

Science of Computer Science Project Report

AZDBLab Integration- An Experiment in Infrastructure

Preetha Chatterjee
preetha@cs.arizona.edu

Lopamudra Sarangi
lopa@cs.arizona.edu

1. Introduction

Design science is a research methodology, which aims to enhance the knowledge and understanding of a problem domain and its solution through the building and application of a design artifact. It also analyzes the behavior and properties of the software application using the artifact. In this paper we focus on experimentation in query optimization as our problem domain. We enhance an existing design artifact AZDBLabs by integrating two new DBMSes, to study the behavior of database management systems with respect to query optimization.

2. Previous Work

The Science of Database Systems project group under Dr.Richard Snodgrass designed the AZDBLabs artifact, to study the relationship between cardinality estimate accuracy and query plan performance. It is a generic Java based interface used to carry out experiments on several open-source and commercial DBMSes. The current AZDBLabs platform has support for three popular DBMS: Oracle, MySQL and Microsoft SQL Server. In our Science of Computer Science project we integrated two new DBMSes, PostgreSQL and IBM DB2 to the AZDBLabs infrastructure. This paper will concentrate on the detailed steps followed in this integration process.

3. Discussion of Work

We take each DBMS in turn and describe the procedure we followed in detail.

3.1 Adding PostgreSQL as a new Experimental DBMS

The AZDBLabs architecture includes an abstract class called `ExperimentSubject`. To add a new DBMS, the `ExperimentSubject` class is extended and all its abstract methods are implemented.

3.1.1 ExperimentSubject methods

In order to add PostgreSQL to AZDBLabs as an experimental DBMS, a new `PgsqlSubject` class should be implemented by extending the `ExperimentSubject` class. The following is the list of inherited methods that need to be implemented in this class.

public PlanNode getQueryPlan(String sql)

The objective of this method is to get the query plan without executing the query. In PostgreSQL, the EXPLAIN clause displays the execution plan that the PostgreSQL query optimizer generates for the supplied query. The syntax of this statement is

```
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement
```

The execution plan shows how the table(s) referenced by the statement are scanned, i.e. by sequential scan or index scan and if multiple tables are referenced, what are the join algorithms used [1].

Parameters

ANALYZE: The ANALYZE option causes the statement to be actually executed. The output includes the total elapsed time expended within each plan node (in milliseconds) and the total number of rows actually returned in the query result.

VERBOSE: The VERBOSE option shows the full internal representation of the plan tree.

Statement: Any SELECT, INSERT, UPDATE, DELETE, EXECUTE, or DECLARE statement.

The explain statement returns information as a set of rows, that form a hierarchical tree representing the steps taken by the PostgreSQL query processor as it executes a query. The numbers that are quoted by EXPLAIN are:

- Estimated total cost
- Estimated number of rows output by this plan node
- Estimated average width (in bytes) of rows output by this plan node

The following example is taken to illustrate the use of EXPLAIN in PgsqlSubject:

```
EXPLAIN SELECT t0.id4 FROM dp10_HT3 t2, dp10_HT1 t1,  
dp10_HT1 t3, dp10_HT1 t0 where (t2.id1 = t1.id2 AND t1.id2  
= t3.id1 AND t3.id1 = t0.id1);
```

```
QUERY PLAN  
-----  
---- Merge Join (cost=453.06..15823.72 rows=882360 width=4)  
Merge Cond: ("outer".id1 = "inner".id2)  
-> Merge Join (cost=339.80..2204.39 rows=108265 width=16)  
Merge Cond: ("outer".id1 = "inner".id1)  
-> Merge Join (cost=226.53..433.94 rows=13284 width=12)  
Merge Cond: ("outer".id1 = "inner".id1)  
-> Sort (cost=113.27..117.34 rows=1630 width=4)  
Sort Key: t2.id1  
-> Seq Scan on dp10_ht3 t2 (cost=0.00..26.30  
rows=1630 width=4)  
-> Sort (cost=113.27..117.34 rows=1630 width=8)  
Sort Key: t0.id1  
-> Seq Scan on dp10_ht1 t0 (cost=0.00..26.30  
rows=1630 width=8)  
-> Sort (cost=113.27..117.34 rows=1630 width=4)  
Sort Key: t3.id1  
-> Seq Scan on dp10_ht1 t3 (cost=0.00..26.30 rows=1630
```

```

width=4)
-> Sort (cost=113.27..117.34 rows=1630 width=4)
    Sort Key: t1.id2
    -> Seq Scan on dp10_ht1 t1 (cost=0.00..26.30 rows=1630
width=4)
(18 rows)

```

The query plan result set is analyzed and parsed to get the operator and table related properties needed to build the plan tree. A search for the string `Scan` or `Join` is done in each row of the result set to extract the values for a single table operation or a multiple table operation respectively. In the above example, `cost`, `rows` and `width` are inserted for a `Merge Join` (first row of result set) with `TYPE` set to `OPERATOR` in the `PLAN_TABLE`. Similarly for a `Seq Scan` (ninth row of result set) the same fields are extracted along with the table name and inserted into `PLAN_TABLE` with the `TYPE` set to `TABLE`.

In addition to parsing the `ResultSet` and identifying the operator and table nodes, the tree structure should also be constructed for this query plan. The `createPlanTree()` method of the `PlanNode` class is overridden to provide implementation for this. It takes the result set containing all rows of the `PLAN_TABLE` as argument and creates table and operator nodes based on this information. Depending on the `TYPE` field, either an instance of a `TableNode` or an `OperatorNode` is created and then the `buildTree()` method is called to build the plan tree. The `TableNode` and `OperatorNode` are subclasses of the `PlanNode` class. The `buildTree` method() creates the tree by inserting these nodes in a hierarchical manner based on their node ids. Nodes in the plan tree are ordered by node ID such that an in-order traversal of the tree yields the nodes in ascending order.

```

public abstract QueryStat timeQuery(String sqlQuery, PlanNode plan,
long cardinality)

```

This method retrieves the execution time for the indicated SQL query. Before executing the given SQL query, it needs to get and record the system current time as the start time. Then, it executes the query and records the system time as the end time. The difference between the start time and the end time is the query execution time. To eliminate the potential system influence that may introduce delay in execution of the query, the query is executed several times using this method. Then by comparing each query execution time, the minimal execution time is chosen as the required SQL query time.

```

public abstract void setVirtualTableCardinality(String tableName,
long cardinality, RepeatablRandom repRand)

```

There are two ways to set the cardinality of a table depending on the privileges granted to the DBMS user. In order to maintain consistency among the different system level statistics, information stored in system tables is not directly modified to change table cardinality. Instead, tuples are manually added to or deleted from the tables. Initially tuples are inserted into the table `tablename` to reach the maximum hypothetical table cardinality defined in the experiment XML file. After getting the query plan, the necessary number of tuples are deleted from the table to change its cardinality. Thus, the query plan is obtained for different table cardinalities from the maximum to zero. This is useful in setting change points, which act as a control parameter in flutter experiments.

Parameters:

`repRand`: This parameter is used only when the table cardinality is to be changed by inserting tuples. Since in this case, the table cardinality is altered by deleting tuples this parameter is not used currently in the implementation of `PgsqlSubject`.

`tableName`: The table whose cardinality should be set

`cardinality`: Desired cardinality value

public abstract String[] getPlanProperties()

This method retrieves the returned properties from all query plans in PostgreSQL. In this method the columns created in the `PLAN_TABLE` are returned to store the query plan properties. The most prominent properties of a query plan retrieved by the `EXPLAIN` statement in PostgreSQL are `cost`, `rows`, `width`, `operation_name`, `object_type` and `object_name`.

public abstract String[] getPlanOperators()

This method retrieves all the possible operator names returned in the query plan in PostgreSQL. From the results of the current experiments, the operator names found in PostgreSQL were, `Seq Scan`, `Hash Join`, `Index Join` and `Merge Join`. This list does not cover all operator names defined in PostgreSQL. In the future, more experiments can be run to discover more operator names.

public abstract void installExperimentTables(DataDefinitionModule myDataDef, String myPrefix, String[] myTables)

This method creates all tables with the name defined in the parameter `myTables`. Each table column name, data type, primary key and foreign key information can be obtained from `myTables` and `myDataDef` parameter. By calling `createTable()` with this table information, an SQL command can be executed to create the table in PostgreSQL.

Parameters:

`MyDataDef`: This parameter defines the tables and referential integrity of the experiment tables. Below some implementation details about this parameter are listed.

- To get each column name, `myDataDef.getTableColumns(myTables[i])` is used. This returns a string array which includes all column names of table `i`.
- To get the data type for the `jth` column of table `i`, `myDataDef.getColumnDataType(myTables[i], columns[j])` is used, where `columns[]` is a string array containing all the column names of table `i`.
- To get the length of the data type for the `jth` column of table `i`, `myDataDef.getColumnDataLength(myTables[i], columns[j])` is used.
- To get the primary key constraints of table `i` `myDataDef.getTablePrimaryKey(myTables[i])` is used. This returns a string array containing all the primary keys of table `i`.

- To get the foreign key constraints of table `i` `myDataDef.getTableForeignKeys(myTables[i])` is used. This returns a string array containing all the foreign keys of table `i`.

`myPrefix`: string which defines the prefix for each table to be installed.

`myTables`: string array which defines all the table names to be installed.

By concatenating `myPrefix` and `myTables[i]`, the complete name for table `i` is obtained as a string. These are the tables which are created using the `create table` SQL command.

3.1.2 GeneralDBMS methods

`GeneralDBMS` is a super class of `PgsqlSubject`. It defines four abstract methods, which need to be implemented.

```
public abstract String getDBMSDriverClassName()
```

This method retrieves the specific DBMS's JDBC driver class name for AZDBLab to register the driver for this DBMS. In PostgreSQL, it returns the JDBC driver class name as a string:

```
"org.postgresql.Driver"
```

```
public String getDBVersion()
```

This method retrieves the specific DBMS detailed version information and returns it as a string. In PostgreSQL, the `SELECT version();` command is used to get the version information of the installed DBMS. An example command and the result obtained is shown below

```
test=# select version();
                version
-----
 PostgreSQL 8.1.8 on i686-redhat-linux-gnu, compiled by GCC gcc (GCC) 4.1.1
20070105 (Red Hat 4.1.1-51)
(1 row)
```

```
public boolean tableExists(String table)
```

This method checks whether the table exists in PostgreSQL. If the table has already been created in the DBMS, it returns `TRUE`. Else, it returns `FALSE`.

Parameter:

`table`: string to represent the table name.

```
protected String getDataTypeAsString(int dataType, int length)
```

This method produces the data type representation for the specific DBMS given the data type supported by AZDBLabs. Currently, AZDBLabs supports three types of data, they are: number (integer), variable length character (varchar) and clob. Thus, this method returns the corresponding data type string used in PostgreSQL for these three data types. These are defined as: `INT`, `VARCHAR` and `BIGINT`.

Parameter:

DataType: Integer representing a specific AZDBLabs data type.

length: It is a valid integer only when dataType parameter represents variable length character data type.

String constant

In GeneralDBMS, a constant string representing the newly added PostgreSQL DBMS is defined as follows.

```
public static final String DBMS_TYPE_POSTGRESQL = "postgresql";
```

The getDBMSID() method is also modified to support PostgreSQL. This method returns the DBMS ID for the given DBMS name. The codes added for PostgreSQL are shown below.

```
        if(strdbms.equals("postgresql"))    return 3;
```

The current count of DBMS supported by AZDBLabs is incremented in order to ascertain the id for the new DBMS. AZDBLabs includes implementations for Oracle, MySql and SQLServer DBMSes with ids starting from 0. Hence the PostgreSQL system is given the id 3.

3.1.3 ExperimentModule constructor

In ExperimentModule class, the constructor checks to see if the system supports the specific DBMS. So the ExperimentModule constructor should be modified to let it support PostgreSQL as a valid DBMS name. The modification in this class constructor is shown:

```
        if (myDBMS.equals(GeneralDBMS.DBMS_NAME_POSTGRESQL)) {  
            _expsubject = new PgsqlSubject(myUsername,  
                myPassword, myConnectionString);}
```

3.1.4 Schema configuration file

In order to check whether the new DBMS experiment file passes the schema validation test, the schema configurations in experiment.xsd is modified. To add PostgreSQL, the value should be added in the format:

```
<xsd:enumeration value='POSTGRESQL' />
```

3.1.5 lib file

The jar files added to the lib directory for running PostgreSQL is:
postgresql-8.2-504.jdbc3.jar

3.2 Adding DB2 as a new Experimental DBMS

3.2.1 ExperimentSubject methods

The new experimental DBMS DB2 is added by implementing the `DB2Subject` class, which extends the `ExperimentSubject` class. The list of inherited methods implemented in `DB2Subject` is given below.

```
public PlanNode getQueryPlan(String sql)
```

This method retrieves the query plan without executing the query. In DB2 to create explain snapshots, the following explain tables should exist for the user ID: [2]

- `EXPLAIN_INSTANCE`
- `EXPLAIN_STATEMENT`

The DB2 `list tables` command is used to check if these tables exist. If they do not, they must be created using the following instructions:

- If DB2 has not already been started, the `db2start` command is issued.
- From the DB2 CLP prompt, connection is established to the database. For e.g: to connect to the `RESEARCH` database, `connect to research` command is issued.
- The explain tables are created, using the sample command file provided in the `EXPLAIN.DDL` file. This file is located in the `sqlib/misc` directory under `db2`. To run the command file, the `db2 -tf EXPLAIN.DDL` command is issued from this directory. This command file creates explain tables that are prefixed with the connected user ID. This user ID must have `CREATETAB` privilege on the database, or `SYSADM` or `DBADM` authority.

When the explain command is executed in DB2, these tables get populated with information regarding the relationships between the operators and data objects in the access plan, the SQL compilation environment and the access plan chosen to execute the SQL query. The following diagram shows the relationships between these tables. [2]

Table Name	Description
<code>EXPLAIN_ARGUMENT</code>	Contains information about the unique characteristics for each individual operator, if any.
<code>EXPLAIN_INSTANCE</code>	The main control table for all Explain information. Each row of data in the Explain tables is explicitly linked to one unique row in this table. Basic information about the source of the SQL statements being explained and environment information is kept in this table.
<code>EXPLAIN_OBJECT</code>	Identifies the data objects required by the access plan generated to satisfy the SQL statement.
<code>EXPLAIN_OPERATOR</code>	Contains all the operators needed to satisfy the SQL statement by the SQL compiler.
<code>EXPLAIN_PREDICATE</code>	Identifies the predicates that are applied by a specific

	operator.
EXPLAIN_STATEMENT	<p>Contains the text of the SQL statement as it exists for the different levels of explain information. The original SQL statement as entered by the user is stored in this table with the version used by the optimizer to choose an access plan.</p> <p>When an explain snapshot is requested, additional explain information is recorded to describe the access plan selected by the SQL optimizer. This information is stored in the SNAPSHOT column of the EXPLAIN_STATEMENT table in the format required by Visual Explain. This format is not usable by other applications.</p>
EXPLAIN_STREAM	<p>Represents the input and output data streams between individual operators and data objects. The data objects themselves are represented in the EXPLAIN_OBJECT table. The operators involved in a data stream are represented in the EXPLAIN_OPERATOR table.</p>

Table I: DB2 EXPLAIN PLAN Tables

The following example is used to illustrate EXPLAIN in DB2:

```
EXPLAIN SELECT * FROM T1, T2, T3, T1 T0 WHERE T1.id = T2.id
AND T2.id = T3.id AND T3.id = T0.id;
```

The corresponding plan tree of this query plan (as obtained from the VisualExplain Tool in DB2) is shown in the figure below.

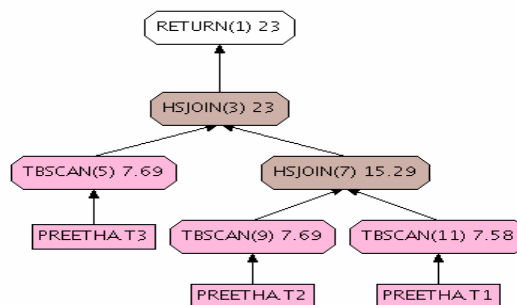


Fig I: Example of a Plan Tree in DB2

The explain_time is retrieved from the EXPLAIN_STATEMENT table for the corresponding text of our query. This parameter is then used to query the EXPLAIN_OPERATOR and EXPLAIN_OBJECT tables to extract the operator_type, total_cost and object_name information. Depending on whether operator_type is a Join or a Scan, the extracted information is inserted into the PLAN_TABLE with the corresponding

type set to OPERATOR or TABLE.

In addition to retrieving the appropriate data from the EXPLAIN tables, and identifying the operator and table nodes, the tree structure for this query plan should be created. The `buildTree()` method is called to build the plan tree by creating table and operator nodes, by using the information from the PLAN_TABLE. Depending on the TYPE field, either an instance of a `TableNode` or an `OperatorNode` is created and added to the tree. Nodes in the plan tree are ordered by node ID such that an in-order traversal of the tree yields the nodes in ascending order.

```
public abstract QueryStat timeQuery(String sqlQuery, PlanNode  
plan,  
long cardinality)
```

This method retrieves the execution time for the indicated SQL query. Before executing the given SQL query, it gets and records the current system time. It then executes the query and again records the current system time. The difference between the first system time and the second system time gives the query execution time. To eliminate the potential system influence that may introduce delay in executing the SQL query, the SQL query is executed several times using this method. The minimum time recorded is chosen as the required SQL query execution time.

```
public abstract void setVirtualTableCardinality(String  
tableName,  
long cardinality, RepeatableRandom repRand)
```

To set the table cardinality for tables in DB2 a manual method is adopted. A user with the appropriate admin privileges can manipulate system table information in DB2. Without admin privileges, an alternative is to directly modify the table cardinality by adding or deleting tuples from the table. Initially tuples are inserted into the table to reach the maximum hypothetical table cardinality defined in the experiment XML file. After getting the query plan, the necessary number of tuples are deleted from the table to change its cardinality. Thus the query plan is obtained for different table cardinalities ranging from the maximum value to zero.

Parameters:

`tableName`: The table whose cardinality is to be changed.

`cardinality`: The cardinality value to be set for the table.

`repRand`: This parameter is used only when the table cardinality is to be altered by inserting tuples. In DB2, table cardinality is changed by deleting tuples and hence this parameter is not used.

```
public abstract String[] getPlanProperties()
```

This method retrieves the properties returned by all query plans in DB2. These are OBJECT_TYPE, SELECT_TYPE, OPERATION, OBJECT_COST, TOTAL_COST, ROW_COUNT, WIDTH.

```
public abstract String[] getPlanOperators()
```

This method retrieves all the possible operator names returned in the query plan in DB2. The

current implementation includes the DB2 operators: HSJOIN, NLJOIN, MSJOIN, IXSCAN and TBSCAN. This list is not comprehensive and is open to extension in future work.

```
public abstract void  
installExperimentTables(DataDefinitionModule myDataDef, String  
myPrefix, String[] myTables)
```

This method creates the tables to store the experiment results with the name defined in the parameter `myTables`. The column name, data type, primary key, foreign key information can be obtained by using `myTables` and `myDataDef` parameters. By calling `createTable()` with these table details, the SQL command can be executed to create the required table in DB2.

Parameters:

`MyDataDef`: This parameter defines the tables and referential integrity of the experiment tables. Below some implementation details about this parameter are listed.

- To get each column name, `myDataDef.getTableColumns(myTables[i])` is used. This returns a string array which includes all column names of table `i`.
- To get the data type for the j^{th} column of table `i`, `myDataDef.getColumnDataType(myTables[i], columns[j])` is used, where `columns[]` is a string array containing all the column names of table `i`.
- To get the length of the data type for the j^{th} column of table `i`, `myDataDef.getColumnDataLength(myTables[i], columns[j])` is used.
- To get the primary key constraints of table `i`, `myDataDef.getTablePrimaryKey(myTables[i])` is used. This returns a string array containing all the primary keys of table `i`.
- To get the foreign key constraints of table `i`, `myDataDef.getTableForeignKeys(myTables[i])` is used. This returns a string array containing all the foreign keys of table `i`.

`myPrefix`: string which defines the prefix for each table to be installed.

`myTables`: string array which defines all the table names to be installed.

By concatenating `myPrefix` and `myTables[i]`, the complete name for table `i` is obtained as a string. These are the tables which are created using the `create table` SQL command.

3.2.2 GeneralDBMS methods

`GeneralDBMS` is a super class of `DB2Subject`. It defines four abstract methods, which need to be implemented.

```
public abstract String getDBMSDriverClassName()
```

This method retrieves the specific DBMS's JDBC driver class name for `AZDBLab` to register the driver for this DBMS. In DB2, it returns the JDBC driver class name as a string:

```
"com.ibm.db2.jcc.DB2Driver"
```

```
public String getDBVersion()
```

This method returns the specific DBMS detailed version information as a string. In DB2, by submitting a query of `SELECT DB2_VERSION FROM EXPLAIN_INSTANCE`, you can get the version information of it. The result obtained from the above query is:

```
DB2_VERSION
-----
09.01.0
```

```
public boolean tableExists(String table)
```

This method checks whether the table exists in DB2. If the table has already been created in the DBMS, it returns `TRUE`. Else, it returns `FALSE`.

Parameter:

table: String to represent the table name.

```
protected String getDataTypeAsString(int dataType, int length)
```

This method produces the data type representation for the specific DBMS given the data type supported by AZDBLab. Currently, AZDBLab supports three types of data, they are: number (integer), variable length character (varchar) and clob. Thus, this method returns the corresponding data type string used in DB2 for these three data types. These are defined as: `INT`, `VARCHAR` and `CLOB`.

Parameter:

dataType: Integer representing a specific AZDBLab data type.

length: It is a valid integer only when dataType parameter represents variable length character data type.

3.2.3 DBMS identifier

In `GeneralDBMS`, a constant string is defined to represent the newly added DB2 DBMS. This string is defined as:

```
public static final String DBMS_TYPE_DB2 = "db2";
```

The `getDBMSID()` method is also modified to support DB2. This method returns the DBMS ID for the given DBMS name. The codes added for DB2 are shown below.

```
if (strdbms.equals(DBMS_TYPE_DB2)) return 4;
```

The current count of DBMS supported by AZDBLabs is incremented in order to ascertain the id for the new DBMS. AZDBLabs includes implementations for Oracle, MySQL, SQLServer, and PostgreSQL DBMS with ids starting from 0. Hence the DB2 system is given the id 4.

3.2.4 ExperimentModule constructor

The constructor for `ExperimentModule` class checks if the system supports the specific DBMS. To support DB2 as a valid database system, it is added in the `ExperimentModule` constructor. This modification is shown below.

```
if (myDBMS.equals(GeneralDBMS.DBMS_NAME_DB2)) {
    _expsubject = new DB2Subject(myUsername,
                                myPassword, myConnectionString);}
```

3.2.5 Schema configuration file

In order to check whether the new DBMS experiment file passes the schema validation test, the schema configurations `experiment.xsd` is modified. To add DB2, the value should be added in the format below.

```
<xsd:enumeration value='DB2' />
```

3.2.6 lib file

The jar files added to the lib directory for running DB2 are, `db2jcc_license_cu.jar`, `db2jcc.jar`

4. Testing an Experiment Subject

Testing of an `experiment_subject` is carried out by creating experiment files in XML format and running them on the new DBMS. The XML file format is defined in `experiment.xsd` schema configuration file. The required information for the experiment file includes the user name and password string to access the corresponding DBMS, the variable table and fixed table definitions used to execute the query, the SQL query executed to get the query plans, the experiment table data including table names, column names and column type for each table.

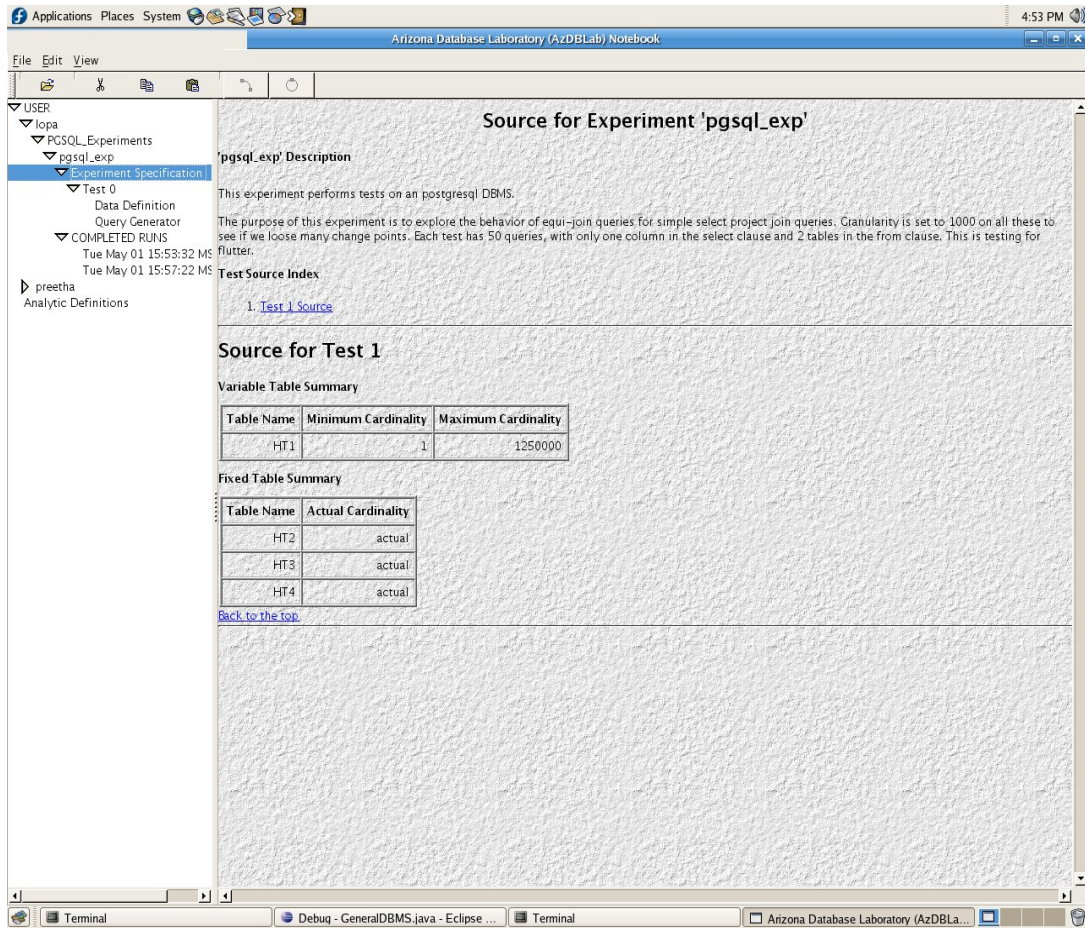


Fig II: Screenshot of a sample experiment run in PostgreSQL

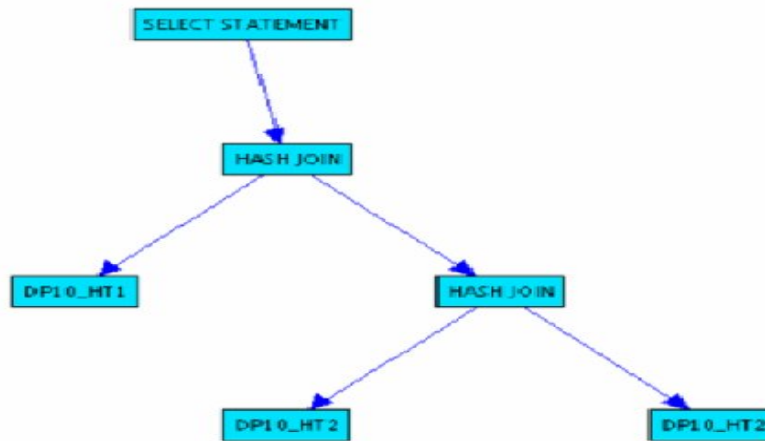


Fig III: Result plan tree for a sample experiment using PgsqSubject

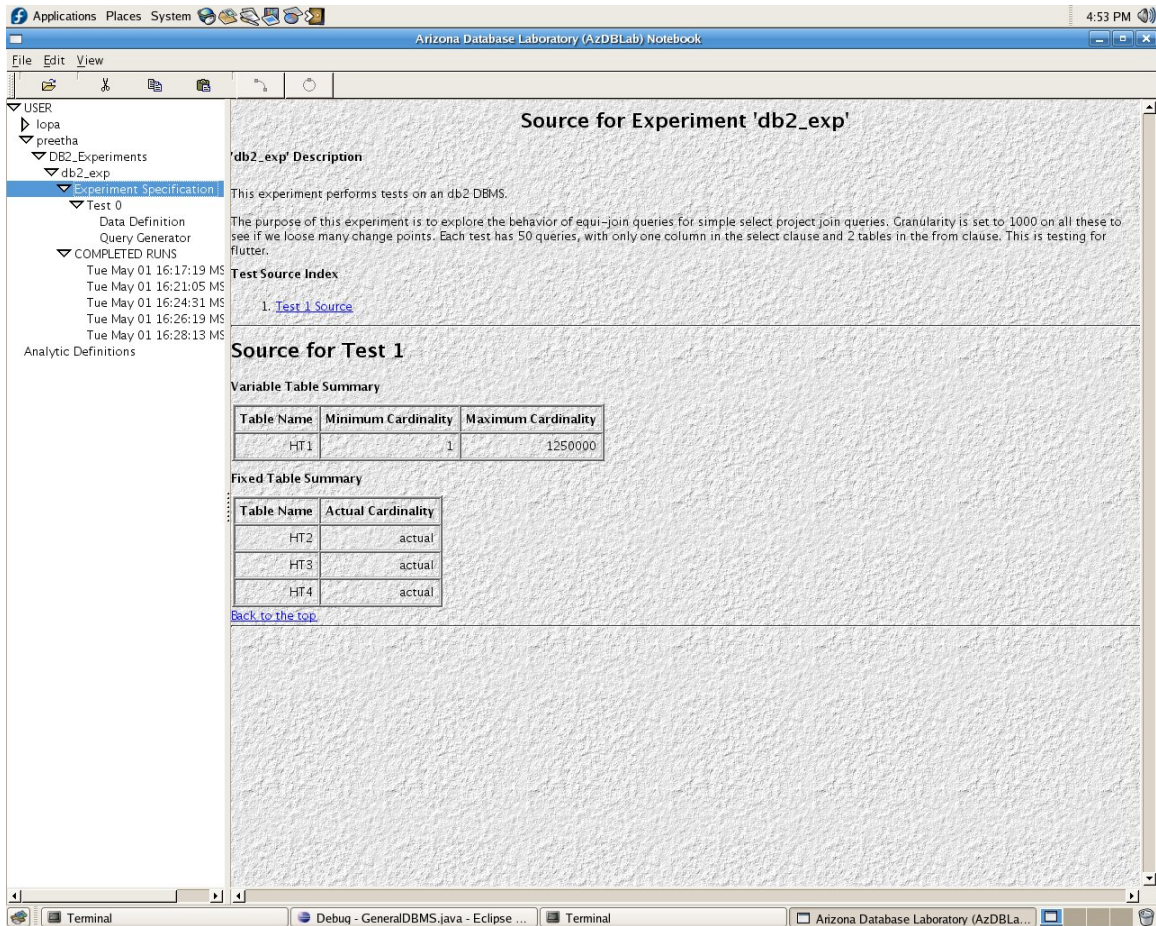


Fig IV: Screenshot of a sample experiment run in DB2

5. Summary and Future Work

The db2 and pgsq1 directories in OSAT/osat/dbms/db2 and OSAT/osat/dbms/pgsq1 respectively are used to store the source codes. The number of lines of code written for DB2Subject is 1167 and that for Pgsq1Subject is 1277 approximately.

Thus the main goals achieved in the project were:

1. Integration of two new DBMS into existing infrastructure
2. Study of the behavior of DB2 and PostgreSQL with respect to
 - Data types and supported operations
 - DBMS specific interfaces
 - Query optimization handling
 - System tables

Future work may involve using the two new DBMSes to conduct flutter and other query optimization experiments. Also, this project might be used to make the process of integrating a new DBMS into AZDBLabs easier.

References

- [1]. PostgreSQL online manual [<http://www.postgresql.org/docs/8.2/interactive/index.html>]
- [2]. IBM DB2 Product manual [<http://publib.boulder.ibm.com/infocenter/db2luw/v8//index.jsp>]
- [3]. Understanding DB2 Learning Visually with examples, Raul F. Chong et.al.