Query Engine

A Pattern for Performing Dynamic Searches in Information Systems

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

Jan 24, 2006

Abstract: This paper presents an architecture pattern for information systems that require complex search capabilities. The pattern includes means to generically describe search requests, a service that interprets search requests and executes them on data sources, and strategies for transmitting results back to the requesting clients.

Context

A typical information system retrieves and manipulates business data in many different ways. Search requests to fetch business data are often created and executed in the middle tier of a three-tier system architecture; sometimes they are dynamically created in a client application. These are a few examples (see Fig. 1):

- Business logic has to be applied to a set of business objects. The objects have to be found and retrieved before they can be modified.
- A client application provides a search dialog that lets users dynamically compose complex search requests. An application server executes the queries on a data source and returns the results to the client application, where they are shown.
- A reporting tool has to find and process data from different sources. Creating the reports may involve operations that are not directly supported by all underlying data sources.

These examples have in common that a flexible mechanism for executing search requests is needed. Since the same search requirements typically apply to different types of business data, a generic mechanism is helpful to avoid redundancies.

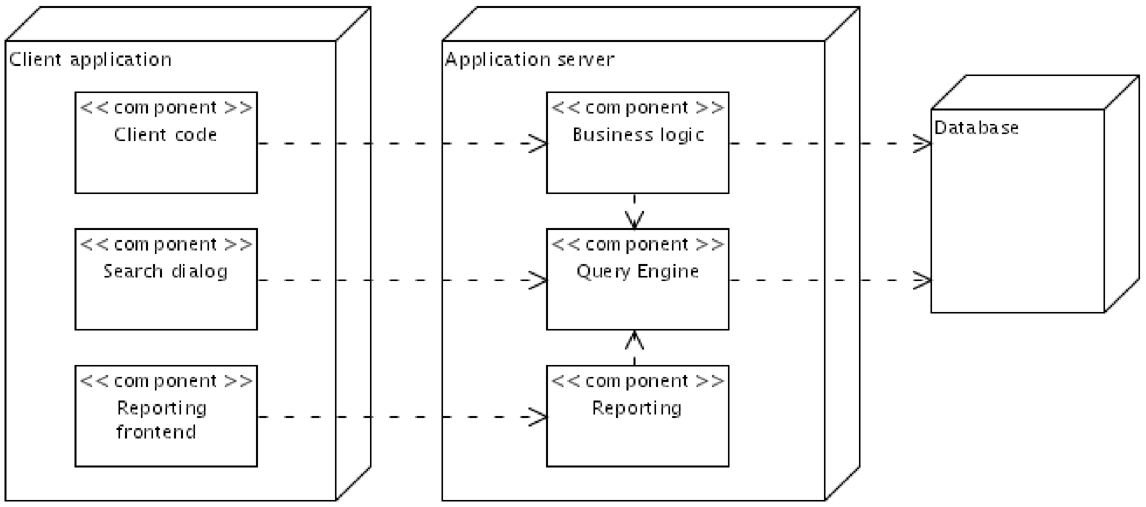

Client application
<< component >> Client code
<< component >> Search dialog
<< component >> Reporting frontend
Application server
<< component >> Business logic
<< component >> Query Engine
<< component >> Reporting
Database

Fig. 1: A system architecture that includes a Query Engine

Problem

An information system needs complex search capabilities. It may be possible to manually write native query statements, execute those queries, evaluate the results, and send the results back to the requesting clients. This approach has severe drawbacks if any of the following *forces* apply:

- *Dynamics.* Some features of an information system require dynamic queries at runtime. In a search dialog for example, users may create queries by arbitrarily combining given search criteria. Although the list of search criteria may be fix, there may be too many valid combinations to implement static, parameterized services for executing the queries.
- *Heterogeneity.* Data sources from different vendors must be supported, sometimes even different types of data sources: relational databases, XML databases, LDAP servers, full-text search engines, or legacy systems. Using separate search libraries with different APIs adds extra complexity to a project.

- *Impedance mismatch.* Queries that refer directly to a data model may be inappropriate. In an object-oriented system, for example, queries should refer to the classes and attributes of an object-oriented domain model that is mapped to tables and columns of a relational database.
- *Loose coupling.* Although client applications may create search requests, the applications should not be bound to database internals.
- *Query syntax.* Writing correct query statements in the native query language may be difficult for an application developer. The data source may have an uncommon query language or there may be too few experts available for writing native queries.
- *Distributed queries.* In a distributed environment, a single query may span over multiple data sources. Instead of executing a query on each system separately and combing the results manually, executing such a query should appear to a client as if there were a single data source.

Solution

Create a QUERY ENGINE, that is a service that takes a description of a search request, evaluates and executes the request, and returns the results back to the caller. This service acts as an intermediate layer between clients and the underlying data sources by interpreting search requests and shielding the clients from details on how to access the data sources.

A Query Engine separates the formulation of individual search requests from their execution. It encapsulates the process how data sources are accessed, how native query statements are formulated, and how those statements are executed. Queries may be created at runtime if they are based on dynamic user input or they may be created at compile-time if they are based on static business requirements.

A Query Engine handles the individual features and limitations of each data source involved while processing search requests and creating native query statements. It may also translate references to a domain model into references to a data model.

A Query Engine might be built as a reusable component or as a customized service. In the former case, a Query Engine can be made independent of specific data sources, specific query languages, and application-specific domain models. In the latter case, some parts of the Query Engine may be library code, whereas other parts may be project code.

Clients may access a Query Engine by performing local procedure calls, remote procedure calls, or by calling a Web Service. The interface of a Query Engine contains classes that represent search requests and classes that wrap results.

In the basic case, the Query Engine pattern has three participants as shown in Fig. 2: Query, Query Engine, and Result Set.

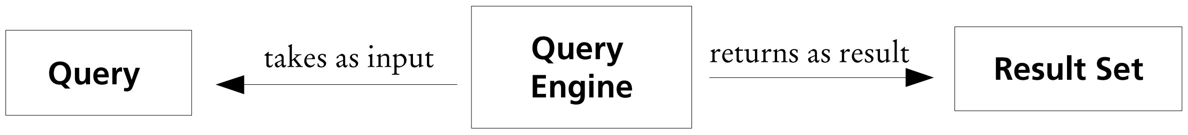


Fig. 2: Participants of the basic case

A *Query* describes a search request. It includes the search target, a condition that refers to the data records to be searched, as well as additional options.

The *Query Engine* performs a search request: it translates a Query into a native query statement, executes it on a data source, and returns a Result Set.

A *Result Set* encapsulates the results retrieved by the Query Engine.

Consequences

A Query Engine introduces a layer of abstraction above the data sources. This has the following *advantages*:

Since Queries are interpreted, they may be created at runtime. It is possible to change parameters of a hard-coded Query and to create Queries based on input that users enter in a search dialog.

A Query Engine abstracts from how data is retrieved from a data source. If a database system is replaced by a different one, creating a new Query Engine implementation might be sufficient – no Queries need to be changed.

A Query Engine separates business logic from technical details on how search requests are executed. Application developers may concentrate on implementing business requirements without knowing details about the data sources.

A Query Engine may provide features that are missing in a specific data source. It may emulate those features so that the application developers don't need to implement manual workarounds. If search requests span over multiple databases, a Query Engine is responsible for executing separate native queries and combining their results.

In object-oriented projects, sometimes it is necessary to execute native query statements directly on a database. A Query Engine is able to transparently support both Queries that refer to an object-oriented domain model and Queries that refer directly to database tables and columns.

There are also the following *drawbacks*:

Developers that join a project have to learn how a Query Engine works and how they implement search requests. Even if they know the native query language well, it takes time to get used to new search facilities.

It might be difficult and time-consuming to provide abstractions for all features available in a database system. If there is only a single database system involved that is unlikely to change, the benefit of abstraction is reduced.

Some features of a Query Engine are costly to implement. If those features are needed only a couple of times, they may not be worth the effort.

Developers may create Queries that show performance problems because the developers are not conversant enough with limitations of the underlying database system.

Adding abstractions to features already provided by and optimized for a database system may additionally degrade performance.

In some cases, using a data warehouse system may solve the problem better. Such a system provides a common layer for requesting information from different data sources.

Known uses

The Query Engine pattern has been used in several industry projects, both for rich client and for web applications. The author has worked with two Query Engine implementations and has himself developed another one.

In a project for a German television company, a Query Engine was built as a transparent layer to hide disparate data sources from client applications. Queries were performed on two different relational database systems and on a full-text search engine. The implementation facilitated Queries that spanned over multiple data sources.

In a project for another television company, a Query Engine was used both as a back-end for dynamic search dialogs and for retrieving business objects from a relational database. The search dialog presented a simplified structure of the business data that could be queried. The Query Engine modified the search requests appropriately before executing them.

Related Work

In his book "Patterns of Enterprise Application Architecture" [Fowler], Martin Fowler describes the Query Object pattern. In his words, "a Query Object is an interpreter [GoF], that is, a structure of objects that can form itself into a SQL query. You can create this query by referring to classes and fields rather than tables and columns."

A shortcoming of that approach is that the Query Object itself creates the native query statement. Therefore, the pattern shouldn't be used in client applications because it would bind the applications to the query language of the data source. Furthermore, this approach does not support heterogeneous data sources and may cause redundancies if different kinds of Queries are needed.

The term “Query Engine” is widely used in the area of searching in XML data. There it describes mechanisms that enable searching by applying an XML search language such as XPath. These Query Engines normally do not address the forces that are essential to this pattern.

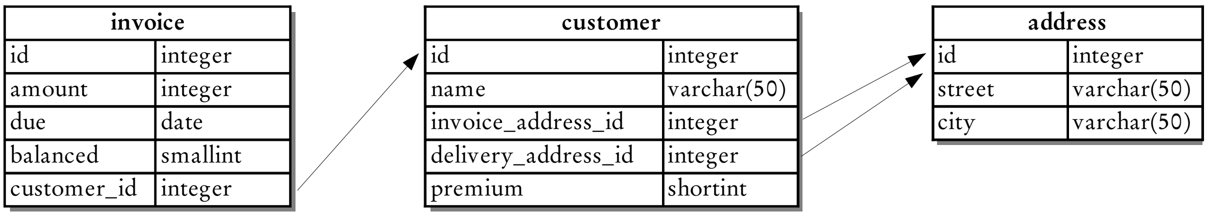
Implementation

This section describes the implementation of a Query Engine. Please note that there are many ways how a Query Engine can be built. This paper presents one implementation that resolves most of the forces. This section introduces the implementation for a basic Query Engine; additional variations follow later on.

Running Example

All examples are based on a simple case study: a company needs a customer relationship management (CRM) system that keeps track of the company's customers and invoices. A client application should provide a search dialog for retrieving data about customers, and it should show the results in a table on the screen. The data is stored in a relational database that uses SQL as query language. Some particular features require business logic that is stored in an application server. The programming language is Java.

The data model consists of the database tables that are shown in Fig. 3: customer, address, and invoice. For each customer, there is a delivery and an invoice address and a list of invoices. A variation for an object-oriented domain model is presented later on.



invoice	
id	integer
amount	integer
due	date
balanced	smallint
customer_id	integer

customer	
id	integer
name	varchar(50)
invoice_address_id	integer
delivery_address_id	integer
premium	shortint

address	
id	integer
street	varchar(50)
city	varchar(50)

Fig. 3: Data model for the CRM example

Query

A Query is the representation of a search request. It consists of the following parts:

- The *search target* defines the type of data requested (the name of a database table, for instance). The results of a Query contain only data records from the requested search target.
- The *condition* defines the properties of the data records to be returned (for example the valid values of a database column). The condition can be expressed in attributional logic: the Result Set will only contain those business data records of the search target type against which the condition holds.
- An optional *filter* defines which elements of the data records have to be returned to the client. A filter may list all required database columns of the search target and of tables associated with the search target.
- Additional search *options* may include sort parameters and the maximum result set size.

The condition is the most difficult part in designing a Query. There are at least two concepts:

- The condition is expressed in text form with Boolean expressions. This form is often expected by full text search engines.
- The condition is a graph of objects that represent the structure of the condition. Typically, these objects form a tree; the COMPOSITE pattern [GoF] may be applied.

The class diagram in Fig. 4 shows the structure of a Query implementation that follows the latter case[1].

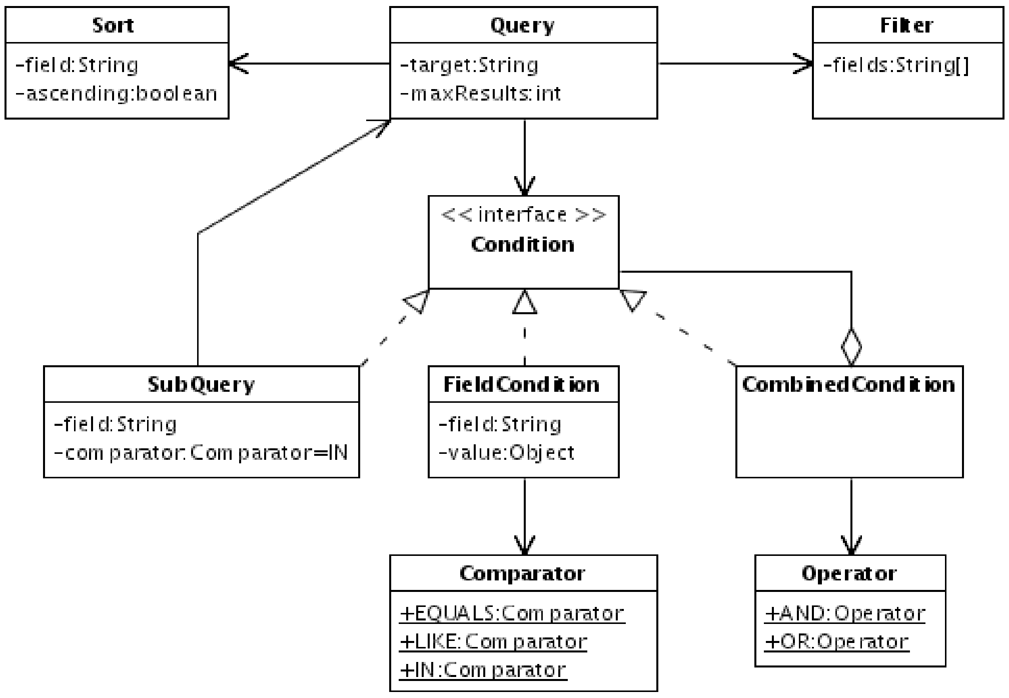


Fig. 4: UML class diagram showing a Query implementation

An object of the main class, `Query`, takes the name of the search target, the maximum number of results, a reference to the condition of the search request, a filter, and sort options. A `Condition` object is either a comparison of a field with a value (`FieldCondition`), a collection of conditions combined by a Boolean operator (`CombinedCondition`), or a sub query (`SubQuery`). A `Filter` instance takes a list of those columns the client expects to retrieve from the server for each business data record.

`FieldCondition` instances refer to columns of the search target table. Associations to other tables are handled by `SubQuery` instances. Sub queries are an abstraction of an association in the data model. They do not determine whether they are resolved by using an SQL sub select statement or by joining the involved tables.

The CRM client application has to show a list of all invoice statements that are not balanced but are due within a period given as parameters by the user. For each data record found, only the amount and the due date should be returned, sorted by the due date.

The following code creates the accordant Query. Such a static Query is normally part of the business logic. It is not created in the client application in order to avoid coupling the client application with details of the data model.

```
private Query createQueryForDueInvoices(Date firstDate, Date lastDate) {

  Query query = new Query("invoice");
  query.setMaximumResults(100);

  query.setFilter(new Filter(new String[]{"id", "amount", "due"}));
```

1 Note that some elements have been left out to simplify the diagram, e.g. additional comparators and operators.

```
  query.setSort(new Sort("due", false));

  CombinedCondition condition = new CombinedCondition(Operator.AND);
  condition.addCondition(
    new FieldCondition("balanced", Comparator.EQUALS, new Integer(0)));
  condition.addCondition(
    new FieldCondition("due", Comparator.GREATER_EQUALS, firstDate));
  condition.addCondition(
    new FieldCondition("due", Comparator.LESS_EQUALS, lastDate));
  query.setCondition(condition);

  return query;
}
```

As an example for a sub query, suppose that only those invoices should be listed that are not from premium customers. To fulfill this requirement, the application developer adds a sub query to include only customers that are not premium customers:

```
private Query createQueryForDueInvoicesNoPremiumCustomers(
  Date firstDate, Date lastDate)
{
  Query invoiceQuery = createQueryForDueInvoices(firstDate, lastDate);

  Query query = new Query("customer");
  query.setFilter(new Filter(new String[]{"id"}));
  query.setCondition(
    new FieldCondition("premium", Comparator.EQUALS, "0"));
  SubQuery subQuery = new SubQuery("customer_id", query);

  invoiceQuery.addCondition(subQuery);

  return invoiceQuery;
}
```

The Query implementation presented in this section has both advantages and drawbacks. For a Java programmer, writing Queries may be easier than writing SQL code. The developer does not need to know the escape syntax for passing values likes Date objects to the database if he does not use `PreparedStatements`, for instance. On the other hand, `FieldCondition` objects refer to database columns and therefore couple the code to the data model as does plain SQL. In this case, the variation *Object-Oriented Domain Model* helps.

The example Queries map easily to SQL. A Query Engine may nevertheless provide operations that do not map directly to SQL but involve additional source code, for example to perform complex calculations before or after a SQL statement is executed. It is the responsibility of the Query Engine to hide such complexity from the client and to provide hooks to integrate those functions.

Query Engine

The Query Engine takes Queries, creates equivalent SQL statements, executes the SQL statements, retrieves the results, and returns the results wrapped in a Result Set. Designed as a service, it is a FACADE [GoF] for all clients that need to execute search requests. Those clients may be rich-client applications or server components with business logic.

The Query Engine provides an interface for its clients as shown in Fig. 5.

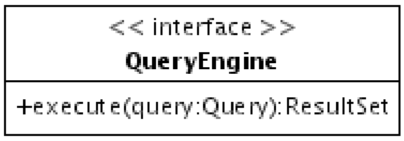


Fig. 5: The interface `QueryEngine`

Given the `Query` object for due invoices that was created in the previous section, the translation to SQL is straightforward. The Query Engine inspects each part of the Query object and generates corresponding SQL code.

- The target of the `Query` becomes part of the 'from' clause: `from invoice`.
- The `Query` has a filter that restricts the results to three columns. Additionally, a result set limit is defined. The following 'select' clause is created: `select top 100 id, amount, due`.
- Transforming the `Condition` object involves visiting the nodes of the `Condition` object tree, using either the Iterator or the Visitor pattern [GoF]. The example contains three `FieldCondition` objects that are combined by an AND operator. Transformed into SQL, a 'where' clause is generated: `where balanced = 0 and due >= '2004-07-07' and due <= '2004-07-11'`.
- Finally, the sort option has to be transferred into SQL: `order by due desc`

Putting all parts together, the Query Engine creates the following SQL statement:

```
select top 100 id, amount, due
from invoice
where balanced = 0 and
      due >= '2004-07-07' and
      due <= '2004-07-11'
order by due desc
```

How this statement is executed and how the data is retrieved depends on the actual system architecture. The CRM system has to show the results in a graphical table component. In that case, a Data Transfer HashMap [Marinescu] is a reasonable choice: for each data record, a hash map is created and filled with the values of the record; their column names as used as keys (the code of the following example does not include exception handling and proper handling of JDBC resources; the class `SimpleResultSet` is introduced in the next section):

```
QueryEngineBean.java:

private ResultSet executeStatement(String sqlStatement, Filter filter) {

  // JDBC driver executes SQL statement
  java.sql.ResultSet sqlResultSet =
    getConnection().createStatement().executeQuery(sqlStatement);

  // Fetch results row by row and create Result Set
  java.util.List results = new java.util.ArrayList();
  while (sqlResultSet.hasNext()) {
    sqlResultSet.next();
    Object dataRecord = retrieveDataRecord(sqlResultSet, filter);
    results.add(dataRecord);
  }
  ResultSet resultSet = new SimpleResultSet(results);

  return resultSet;
}

private Map retrieveDataRecord(java.sql.ResultSet resultSet,
                               Filter filter) {

  // Put all columns' values into a hash map as defined by the filter
  Map dataTransferHashMap = new HashMap();
  for (int i = 0; i < filter.size(); i++) {
    String columnName = filter.get(i);
    Object value = resultSet.get(columnName);
    dataTransferHashMap.put(columnName, value);
  }

  return dataTransferHashMap;
}
```

The CRM client application has to show the results in a table: A `javax.swing.TableModel` implementation acts as an ADAPTER [GoF] between a `javax.swing.JTable` instance and the `ResultSet` object.

The implementation of the `QueryEngine` interface and the FACADE [GoF] for retrieving the list of due invoices are implemented as follows (the actual transformation of the `Query` object into SQL is left out because of its length):

```
QueryEngineBean.java:

public class QueryEngineBean implements QueryEngine {

  public ResultSet execute(Query query) {
    String sqlStatement = createSqlStatement(query);     // Left out here
    ResultSet resultSet =
      executeStatement(sqlStatement, query.getFilter());
    return resultSet;
  }
}

BusinessFacadeBean.java:

public ResultSet getDueInvoices(Date firstDate, Date lastDate) {
  Query query = createQueryForDueInvoices(firstDate, lastDate);
  ResultSet resultSet = getQueryEngine().execute(query);
  return resultSet;
}
```

The Query Engine implementation is bound to relational databases and SQL as native query language. The variations *Query Transformer* and *Query Performer* present an approach to integrate arbitrary data sources.

Transforming a Query directly into SQL makes it difficult to optimize the native query statement. If it is possible to express optimizations by rules that can be applied consistently to all Queries, each Query may be analyzed and modified before it is transformed into the native query statement (see the variation *Query Transformer*).

Result Set

A Result Set is a wrapper for objects that are returned to a client as the result of the execution of a Query.[2] Such results may be stored in plain collection objects, for example in instances of `java.util.ArrayList`. An explicit interface, however, provides the opportunity to return specialized implementations.

Fig. 6 shows an interface for a Result Set that provides operations to retrieve individual result set elements and sub lists of result elements. The basic implementation (`SimpleResultSet`) only carries a list of objects.

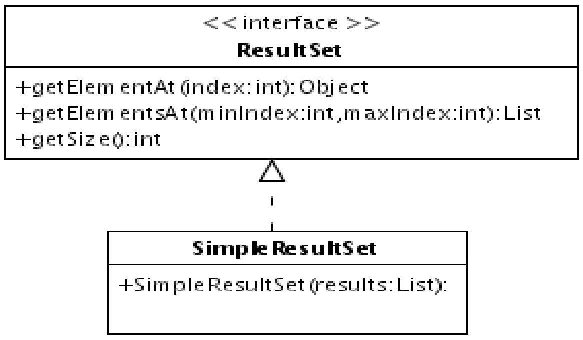

<< interface >>
ResultSet
+getElementAt(index:int):Object
+getElementsAt(minIndex:int,maxIndex:int):List
+getSize():int
SimpleResultSet
+SimpleResultSet(results:List):

Fig. 6: The interface `ResultSet` *and its implementation* `SimpleResultSet`

The interface `ResultSet` does not require a common interface or super class for the elements it transfers. Explicit interfaces, for example `BusinessObject` and `BusinessObjectList`, are a good choice, if all result set elements share common functionality.

The simple Result Set implementation instantly transfers all data records found by executing a Query. For a client application that needs all results at once, this mode is appropriate. On the other hand, a web application that shows results page by page needs a different solution. An approach to solve this and similar requirements is discussed in the variation *Remote Result Sets*.

2 *Result Set* is different from the JDBC class `ResultSet`: it is rather an abstraction from the mechanism to retrieve results.

Variations

This section introduces four variations[3]:

- A *Query Transformer* encapsulates how the Query Engine creates a native query statement.
- A *Query Performer* encapsulates how the Query Engine executes a native query statement and how it creates a Result Set.
- An *Object-Oriented Domain Model* can be used as an alternative to a relational data model.
- *Remote Result Sets* are variations of how results are transferred to the requesting clients.

Query Transformer

Transforming a Query into a native query statement can be done inside the core of the Query Engine by iterating over all of its objects and transforming each one directly. A more sophisticated approach is useful if any of the following constraints applies:

- The Query Engine has to support multiple native query languages or dialects.
- A Query should be optimized automatically.

A Query Transformer is a set of classes that correspond to the Query classes: Each transformer object receives information from a Query object and creates a fragment of the native query statement. Fig. 7 shows some of the classes of the Query Transformer for generating native SQL statements.[4]

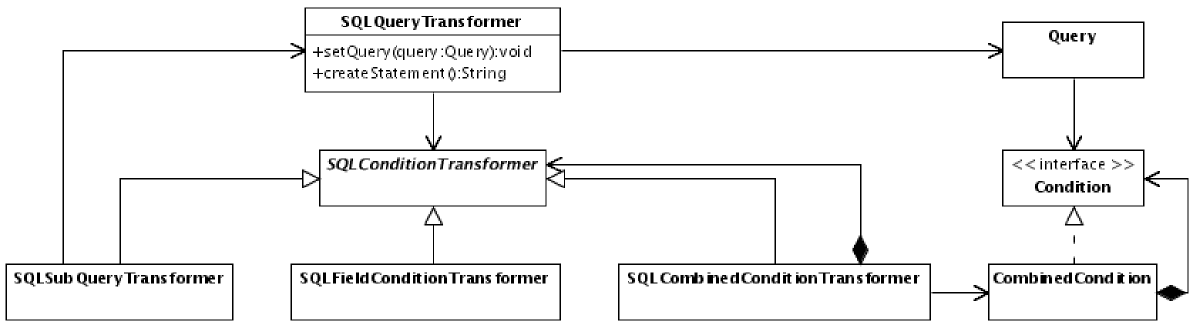

SQLQueryTransformer
+setQuery(query:Query):void
+createStatement():String
Query
SQLConditionTransformer
<<interface>>
Condition
SQLSubQueryTransformer
SQLFieldConditionTransformer
SQLCombinedConditionTransformer
CombinedCondition

Fig. 7: The Query Transformer for SQL

An instance of `SQLQueryTransformer` is initialized with a `Query` instance. The `SQLQueryTransformer` object then creates an instance of an appropriate sub class of `SQLConditionTransformer` that corresponds to the type of the `Condition` object. These classes create further transformer objects accordingly. After initialization, there is a tree of transformer objects whose structure is similar to but not necessarily the same as the tree of query objects.

To create the native SQL statement, the method `SQLQueryTransformer.createStatement` is called. This call is recursively forwarded to all transformer objects. Each transformer object cre-

3 Some of the variations have forces of its own. Whether these variations are actually patterns still has to be worked out.

4 Note that the Query Transformer structure has some similarities to the BRIDGE pattern [GoF].

ates a fragment of the whole SQL statement, combines it with the fragments of its child objects, and returns the combined phrase.

To support transformations into dialects of SQL that differ only slightly from standard SQL, create sub classes of individual transformer classes that perform the required behavior. If, for example, creating a sub query statement for database A is different from the standard case, a class `SQLASubQueryTransformer` inherits from `SQLSubQueryTransformer`. In that case, the AbstractFactory pattern ([GoF]) simplifies the construction of the transformer object tree.

If a query should be automatically optimized before it is executed on a data source, the transformer objects may be manipulated before they create the native query statement. Any kind of modification is possible. You could, for example, analyze a transformer tree and modify each sub query to use joins instead of sub select statements.

Query Transformer objects are created by a Query Transformer Factory, as can be seen in Fig. 8. Such a factory examines a `Query` object and creates and returns an appropriate `QueryTransformer` instance. In addition to a SQL Query Transformer, many other transformers are possible, for example for the Hibernate query language HQL[5] or for LDAP queries.

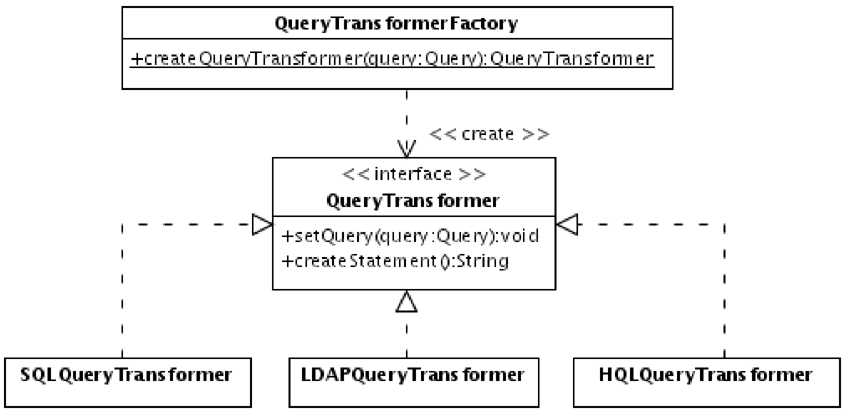


Fig. 8: Query Transformer Factory

Using this approach, the same query could be transformed transparently into different native query languages. The factory chooses the appropriate Query Transformer either by applying heuristics (if the query target is a class name then return the Hibernate transformer, etc.) or by specific settings.

By using Query Transformers, the core of the Query Engine is separated from details of creating native query statements. If new data sources with different query languages or query dialects have to be integrated, new Query Transformer implementations can be added without modifying the core.

Query Performer

If there is only a single data source, the code for executing a native query statement can reasonably be part of the core of the Query Engine. Use distinct Query Performers instead when one or more of the following constraints apply:

- The project involves data sources that provide different APIs for querying.

5 Hibernate is an open source object/relational mapping tool. The variation *Object-Oriented Domain Model* explains the integration of such a tool in-depth.

- Different persistence strategies have to be integrated.
- Remote Result Sets are used (see the last variation).

A Query Performer encapsulates how a native query statement is executed and how a Result Set is built from the query results. Within the Query Engine, Query Performers are defined by an interface called `QueryPerformer`. Fig. 9 shows both the interface and three implementations: `JdbcPerformer` executes queries in SQL using a JDBC driver, `LDAPPerformer` executes LDAP queries, and `HibernatePerformer` executes queries in the Hibernate query language HQL using the API of Hibernate.

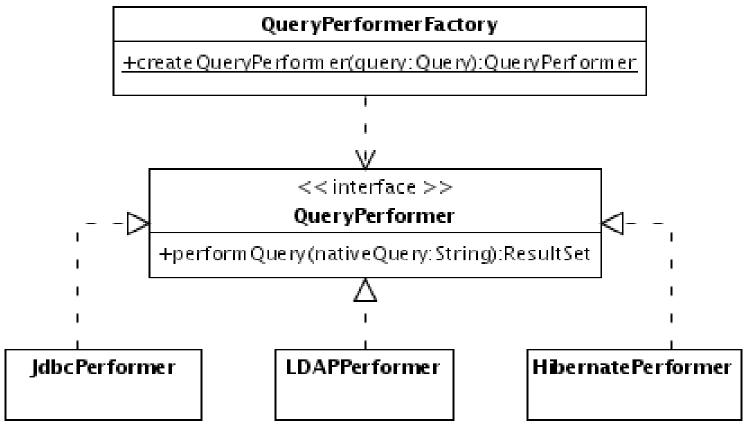


Fig. 9: Interface for a Query Performer and three implementations

Separating the transformation a Query from the execution of the native query statement has advantages: Data sources that use different query language dialects may provide the same API for executing queries. In Java, for example, JDBC drivers encapsulate the access to relational database systems from different vendors. Therefore, a single Query Performer may be sufficient, whereas several Query Transformer implementations may be necessary to create native query statements adapted to each relational database system.

Fig. 10 shows an updated diagram of Fig. 2: the participants in a Query Engine that uses both a Query Transformer and a Query Performer.

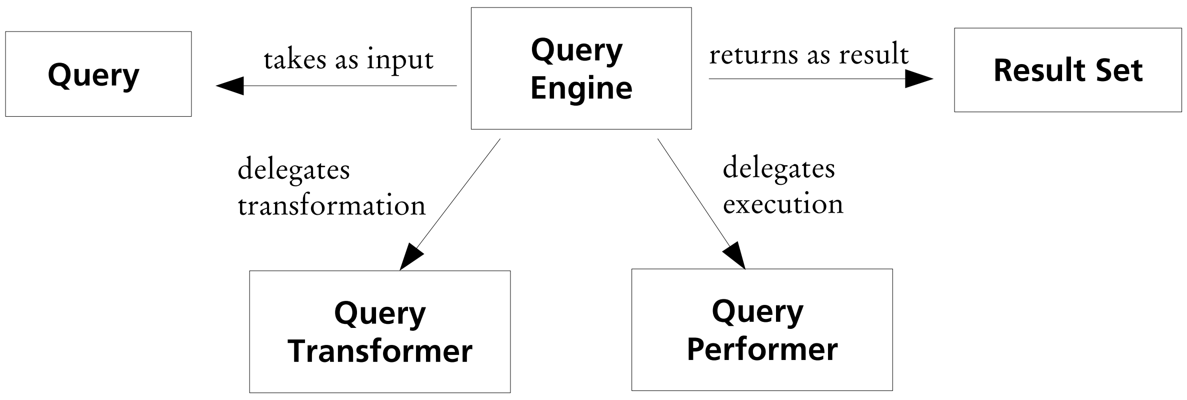


Fig. 10: Query Engine participants including Query Transformer and Query Performer

The implementation of a Query Engine that uses both Query Transformers and Query Performers is simplified:

```
QueryEngineBean.java:

public ResultSet executeQuery(Query query) {

  QueryTransformer queryTransformer =
    QueryTransformerFactory.createQueryTransformer(query);

  String queryStatement = queryTransformer.createStatement();

  QueryPerformer queryPerformer =
    QueryPerformerFactory.getQueryPerformer(query);

  ResultSet resultSet = performer.performQuery(queryStatement);

  return resultSet;
}
```

Object-Oriented Domain Model

Many information systems have a persistence layer that supports an object-oriented domain model. These layers map between business objects and their corresponding database tables and columns and retrieve and store business objects. Fig. 11 shows an updated reference architecture from Fig. that includes a persistence layer.

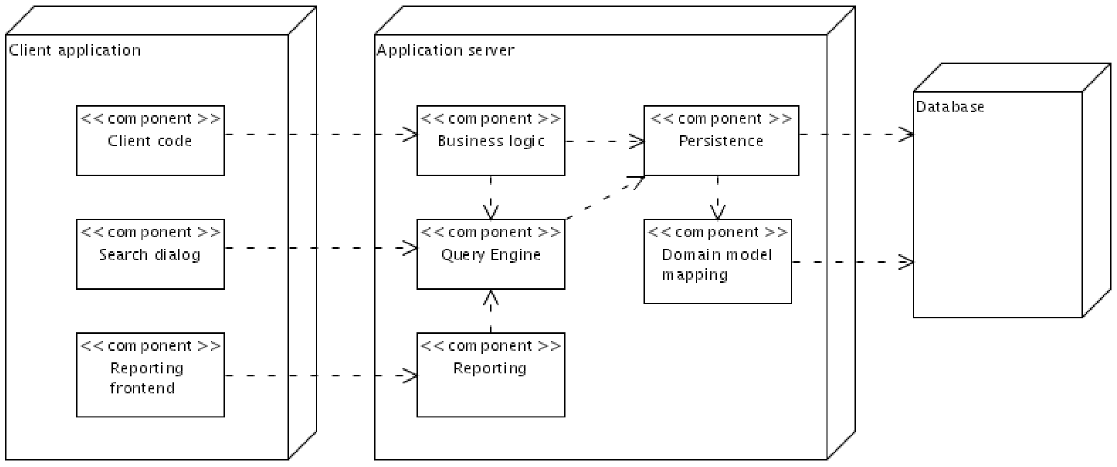

Client application
<<component>> Client code
<<component>> Search dialog
<<component>> Reporting frontend
Application server
<<component>> Business logic
<<component>> Query Engine
<<component>> Reporting
<<component>> Persistence
<<component>> Domain model mapping
Database

Fig. 11: A system architecture that involves object-relational mapping

An application that uses an object-oriented domain model should avoid Queries that refer to database tables and columns. Instead, Queries should refer to business classes and their attributes. The Query implementation introduced earlier in this paper does not make any assumptions about the domain model because all references are given as plain strings. It can therefore be used unmodified.

Fig. 12 shows the class diagram of the business domain model of the CRM application. It corresponds to the relational data model shown in Fig. 3 (getter and setter methods are left out).

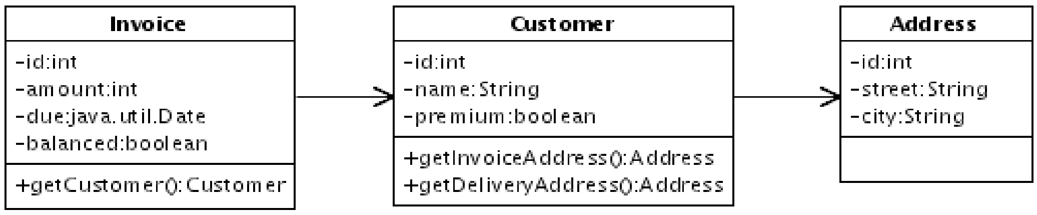


Fig. 12: An object-oriented domain model for the CRM application

For a client, writing a Query based on this domain model is straightforward. The following code creates a query for due invoices, similar to the example given earlier:

```
private Query createQueryForDueInvoices(Date firstDate, Date lastDate) {

  Query query = new Query(Invoice.class.getName());
  query.setMaximumResults(100);

  query.setFilter(new Filter(new String[]{Invoice.class.getName()}));
  query.setSort(new Sort(Invoice.DUE, false));

  CombinedCondition condition = new CombinedCondition(Operator.AND);
  condition.addCondition(
    new FieldCondition(Invoice.BALANCED, Comparator.EQUALS,Boolean.FALSE));
  condition.addCondition(
    new FieldCondition(Invoice.DUE, Comparator.GREATER_EQUALS, firstDate));
  condition.addCondition(
    new FieldCondition(Invoice.DUE, Comparator.LESS_EQUALS, lastDate));

  query.setCondition(condition);

  return query;
}
```

This code differs from the earlier example, which was based on database tables and columns:

- The names of business fields are not hard coded. Instead, class names and constants are used.
- The filter does not contain individual field names because business objects are always loaded completely. Therefore, the name of the target business class is set.
- The database column `balanced` was compared to the integer value `0`. That column is now mapped to the boolean attribute `Invoice.BALANCED`, and it is compared to the value `Boolean.FALSE`.

For a client, executing a Query referring to an object-oriented domain model is the same as executing a Query referring to a relational data model. The Query Engine must have both a specialized Query Transformer and a Query Performer: the Query Transformer creates a statement in the native query language of the persistence layer and the Query Performer executes the native query statement by calling the persistence API.

The open source persistence library Hibernate, for example, could be integrated into a Query Engine by creating a `HQLQueryTransformer` and a `HibernatePerformer`.

Many persistence layers support loading object graphs. This indicates that not only the target object is retrieved but also associated objects. In most cases, though, not all associated objects are needed. A hierarchical filter allows to explicitly define a sub set of the object graph that should be retrieved. If, for example, both the customer and the invoice address are required, the following filter may be set:

```
new Filter(new String[]{"Invoice",
                        "Invoice.customer",
                        "Invoice.customer.invoiceAddress"});
```

The given filter lists all classes and the fields that form the associations between the classes. How such hierarchical filters are implemented depends on the persistence layer.

Only those persistence libraries and components can be integrated into a Query Engine that support dynamically created query statements. Entity Beans that are part of the EJB framework cannot be used: Queries in EJB-QL have to be statically defined in descriptors; only parameters can be set. Therefore, a J2EE environment requires a more flexible object/relational mapping tool.

Within an application, several approaches to persistence may co-exist and be combined. In the CRM application, it may be useful to retrieve a hash map of business data from a joined database view if the data is shown read-only. On the other hand, the same business data may be manipulated in form of business objects if the data is updated in the application.

Remote Result Sets

Depending on the client application, different strategies are necessary for how and when the results from an execution of a Query are transferred. Some applications need a low memory footprint for results, whereas execution speed is a less important concern. Other applications require fast and immediate transfer of all results. A web application does normally not show all results of a search request on a single page, whereas in a rich-client application it may be important to receive all results as soon as possible to show them in a table.

Remote Result Sets are Result Sets that do not return the result objects to the client at once. Instead, they keep references to the resources of the underlying data sources and provide PROXY [POSA] objects.

Fig. 13 contains a class diagram that shows several `RemoteResultSet` implementations and their relationships. The interface `RemoteResultSet` is a marker interface and does not add any additional operations to `ResultSet`.

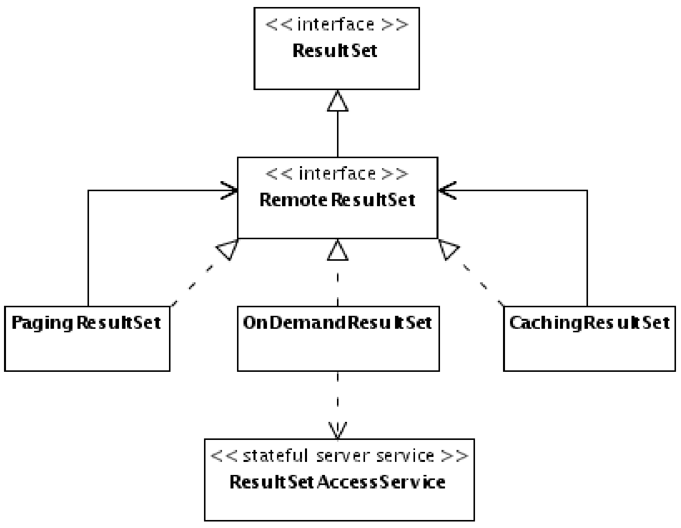


Fig. 13: The interface `RemoteResultSet` and its implementing classes

The class `OnDemandResultSet` forwards each request for result set elements to a server component. The service `ResultSetAccessService` has to be stateful and must be bound to the `OnDemandResultSet` object.

There are two alternatives on how the service may fulfill the requests: it may permanently hold a connection to the native result set object, for example to a `java.sql.ResultSet`. Or it may keep a list of identifiers of those business objects that were retrievable when the native query was executed. The first alternative might cause resource shortages when many search results are kept open. The second alternative yields further database queries whenever the client application calls the Result Set.

Both `CachingResultSet` and `PagingResultSet` are DECORATORS [GoF]. They wrap around other `RemoteResultSet` objects and provide additional functionality. A chain of `RemoteResultSet` objects is possible: it is useful as long as the chain ends with an `OnDemandResultSet` instance that actually fetches objects from the server.

A `CachingResultSet` instance forwards requests for result set elements to the underlying `RemoteResultSet` objects only if the elements have not been requested before and are therefore not in the cache. How such a client-side cache is organized may be defined by further sub classes or by separate caching strategies.

A `PagingResultSet` object always retrieves blocks of result set elements even if only single elements were requested and it always keeps the last block it has fetched. The benefit of this class is improved performance without a memory consuming cache in those cases when a client application typically requests neighboring result set elements. When the size of the cache is not an issue, a `PagingResultSet` instance as decorator for a `CachingResultSet` object might improve performance.

The creation of a `RemoteResultSet` object differs from the creation of a `SimpleResultSet` object. Instead of passing all data objects into the Result Set object, a stateful instance of `ResultSetAccessService` is created and references to the native result set resources are passed to it. It is therefore useful to create individual Query Performers for Remote Result Sets.

The Query Engine determines which Result Sets are used and chained, either by applying heuristics or by evaluating additional search options from the client.

The main benefit of chaining `RemoteResultSet` objects is that specific requirements about fetching data may be fulfilled by only one conforming Remote Result Set. Since the client application only depends on the super interface `ResultSet` it may at first be sufficient to use the `SimpleResultSet` implementation. More complex `RemoteResultSet` implementations may be created and integrated at a later point of time without affecting the client code.

In the Java world, there is a new standard emerging for JDBC result sets that resembles Remote Result Sets: Row Sets[6]. A Row Set is a disconnected `java.sql.ResultSet` whose behavior can be manipulated by selecting strategies. One such strategy allows transferring a Row Set with all results to a client application. Row Sets can easily be integrated into the Query Engine by writing a small wrapper that implements the `ResultSet` interface and by removing code from the Query Engine that creates data transfer objects. Row Sets are not as flexible as Remote Result Sets, though. For example, they do not support on demand loading results from a client, and it is not possible to combine retrieval strategies. Furthermore, they couple the client application at runtime to the classes of the JDBC driver. Row Sets are useful if the client application makes changes to the data and needs to write them back into the database.

6 Row Sets are defined in JSR 114 and part of JDBC 4.0

Acknowledgements

I'd like to thank my EuroPLoP shepherd Allan Kelly who helped a lot to improve the paper. He provided many useful comments, especially regarding the forces of the pattern and the structure of the paper.

I would also like to thank the participants of EuroPLoP 2004 who workshoped the paper. They gave valuable feedback that helped me to put the paper into its final form.

Last but not least, I would like to thank Lutz Hankewitz, Dominik Dunekamp, and Astrid Bauereisen. Lutz in particular gave important feedback on early drafts.

References

[Fowler]	*Patterns of Enterprise Application Architecture*. Martin Fowler. 2002. Addison Wesley Professional.
[GoF]	*Design Patterns. Elements of Reusable Object-Oriented Software*. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. Addison-Wesley.
[Marinescu]	*EJB Design Patterns*. Floyd Marinescu. 2002. John Wiley & Sons.
[POSA]	*Pattern-oriented Software Architecture - A system of patterns*. Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. John Wiley and Sons.