Siou Lin
CSc 630—The Science of Computer Science
Professor Snodgrass
2 May 2007

Query Generation Infrastructure for AZDBLab

1 Introduction

1.1 Background

Automatic query generation has a wide array of advantages. First, it is faster than manual query generation. This advantage is beneficial in data mining, where vast numbers of queries are generated and executed in order to help find correlations within the data. Additionally, automatic query generation can be used to decrease predictability. This property is useful in benchmarking applications, where it is undesirable for a DBMS to cache previously computed results. Furthermore, automatic query generation can be used to keep some parameters constant while varying others. This is useful in the study of flutter because it allows for the automated testing of query variations to help determine which factors contribute to flutter.

1.2 Project Overview

This project focuses on developing the query generation infrastructure for AZDBLab. AZDBLab is a modular infrastructure designed for conducting database research. It interfaces with multiple DBMSes and contains several modules, including modules for experiment specification, data generation, and query generation. The modules in AZDBLab work together to allow a user to specify, run, and view the results of an experiment. The results are stored within AZDBLab to allow for later retrieval.

Three main areas of research for developing the query generation infrastructure were: creating a query generator, creating a query specification GUI, and conducting experiments to determine interesting query variations. These topics are addressed in Sections 2-4 of this paper, respectively. Section 5 gives conclusions and outlines areas for future work.

2 Creating a Query Generator

A query generator consists of three core code components: a query specification language, a query generator, and a query display module. The specification language defines a grammar for the query template. The query generator generates queries from the query template. Figure 2-1 shows the relationship between the specification language, query template, and query generator.

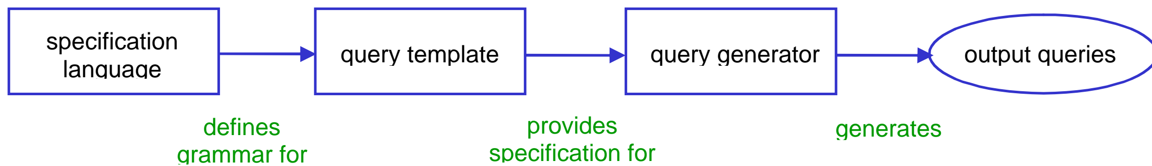

Figure 2-1: Automatic Query Generation

The query display module integrates the query generator with the AZDBLab GUI. When a user clicks on a query in the navigation, the query display module is used to translate the XML query template into an HTML display.

This section starts by giving an overview of how a query generator interacts with AZDBLab. It then continues to explain how to implement each of the three core components and integrate them into AZDBLab. Specifically, these core components are implemented for a permutable query generator. Finally, it explains how to test a new query generator.

2.1 Interaction with AZDBLab

Figure 2-2 shows how the classes in AZDBLab interact to generate, run, and display queries. Because this diagram is meant to give a general overview of the underlying interaction, most class diagrams have been collapsed to only display the class name. First, when AZDBLab is started, the Main class ensures that all query templates loaded in the system generate queries and are ready to run. Next, if a user runs an experiment, the main class handles the event by calling the generateQueries method which communicates with the database to run the queries associated with the experiment.

The InternalDatabaseController contains an OracleSubject object, which extends ExperimentSubject and implements LabNotebook. ExperimentSubject is contains methods to help run experiments. The LabNotebook interface contains methods that store and retrieve experiments from the database. Thus the InternalDatabaseController is able to run the specified experiments and store the results.

ExperimentData models the view content. This class contains references to the query display module (not shown) and other display objects. This class is used by ResultBrowserFrame to determine the content of the main application frame.

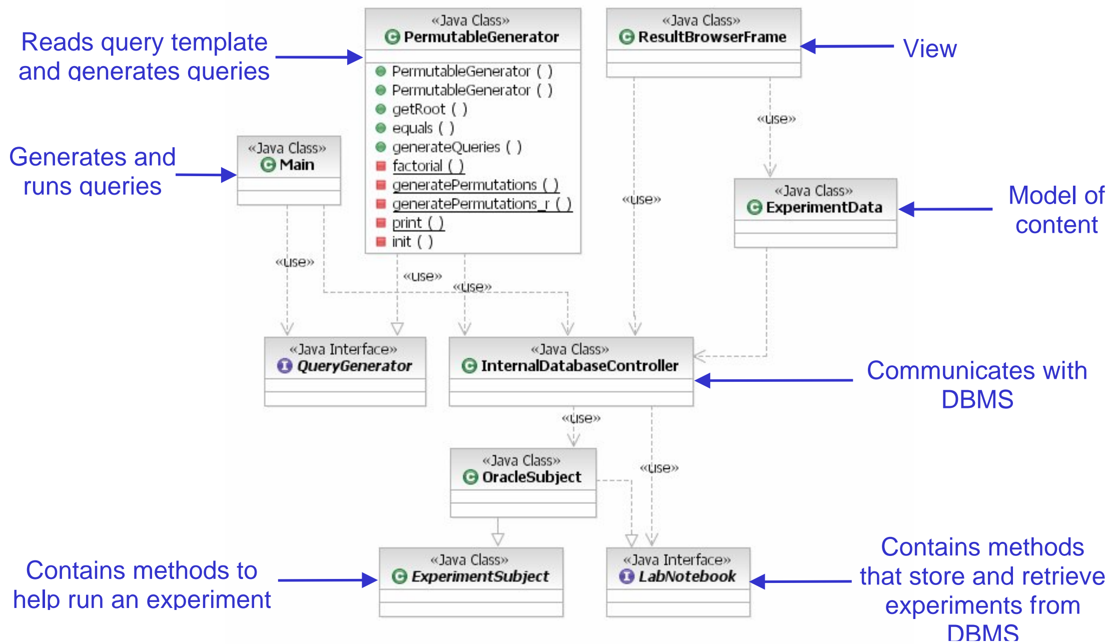

Figure 2-2: Class Interaction Diagram for Query Generator

2.2 Creating a Query Specification Language

The query specification language is written using XML Schema Definition (XSD). Figure 2-3 shows the schema diagram for the permutable query specification language. This schema file is named permutableQueries.xsd and placed in the xml_schema directory. Refer to the source file for details on the implementation.

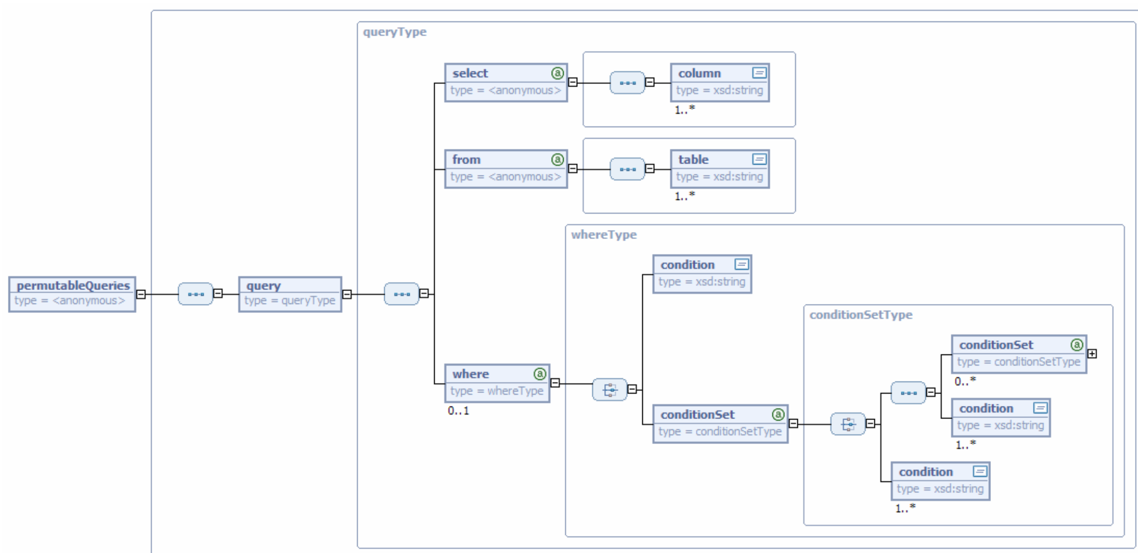

Figure 2-3: XML Schema Diagram for Query Specification Language

To briefly explain the schema diagram, the root element for this schema is permutableQueries. It is composed of one query element of type queryType. The queryType type is composed of exactly one select and one from element. The select element contains one or more column elements. The from element contains one or more table elements.

Additionally, the queryType type contains an optional where element, which is of type whereType. The whereType type is composed of condition and conditionSet elements. Note that the conditionSet element is of type conditionSetType and contains an optional recursive reference to itself. This allows for the specification of hierarchical where statements. Finally, the select, from, and where elements each contain a permutable attribute. If this attribute is set to true for any element, then the corresponding child elements will be permuted. This attribute is set to false by default.

In order to integrate the new schema definition into AZDBLab, the experiment.xsd file must be modified to allow query generators of this new type. In the queryDefinitionReference element, add:

```
<xsd:enumeration value="permutableQueries" />
```

In the queryDefinition element, add:

```
<xsd:element name="permutableQueries" />
```

Next, references to the new query generator must be added to the OSAT.java class in the osat package. Add the following three lines of code:

```
public static final String S_GENERATOR_TYPE_PERMUTABLE
      = "permutableQueries";
public static final String PERMUTABLE_SCHEMA
      = XML_SCHEMA_DIRECTORY + "permutableQueries.xsd";
public static final int GENERATOR_TYPE_PERMUTABLE = 4;
```

The variable S_GENERATOR_TYPE_PERMUTABLE is set to the name of the root element of the query generator schema. The variable PERMUTABLE_SCHEMA is set to the location of the schema file. Finally, the variable GENERATOR_TYPE_PERMUTABLE is set to the value of the enumeration defined in experiment.xsd.

2.3 Creating a Query Generator

The query generator is written in Java. It has three main functions: validating the query template against the corresponding schema, parsing the query template, generating output queries, and running the output queries. Figure 2-4 shows the class diagram for the permutable generator. This class can be found in the osat.analysis.queryGenerator package.

Figure 2-4: Class Diagram for PermutableGenerator

At this point, it would be useful to briefly explain how to write a query generator. First, a query generator must implement the QueryGenerator interface. This interface contains three methods to be implemented: equals, generateQueries, and getRoot.

The equals method is used to determine whether one instance of a query generator is exactly equivalent to another instance. To do this, it traverses the Document Object Model (DOM) tree of the query template for both queries and compares each of the fields. The generateQueries method interacts with the database controller to run the queries generated from the template. It would be more appropriately named runQueries, but the method name is retained to maintain compatibility with the existing QueryGenerator interface. The getRoot method simply returns the root of the DOM tree.

A query generator contains two constructors. One is used if a reference to the query template is passed in, while the other is used if a DOM of the query template is passed in. The purpose of the constructor is to initialize all the global variables and call the init function to parse the DOM tree and store the output queries. The rest of the methods are helper functions that are used to translate the DOM tree into a set of queries.

After the query generator is written, it must be integrated into AZDBLab. To do this, the following lines of code should be added to the constructor of osat.analysis.experiment.Test:

```
case OSAT.GENERATOR_TYPE_PERMUTABLE: {
      if (generatorHasRef()) {
            myQueryGenerator = new PermutableGenerator(
                 getQueryGeneratorHRef(), myDataDef, numberQueries,
                 _controller, user_name, notebook_name,
                 myExperimentName, myTestNumber);
     } else {
      myQueryGenerator = new PermutableGenerator(
            getQueryGeneratorElement(), myDataDef,
            numberQueries, _controller, user_name,
            notebook_name, myExperimentName, myTestNumber);
      }
      break;
}
```

2.4 Creating a Query Display Module

The query display module is written in eXtensible Stylesheet Language (XSL). It is used to translate the query template into an HTML display. This display will be shown in the main frame of AZDBLab when a Query Definition item is selected in the navigation. Figure 2-5 shows the display for a permutable query template.

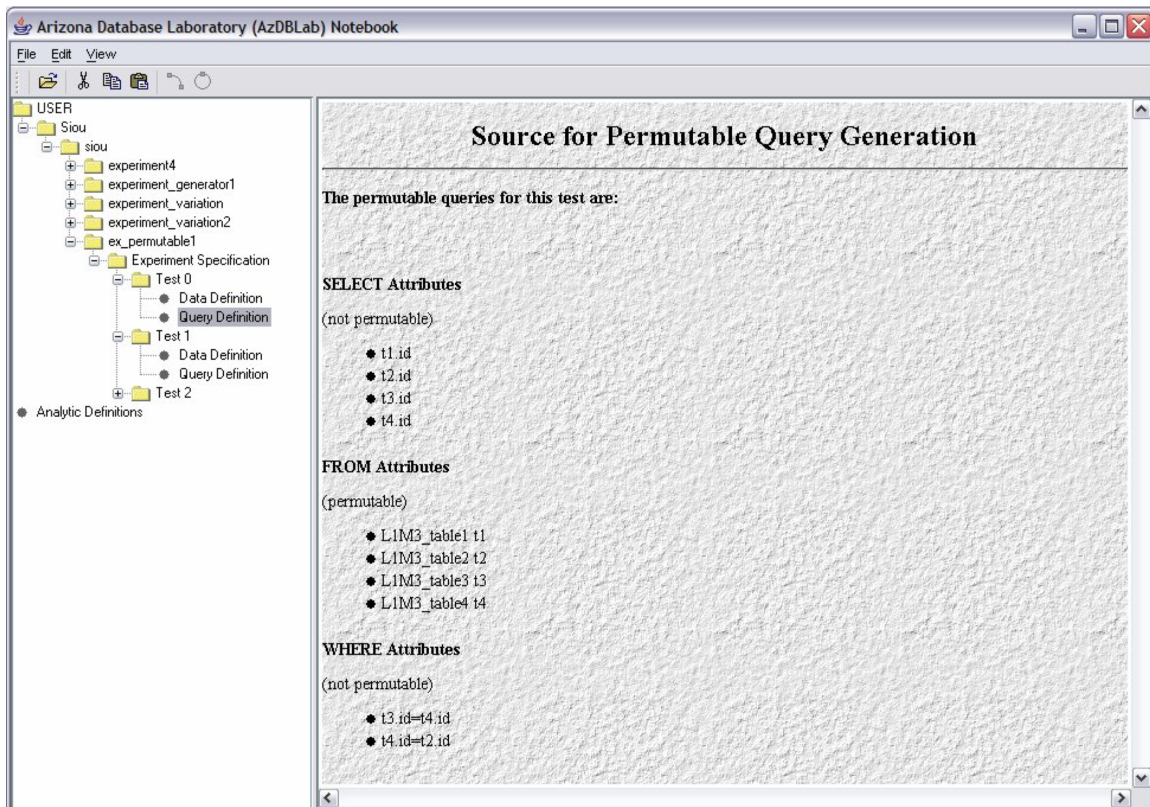

Figure 2-5: Display of Permutable Query Generator

It is not necessary to complete the query display module before testing query generator. If this step is not completed, the screen will be blank when a Query Definition item is selected for a permutable query.

The code fragment below is added to source.xsl in the xml_style directory to help parse the XML query template and output a corresponding HTML display for permutable queries. Note that this code fragment only parses the select element in the xml template. Refer to source.xsl for the entire code.

```
<xsl:template match="/permutableQueries">
<HTML>
      <BODY>
            ...
            <p>
            <h3>SELECT Attributes</h3>
            <xsl:choose>
                  <xsl:when test="query/select/@permutable =
'true'">(permutable)</xsl:when>
                  <xsl:otherwise>(not permutable)</xsl:otherwise>
            </xsl:choose>
            <ul>
            <xsl:for-each select="query/select/column">
                  <li><xsl:value-of select="."/></li>
            </xsl:for-each>
            </ul>
            ...
            </p>
      </BODY>
</HTML>
</xsl:template>
```

2.5 Testing the Query Generator

To test the query generator, it is necessary to create an experiment specification file, a data definition file, and a query template file in the xml_source director. All of these files are specified in XML. Existing experiment and data definition files should be available in the workspace for other query generators. Thus, it is recommended, for testing purposes, to modify some existing files to work with a new query template.

A query template is created in accordance to the query definition schema. Below is an example of a permutable query template.

```
<?xml version="1.0" encoding="UTF-8"?>
<permutableQueries xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:noNamespaceSchemaLocation="../xml_schema/permutableQueries.xsd">
      <query>
            <select permutable="false">
                  <column>t0.id3</column>
                  <column>t0.id2</column>
            </select>
            <from permutable="false">
                  <table>dp10_HT1 t0</table>
                  <table>dp10_HT2 t1</table>
                  <table>dp10_HT2 t2</table>
            </from>
            <where permutable="true">
                  <conditionSet operator="AND">
                        <condition>t0.id4=t1.id2</condition>
                        <condition>t1.id2=t2.id2</condition>
                  </conditionSet>
            </where>
      </query>

</permutableQueries>
```

Note that the where clause is the only one that is specified as being permutable. Since there are two conditions joined by a logical connective, two possible permutations exist for this query. One possible permutation of this query is:

```
SELECT t0.id3, t0.id2
FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2
WHERE (t0.id4=t1.id2 AND t1.id2=t2.id2)
```

To test the permutable query generator, simply start AZDBLab, create a new experiment, and run the experiment.

3 Creating a Query Specification GUI

A query specification GUI is used as an alternative to manually writing queries in XML. The user is able to graphically specify a query and output the resulting query in XML format. Figure 3-1 shows a screenshot of the query specification GUI.

The query specification GUI is implemented according to the Model-View-Controller (MVC) design pattern. There are five components that need to be implemented to create a query specification GUI:

- an observable query model
- an query specification view
- controllers that support user actions
- mediators to control the interaction between the Java model and XML template

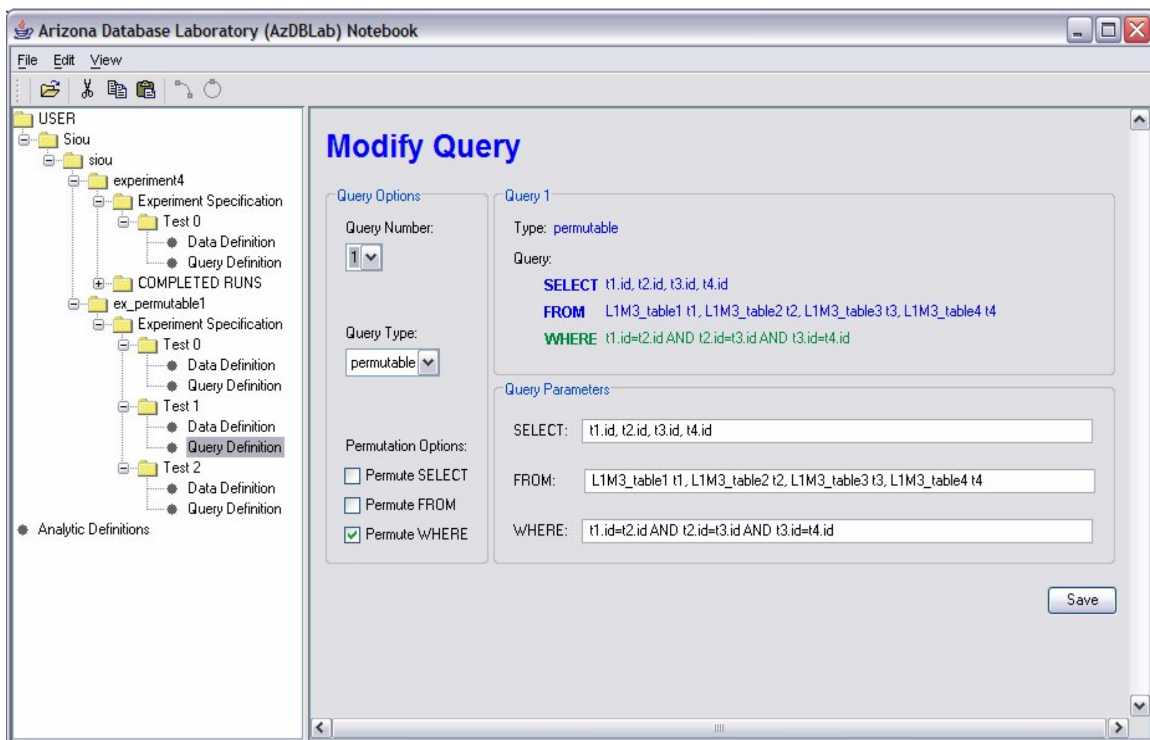

Figure 3-1: Query Specification GUI

This section starts by giving an overview of the MVC design pattern. Then, an overview of how the query specification GUI interacts with AZDBLab is presented. Finally, implementation examples are given for each type of component.

3.1 Overview of MVC

Before delving into implementation details, it would be beneficial to provide some background on the MVC design pattern. The main purpose of the MVC pattern is to decouple the interactions between the model, view, and controller. This allows a developer the flexibility to alter each standalone component with minimal impact to the other components. This design pattern also encourages code reuse. Figure 3-2 illustrates how MVC operates.

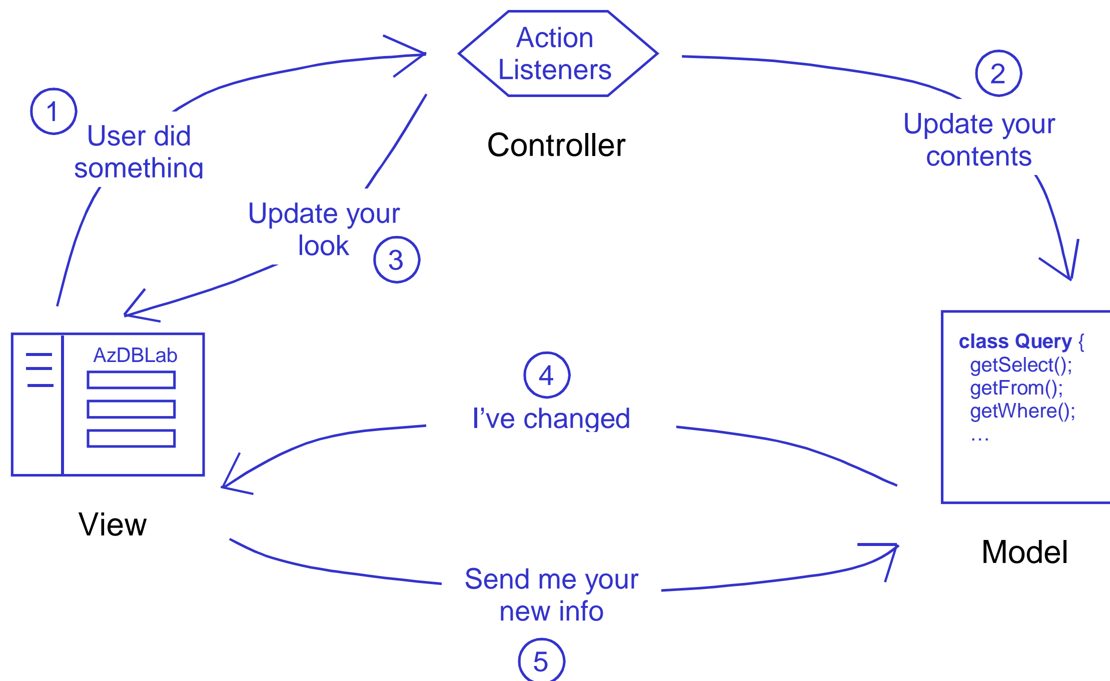

Figure 3-2: Model-View-Controller Design Pattern

This diagram is based on the MVC diagram from p. 530 of Head First Design Patterns as cited in the reference section. First, the view notifies the controller when a user performs some action. The controller then tells the model to update its contents and the view to update its look. Meanwhile, the model notifies the view that it has new information. Finally, the view asks the model to send over the new information so that it can update its own content.

3.2 Interaction with AZDBLab

Figure 3-3 shows how each of the individual components work together to form a query specification GUI. ResultBrowserFrame is the main view for AZDBLab. ExperimentData models the content of the main frame inside the view of AZDBLab. The QueryModifyPanel is displayed when the user selects to modify a query in the navigation. DefaultQuery is an observable model for a query object. It is observed by the view and notifies the view whenever its content is updated. The controllers communicate with the view and tell the model when to update its contents in response to user actions. When the user requests to save a query template, a controller calls the DefaultQueryWriter proxy to write the contents of the DefaultQuery model to a specific XML file.

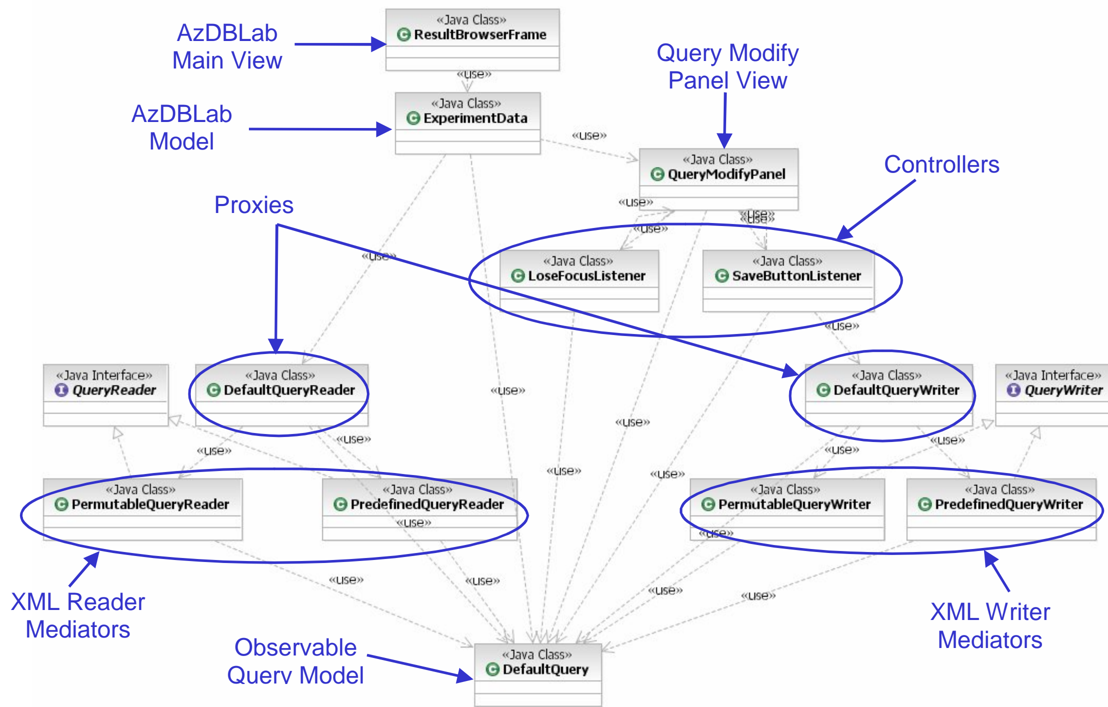

Figure 3-3: Class Interaction Diagram for Query Specification GUI

3.3 Creating an Observable Query Model

The observable query model is used store an updatable version of the query template. It updates its observers whenever its contents are updated. Figure 3-4 shows the class diagram for the DefaultQuery class. Refer to osat.analysis.query.model.DefaultQuery for implementation details.

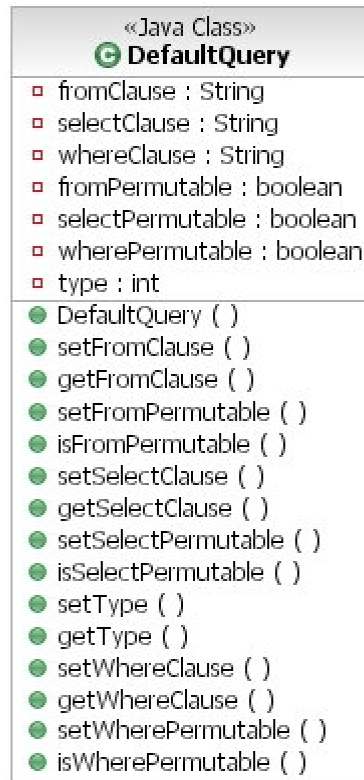

Figure 3-4: Class Diagram for DefaultQuery

3.4 Creating a Query Specification View and Controllers

The view defines look for the GUI. Additionally, it contains the controllers that are in charge of updating the model. Figure 3-5 shows the class diagram for the QueryModifyPanel class. Refer to osat.browser.small.view.QueryModifyPanel for implementation details.

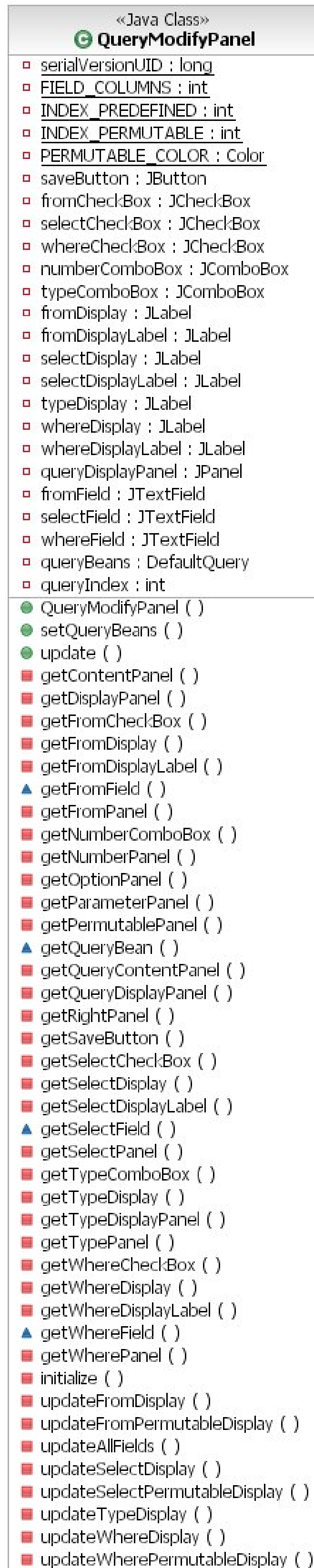

Figure 3-5: Class Diagram for QueryModifyPanel

3.5 Creating a Mediator

The mediator helps coordinate the interaction between an XML template and a Java model. Figure 3-6 shows the class diagram for the PermutableQueryWriter class. Refer to osat.browser.small.view.PermutableQueryWriter for implementation details.

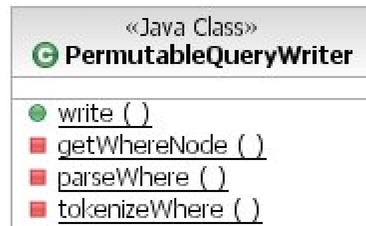

Figure 3-6: Class Diagram for PermutableQueryWriter

4 Determining Interesting Query Variations

4.1 Background

One reason automatic query generation is valuable to the study of flutter in that it is able to quickly and systematically produce variations of an interesting query. Thus, the first question to ask when developing a query generator is: which query variations are interesting in the study of the flutter phenomenon? To help answer this question, it is imperative to first define flutter.

So what is flutter? Generically, it is a phenomenon related to query optimization. Thus, it would be useful to present a brief background on query optimization. A query optimizer works by estimating the cost of some subset of equivalent query plans and determining a good query plan based on these cost estimates. It follows that two core questions are: which subset of equivalent query plans is considered, and how are query plan costs estimated? Each individual query optimizer implementation answers these questions differently, but some general trends are described below.

Since it may be computationally infeasible to consider the set of all possible equivalent query plans, some subset of plans must be selected. Modern optimizers often favor left-deep plans because they may be fully pipelined. However, for complex queries, it may be computationally infeasible to evaluate even the entire subset of left-deep plans. In this case, heuristics or randomized plan generation may be used to select a subset of plans to evaluate (Ramakrishnan 507).

Another issue affecting how query optimizers work is how query plan costs are estimated. Information stored in a system catalog is used to estimate the cost of each query plan. For example, the sort order and the size of the input tables may be used in cost estimation (Ramakrishnan 416). For each node in a relational algebra tree, cost estimation consists of two parts: estimating the cost of performing the corresponding operation, and estimating the size of the result (Ramakrishnan 482).

So, what is the relationship between flutter and query optimization? As mentioned, input table cardinality can be a major contributing factor to query optimization. When the query optimizer is lied to about this factor, it may output some non-optimal query plan estimates. Flutter is an oscillation in query plan number corresponding to reported input table cardinality. Figure 4-1 shows an example of flutter where Query Plan 0 is output by the query generator for two discontinuous reported cardinality ranges.

The values in the table below show how the DBMS performed with incorrect cardinality statistics.

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	1	0 ms
HT1	175000	0	0 ms
HT1	107000	2	0 ms
HT1	104000	0	0 ms
HT1	64000	3	0 ms

oscillation in query plan

Figure 4-1: Example of Flutter

4.2 Experiment Overview

Having defined the nature of the phenomenon, it is then possible to design some experiments testing it. The objective of the experiment conducted for this paper is to run variations of a fluttered query to determine if the variations (1) generate flutter and (2) display the same type of flutter as the original query. The original fluttered query used in the experiment is shown below.

```
SELECT t0.id3, t0.id2
FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2
WHERE (t0.id4=t1.id2 AND t1.id2=t2.id2)
```

This query is subject to the following constraints:

- The actual cardinality of HT1 is 100000, but the reported cardinality from 1 to 1250000.
- The actual cardinality of HT2 is 100000, and the reported cardinality is the same.
- Table entries are randomly generated numbers from 1 to 1000000.

Table 4-1 lists the six different query variations that were examined in the experiment.

Variation	Query
Permute attribute order	SELECT t0.id2, t0.id3 FROM dp10_HT2 t1, dp10_HT1 t0, dp10_HT2 t2 WHERE (t1.id2=t2.id2 AND t0.id4=t1.id2)
Substitute logical connective	SELECT t0.id3, t0.id2 FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2 WHERE (t0.id4=t1.id2 OR t1.id2=t2.id2)
Substitute comparison operator	SELECT t0.id3, t0.id2 FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2 WHERE (t0.id4>t1.id2 AND t1.id2<t2.id2)
Drop condition	SELECT t0.id3, t0.id2 FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2 WHERE t1.id2=t2.id2
Append condition	SELECT t0.id3, t0.id2 FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2 WHERE (t0.id4=t1.id2 AND t1.id2=t2.id2) OR t2.id2=t0.id4
Negate condition	SELECT t0.id3, t0.id2 FROM dp10_HT1 t0, dp10_HT2 t1, dp10_HT2 t2 WHERE (t0.id4=t1.id2 AND NOT(t1.id2=t2.id2))

Table 4-1: Variations of Fluttered Query

4.3 Hypotheses and Predictions

Based on intuition and a basic understanding of how query optimizers work, three falsifiable hypotheses were proposed:

1) Equivalent queries will exhibit the same flutter.
2) Substituting operators into a fluttered query will produce a fluttered query.
3) Appending a condition to a fluttered query will produce a fluttered query.

These three hypotheses are falsifiable because they can each be objectively disproved. Hypothesis (1) is based on the fact that a query generator examines a subset of equivalent queries in order to determine an output query plan. Furthermore, table cardinalities play a large role in determining cost estimates. Using these two facts, it is predicted that the same subset of equivalent queries, as based on table cardinality, will be considered no matter which permutation of a query is presented.

Hypotheses (2) and (3) are based on intuition that the complexity of a query positively correlates to the presence of flutter. Considering all logical connectives to be of equal complexity, and all comparison operators to be of equal complexity, hypothesis (2) follows. Similarly, because appending a condition to a query increases the complexity of that query, hypothesis (3) follows.

Based on these hypotheses, predictions were made on each query variation concerning whether they variation would (1) generate flutter and (2) display the same type of flutter as the original query. Table 4-2 lists the predictions made.

Variation	Flutter Exists	Same Flutter Exists
Permute attribute order	yes	yes
Substitute logical connective	yes	no
Substitute comparison operator	yes	no
Drop condition	no	no
Append condition	yes	no
Negate condition	yes	no

Table 4-2: Predictions Based on Hypotheses

Permuting the attribute order of the query should not affect the type of flutter exhibited according to hypothesis (1). Because permutation of attribute order is the only variation that results in an equivalent query, it should also be the only query that is guaranteed to produce the exact same type of flutter. According to hypothesis (2), substituting logical connectives or comparison operations into a fluttered query should produce a fluttered query. Since negating a condition seems to add complexity on the order of appending a condition, hypothesis (3) predicts that these variations will also produce a fluttered query. Dropping a condition from a query is not covered by any of the hypotheses. However, in keeping with the intuition that the complexity of a query positively correlates to the presence of flutter, dropping a condition is predicted to result in elimination of flutter.

4.4 Results and Conclusions

Tables 4-3 through 4-9 show the results of running each query variation.

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	1	0 ms
HT1	175000	0	0 ms
HT1	107000	2	0 ms
HT1	104000	0	0 ms
HT1	64000	3	0 ms

Table 4-3: Result of Original Fluttered Query

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	1	0 ms
HT1	175000	0	0 ms
HT1	107000	2	0 ms
HT1	104000	0	0 ms
HT1	64000	3	0 ms

Table 4-4: Result of Permuting Attributes

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	1	0 ms
HT1	108000	0	0 ms

Table 4-5: Result of Substituting Logical Connective

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	0	0 ms

Table 4-6: Result of Substituting Comparison Operator

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	0	0 ms
HT1	11000	1	0 ms

Table 4-7: Result of Dropping Condition

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	0	0 ms
HT1	96000	1	0 ms
HT1	64000	2	0 ms

Table 4-8: Result of Appending Condition

Table Name	Cardinality of Change Point	Query Plan Number	Execution Time of Plan
HT1	1250000	1	0 ms
HT1	1011000	0	0 ms

Table 4-9: Result of Negating Condition

From these results, it is apparent that several of the original predictions made were incorrect. Table 4-10 corrects the original predictions to be consistent with the experimental results.

Variation	Flutter Exists	Same Flutter Exists
Permute attribute order	yes	yes
Substitute logical connective	~~yes~~ no	no
Substitute comparison operator	~~yes~~ no	no
Drop condition	no	no
Append condition	~~yes~~ no	no
Negate condition	~~yes~~ no	no

Table 4-10: Corrections to Original Predictions

After obtaining the results to these predictions, it is possible to evaluate whether the three hypotheses proposed were falsified. The first hypothesis states: equivalent queries will exhibit the same flutter. This hypothesis has not been falsified because the query permuting the attribute order (the only equivalent query) showed the exact same flutter as

the original query. Additional independent experiments using the permutable query generator on more complex queries support this claim. In all the experiments run, query plans remained consistent across permutations.

The second hypothesis states: substituting operators into a fluttered query will produce a fluttered query. This hypothesis has clearly been falsified both in the case substituting logical connectives and in the case of substituting comparison operators. Recall that for each node in a relational algebra tree, cost estimation consists of both estimating the cost of performing the corresponding operation, and estimating the size of the result. Intuitively, it seems that in the absence of an index, the cost of all comparison operators would be the same. However, it is feasible that different comparison operators could produce variations in the size of the result. For example, it would be reasonable to estimate that the equals operator would produce a much smaller resultant size than an inequality operator.

The third hypothesis states: appending a condition to a fluttered query will produce a fluttered query. This hypothesis has also clearly been falsified. When an additional condition was appended to a fluttered query, the resulting query plans did not exhibit any flutter. Thus, both hypotheses which were based on the intuition of a strong correlation between query complexity and flutter were falsified. However, there did seem to be some evidence supporting this intuition. The results show that the query plan resulting appending a condition contained more change points than the query plan resulting from dropping a condition.

The fact that flutter did not occur in most of the query variations explored suggests that some very specific conditions must be met in order to observe this phenomenon. Based on the results of this experiment, a list of interesting variations will be discussed in the future works section.

5 Conclusion and Future Work

This paper shows how to add a query generator and a query specification GUI to AZDBLab. Furthermore, it documents an experiment conducted to search for interesting query variations. The scope of this project was to aid in developing the infrastructure for automatic query generation in AZDBLab. In summary, fifteen new files were created to expand the query generation and query specification infrastructure. A very rough estimate of the new lines of code (including white space and comments) is around 6000. Several existing classes also had to be modified to accommodate the new functionality. Future work should focus on improving this infrastructure and create more robust query generators and specification GUIs.

5.1 Future Work on Query Variation Experimentation

It would be desirable to conduct more experiments on query variations. The first hypothesis that equivalent queries would exhibit the same flutter withstood falsification. Thus, it would be desirable to subject this hypothesis to more rigorous testing. Specifically, experiments were only conducted on simple queries. It would be interesting to determine if this hypothesis holds for more complex queries as well. The permutable query generator could help in examining this correlation. However, the generator

currently does not handle where clauses that contain a hierarchy of conditions, even though the grammar allows for it. Thus, more logic would need to be added to the permutable query generator.

Also, this experiment only considered logically equivalent queries obtained by permuting attribute orders. It would be interesting to determine if the hypothesis held for other classes of equivalent queries. For example, equivalent queries can be generated by using DeMorgan's Law. This would add several more negation operators to a query. Based on the experimental results, a query with a negated condition did not produce the same flutter as the original query. However, the query considered was not logically equivalent to the original query. Queries generated using DeMorgan's Law would be logically equivalent to the original query.

5.2 Future Work on Query Generators

In addition to finding more interesting query variations, future work should focus on improving query generation. When new, interesting variations are found, the query language should be extended to allow for specification of these new variations. After developing an entirely new language for the permutable query generator, it was realized that a large amount of similarities existed between the permutable query generator and the predefined query generator. In fact, the language of the predefined query generator could have been extended to allow for specification of permutable queries.

Because creating an entirely new query generator can involve rewriting a lot of existing functionality, it is recommended that languages be extended, rather than created, when possible. For example, the language for specifying permutable queries could be extended to allow the specification of a larger set of equivalent queries.

Regarding the infrastructure of the existing query generators, it would be desirable to separate the task of parsing of the XML template to a different class. For this project, mediators where created to read and write between the XML templates and Java query models. Thus, it would be desirable to remove the duplicate functionality inside the query generation module.

5.3 Future Work on the Query Specification GUI

Finally, future work should focus on improving the query specification GUI. Currently, the GUI only supports specification of predefined and permutable queries. The query specification GUI needs to be extended to include specification of random queries. Work also needs to be done to integrate the GUI with the AZDBLab navigation. Furthermore, because it is desirable for AZDBLab to eventually become a standalone system for query specification, experiment and test data specification GUIs also need to be developed. In conclusion, this projected helped develop the infrastructure for automatic query generation and specify some future objectives.

References

[1] M, Poess and J. M. Stephens, Jr., “Generating Thousand Benchmark Queries in Seconds” in *Proceedings of the 30th VLDB Conference*, 2004.

[2] N. Bruno, S. Chaudhuri, and D. Thomas, “Generating Queries with Cardinality Constraints for DBMS Testing” in *IEEE Transactions on Knowledge and Data Engineering*, vol. 18, no. 12, 2006.

[3] R. Ramakrishnan and J. Gehrke, **Database Management Systems**, McGraw Hill, 2003.

[4] Eric Freeman and Elisabeth Freeman, **Head First Design Patterns**, O’Reilly, 2004.