You are supposed to develop a Java codebase that allows for the evaluation certain types of database queries, known apriori, specified in SQL.

You may assume a storage manager that provides you with a stream of tuples. Such a stream of tuples may be accessible through a JDBC connection.

Candidate storage managers are MySQL, PostgresSQL, MonetDB, Berkeley DB, Apache Derby, etc. When, using a full–blown DBMS as a storage manager, you should avoid using its query processing capabilities. Few exceptions are to scan the tuples in stored tables, store tuples in temporary tables, and compute certain statistical metadata for the existing tables.

For each database table already defined, there is an input tuple stream. The order in which tuples are retrieved from such an input stream, can be random or sorted in a certain order. Input tuple streams for existent database tables return tuples in an random order. Some of them, may return tuples in a sorted order (where the order is to be determined).

You may materialize temporary relations, by creating temporary database tables and store tuples in them via an output tuple stream. Care is needed to avoid severe performance degradation.

The database tables available are those in http://www.csee.umbc.edu/~kalpakis/courses/661/sampleDB/sampleDB.html,

account(account_number*, branch_name, balance);
branch(branch_name*, branch_city, assets);
customer(customer_name*, customer_street, customer_city);
loan(loan_number*, branch_name);
depositor(customer_name*, account_number*);
borrower(customer_name*, loan_number*);

Assume that all the obvious referential integrity constraints hold. Sample data to populate these tables are provided at the URL above.

(a) Develop appropriate classes for handling tuples of various schemas. Assume that all the tuple schemas are known ahead of time (at compile time).

(b) Develop appropriate iterators that allow you to implement the following types of SQL queries.

(i) SELECT–PROJECT queries:

   1. SELECT *
      FROM customer;
2. SELECT account_number
   FROM account
   WHERE balance BETWEEN 700 AND 900;

3. SELECT customer_name
   FROM customer
   WHERE customer_street LIKE '%Hill'

4. SELECT DISTINCT customer_name
   FROM customer
   WHERE customer_street LIKE '%Hill';

5. SELECT *
   FROM loan
   ORDER BY amount DESC, loan_number ASC;

(ii) Grouping and aggregation:

1. SELECT COUNT(*)
   FROM customer;

2. SELECT branch_name, AVG(balance)
   FROM account
   GROUP BY branch_name;

3. SELECT branch_city, SUM(assets) AS total_assets
   FROM branch
   GROUP BY branch_city
   ORDER BY total_assets DESC;

4. SELECT branch_name, AVG(balance)
   FROM account
   GROUP BY branch_name
   HAVING AVG(balance) > 700;

5. SELECT branch_name, COUNT(distinct customer_name)
   FROM depositor, account
   WHERE depositor.account_number = account.account_number
   GROUP BY branch_name;

(iii) JOINs with two and three tables:

1. SELECT distinct customer.customer_name, customer_city
   FROM borrower, customer
   WHERE borrower.customer_name = customer.customer_name;

2. SELECT DISTINCT T.branch_name
   FROM branch T, branch S
   WHERE T.assets > S.assets AND S.branch_city = 'Brooklyn';
3. SELECT DISTINCT C.customer_name, customer_city
   FROM customer C, borrower B, loan L
   WHERE C.customer_name = B.customer_name AND
       B.loan_number = L.loan_number AND
       branch_name = 'Perryridge';

4. SELECT branch_name
   FROM branch
   WHERE assets > ALL
     (SELECT assets
      FROM branch
      WHERE branch_city = 'Brooklyn');

5. SELECT DISTINCT customer_name
   FROM borrower, loan, branch
   WHERE borrower.loan_number = loan.loan_number AND
       loan.branch_name = 'Perryridge'
   ORDER BY borrower.customer_name;

You may assume that there is sufficient main memory available so that two–passes are always sufficient for each iterator.

(c) Implement each query above in your codebase. Indicate the expression tree of iterators implemented for evaluating each query.

(d) Provide an appropriate user interface (UI) for interacting with your implementation in (c). A web–based UI is preferred, though a command–line UI is also acceptable.

(e) Consider the grouping and aggregation queries. Extend your implementation so that you can indicate the running aggregates and groups on your UI, during the execution of your queries. Provide an indication to the user, when the query completed its execution.

(f) Extend your implementation to provide at your UI, for the result set of each query, an estimate of the number and size of the tuples, and the number of distinct values for each attribute of these tuples.

You are allowed to maintain a small fixed–size amount of metadata for each existing database table. You may use appropriate SQL statements to compute such metadata for the existing tables.

(g) Consider the grouping and aggregation queries. Extend your implementation to provide on your UI a way for the user to indicate, at runtime, a group for which s/he is interested in its running aggregate. In response to that, your implementation should attempt to provide higher accuracy (process tuples from this group with higher priority).
(h) Write a project report describing your solution approach.

Your codebase should be in Java (the latest version recommended by Sun Microsystems is Java JDK 6 Update 12). You may develop your code on any system of your choice. Your code should be running on a Windows XP or Linux machine with the Java JDK 6 Update 12 installed.

Your implementation will be evaluated for correctness, design, extensibility, performance. It is expected, that it would be straightforward and simple to extend your codebase to handle a new SQL query, similar in type to the ones above.