data types must be mapped positionally in the form of a List<Class>
  - For a 2-or-more-column query, `result.getResult()` returns a `List< List<Object> >`
  - Each element of the outer List represents a “query-result row”;
  - Each “query-result row” is a positional List of column values

```
String queryStr = "SELECT {vin}, {weight} FROM {Car}";
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Class[] resultTypesArray = { String.class, Integer.class };
fsq.setResultClassList( Arrays.asList( resultTypesArray ) );
SearchResult< List<Object> > result =  getFlexibleSearchService().search( fsq );
List< List<Object> > resultRowList = result.getResult();
for( List<Object> columnValuesForRow : resultRowList ) {
     final String vin = (String)columnValuesForRow.get(0);
     final Integer weight = (Integer)columnValuesForRow.get(1);
     System.out.println( "Car with vin " + vin + " weighs " + weight.intValue() + "
kg." );
}
```

Querying against today's date • Caching Considerations

- When comparing with today's date, truncate date value
  - Every Flexible Search query is cached, but using the current date/time value — which changes every millisecond — has the effect of never being able to reuse the cached query result
  - Truncate date as needed — for example, to nearest day:
    (if the attribute was meant to be date-only, you would want to truncate the time portion)

```
String queryStr = "SELECT {PK} FROM {Car} WHERE {Car.warrantyExpiry} < ?today";
final Calendar cal = Calendar.getInstance();  //initializes to current system time
   cal.set(Calendar.HOUR_OF_DAY, 0);
   cal.set(Calendar.MINUTE, 0);
   cal.set(Calendar.SECOND, 0);
   cal.set(Calendar.MILLISECOND, 0);   //VERY easy to forget to zero-out milliseconds
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
Date todayDate = cal.getTime();  // need an instance of java.util.Date for query param
value
fsq.addQueryParameter("today", todayDate );
SearchResult<CarModel> searchResult = getFlexibleSearchService().search( fsq );
List<CarModel> cars = searchResult.getResult();
```

Pagination

- Paginate to reduce data-transfer bandwidth

```
final int PAGE_SIZE = 5;
String queryStr = "SELECT {PK} FROM {Car}";
FlexibleSearchQuery fsq = new FlexibleSearchQuery( queryStr );
fsq.setNeedTotal( true );
fsq.setCount( PAGE_SIZE );
. . .
List<CarModel> pageOfCars getCarsByPage( fsq, desiredResultPage ); //reuse fsq for all
pages
. . .
List<CarModel> getCarsByPage( FlexibleSearchQuery fsq, int pageNum ) {
    fsq.setStart( (pageNum – 1) * PAGE_SIZE );
    //Note the Java Generics syntax when both lines combined into one
    return getFlexibleSearchService().<CarModel>search( fsq ).getResult();
}
```

Referring to Model Attributes • Failsafe Approach

- When building a query in Java code, use the static constants defined in each model class to refer to its attribute names
  - Each attribute's name is held by a corresponding all-upper-case constant
  - The name of the SAP Commerce type represented by this model class is contained by `_TYPECODE`
  - Using constants makes your code more difficult to read, but make it impossible to misspell attribute names without causing an immediate compilation error in your DAO classes

- For example, instead of

```
String queryStr = "SELECT {code}, {hp} FROM {Car}";
```

- Use the static constants

```
String queryStr = "SELECT {" + CarModel.CODE + "}, {" + CarModel.HP + "}"
                + " FROM {" + CarModel._TYPECODE + "}";
```

Flexible Search Alternatives

GenericDao

A helper class that dramatically simplifies basic parameter searches

- Use DefaultGenericDao as an alternative to Flexible Search
  - Configure target item type in constructor
  - Tip: use Spring Expression Language shortcut to refer to Model's static typecode variable

```
<bean name="productDao" class="de.hybris.platform.servicelayer.internal.dao.DefaultGenericDao">
    <constructor-arg value="#{T(de.hybris.platform.core.model.product.ProductModel)._TYPECODE}" />
</bean>
```

  - Perform basic search using Map of parameter name/value pairs
    - As a very simple example, to return all products of a particular weight:

```
public DefaultGenericDao<ProductModel> productDao;      // Spring bean ref. injected using setProductDao(...)
. . .
public List<ProductModel> getProductsByWeight( int weight ) {

    final Map<String, Object> params = new HashMap<>();
    params.put(ProductModel.WEIGHT, Integer.valueOf(weight) );
    List<ProductModel> products = productDao.find(params); // Note how productDao.find() returns desired type
    return products;
}
```

GenericSearch

- Similar to *HibernateCriteriaSearches*
- Search for items as well as raw data fields
- Unlimited number of conditions
- Inner joins and outer joins between item types possible
- Unlimited number of “order by” clauses
- Sub-selects supported

Better performance?

No, because it is translated to a FlexibleSearch query

But it is very strongly typed

GenericSearch Example

```
GenericQuery query = new GenericQuery(CarModel._TYPECODE);
GenericSearchField carField = new GenericSearchField(CarModel._TYPECODE,
CarModel.MANUFACTURER);
GenericCondition condition = GenericCondition.createConditionForValueComparison(carField,
Operator.LIKE,
                                                                    "BMW");
query.addCondition(condition);
query.addOrderBy(new GenericSearchOrderBy(carField, true));  //param ascending=true

SearchResult<CarModel> result = genericSearchService.search(query);
List<CarModel> cars = result.getResult();
```

Flexible Search abstracts a database query into a SAP Commerce Item query

It returns a List of **models** (SAP Commerce items) – except in the HAC

It is **translated into native SQL** statements on execution

Queries go through the **cache** – so try to avoid frequently-changing queries

Use joins for 1:n and n:m relations

Use **`FlexibleSearchService`** to execute queries in Java

In queries, refer to Model Attributes (static constants) for a more failsafe approach

Use **`DefaultGenericDao`** (for simple parameters) as an alternative to Flexible Search

Use **GenericSearch** as even more failsafe and stronger-typed approach

Exercise

Flexible Search

Thank you.