ANSI SQL Isolation Levels

A Summary of the Original Paper “A Critique of ANSI SQL Isolation Levels” with concrete SQL-DML Examples

Seminar Thesis

Master of Science in Engineering
Specialization: Information and Communication Technologies (ICT)

Advisor: Prof. Martin Studer
Author: Stefan Lütolf

Version: 2.1
Date: 14/01/2014
Abstract

The Isolation is one of the four ACID properties in a database management system. It depends on the behaviour of the database in multi user mode. Concurrent transactions can execute at different isolation levels. Choosing the isolation level depends on the consideration of performance and consistency. Lower isolation levels increase transaction concurrency, but have the risk to get invalid data. The highest isolation level is perfect serializability. Running concurrent transactions at lower isolation level can access states that are not committed or never really existed.

The ANSI SQL-92 [1] standard defines four isolation levels based on the serializability definition and the three actions, called phenomena. These phenomena are “Dirty Read”, “Non-repeatable Read”, and “Phantom”. The paper “A Critique of ANSI SQL Isolation Levels” [2] criticizes the definition of the isolation levels and gives proposals to define these more specific. It shows weaknesses in the anomaly approach to define isolation levels, and contains a formal redefinition.

This paper gives a less abstract summary about the original paper mentioned above [2]. It contains the parts about the four ANSI SQL isolation levels and the critique about it. Further, it shows examples of the discussed topics on concrete SQL-DML samples on Oracle Java DB [3].
Table of Contents

1 Introduction ......................................................................................................................... 5
   1.1 Topic Description ........................................................................................................... 5
   1.2 Content of the Chapters ............................................................................................... 5

2 Isolation Level Concepts .................................................................................................. 6
   2.1 Serializability Concepts ............................................................................................... 6
   2.2 ANSI SQL Isolation Levels .......................................................................................... 9

3 Analysis of ANSI SQL Isolation Levels ........................................................................... 13
   3.1 Notation ....................................................................................................................... 13
   3.2 Simulation of Isolation Levels ..................................................................................... 13
   3.3 Dirty Read .................................................................................................................. 13
   3.4 Non-repeatable or Fuzzy Read ...................................................................................... 16
   3.5 Phantom ..................................................................................................................... 18
   3.6 Summary ..................................................................................................................... 20

4 Implementation of Isolation Levels ................................................................................... 21
   4.1 PostgreSQL ............................................................................................................... 21
   4.2 Oracle Java DB .......................................................................................................... 21

5 Bibliography ..................................................................................................................... 22

6 Annex ................................................................................................................................. 23
   6.1 UML Class Diagram .................................................................................................. 23
   6.2 Code Listings ............................................................................................................. 24
   6.3 Simulation of A1 ........................................................................................................ 34
   6.4 Simulation of H1 ........................................................................................................ 35
   6.5 Simulation of A2 ........................................................................................................ 36
   6.6 Simulation of H2 ........................................................................................................ 37
Table of Figures

Figure 1 dependency graph for Table 1 schedule 1 and Table 2 schedule 2 ............................................ 7
Figure 2 dependency graph of the schedule 2 ......................................................................................... 8
Figure 3 Anomalies "Dirty Read".............................................................................................................. 9
Figure 4 Anomalies "Non-repeatable Read".......................................................................................... 10
Figure 5 Anomalies "Phantom"............................................................................................................... 11
Figure 6 History H1 .................................................................................................................................. 15
Figure 7 History H2 .................................................................................................................................. 17
Figure 8 History H3 .................................................................................................................................. 19
Figure 9, a diagram of the isolation levels and their relationships ......................................................... 20

Table of Charts

Table 1 schedule 1 ....................................................................................................................................... 7
Table 2 schedule 2 ....................................................................................................................................... 7
Table 3 schedule 3 ....................................................................................................................................... 8
Table 4 ANSI SQL Isolation Levels defined in terms of the tree phenomena ......................................... 12
1 Introduction

1.1 Topic Description

Concurrent transactions can execute at different isolation levels. Choosing the isolation level depends on consideration of performance and consistency. Lower isolation levels increase transaction concurrency but have the risk to get invalid data. The highest isolation level is perfect serializability. Running concurrent transactions at lower isolation level, they can access states that are not committed or never really existed.

The ANSI SQL-92 [1] specifications defines four isolation levels. The definition based on the classical serializability definition plus three prohibited action subsequences, which are called phenomena. Those are “Dirty Read”, “Non-repeatable Read” and “Phantom”. These phenomena causes anomalous behaviour, which is not serializable.

The paper “A Critique of ANSI SQL Isolation Levels” [2] criticizes the definition of the isolation levels of ANSI SQL-92 and gives proposals to redefine it. It shows weaknesses in the anomaly approach to define isolation levels, and contains a formal redefinition.

The content of this paper is a less abstract summary of the original paper. It contains the parts about the four ANSI SQL isolation levels and the critique about it. Further, it shows examples of the discussed topics on concrete SQL-DML samples on Oracle Java DB [3]. Chapter 1.2 lists the content of the chapters in this paper as an overview. The parts about locking and other isolation types are not treated in this paper.

In other word, the chapters 1, 2 (without chapter 2.3 “Locking”) and 3 of the original paper “A Critique of ANSI SQL Isolation Levels” are treated in this paper.

1.2 Content of the Chapters

In chapter 2 “Isolation Level Concepts” the concepts of the isolation levels specification are declared. The basic concept of conflict Serializability is described. Further, the three anomalies are explained by the ANSI SQL standard. For all of these anomalies, an example is shown as a figure.

In chapter 3 “Analysis of ANSI SQL Isolation Levels” the critique of the original paper [2] is listed. The formal redefinition is explained by SQL-DML examples and a figure.

Some database systems are mentioned in chapter 4 “Implementation of Isolation Levels”. It shows, how they implemented the isolation levels.
2 Isolation Level Concepts

In this chapter, the concepts of the isolation levels specification are declared.

The ANSI SQL-92 [1] specifications contain the definition of four isolation levels. The definition is based on the classical serializability definition plus three prohibited action subsequence, which are called phenomena. These are “Dirty Read”, “Non-repeatable Read” and “Phantom”, which cause anomalous behaviour.

2.1 Serializability Concepts

In the paper, the used serializability theory is the conflict serializability.

A history contains actions, such as reads and writes of data items, from one or more transactions. A serial history contains one transaction at a time in sequence.

A conflict is, if two actions in a history operate with the same data item and at least one of those is a write action.

A dependency graph shows how the transactions are interdependent. Committed transactions are represented as graph nodes. If an action of transaction T1 conflicts an action of transaction T2, it is shown as an edge in the dependency graph.

Two histories are equivalent if they have the same committed transactions and if they have the same dependency graph.

A history is serializable if its dependency graph is equivalent to a serial history.
An *example* of a history, which is conflict serializable.

Transaction T1 and T2 performs some read and write actions, listed in Table 1 schedule 1.

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

**Table 1 schedule 1**

The equivalent serial history of the history in Table 1 schedule 1:

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td></td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Read(B)</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td></td>
</tr>
<tr>
<td>Write(B)</td>
<td>Read(A)</td>
</tr>
<tr>
<td></td>
<td>Write(A)</td>
</tr>
<tr>
<td></td>
<td>Read(B)</td>
</tr>
<tr>
<td></td>
<td>Write(B)</td>
</tr>
</tbody>
</table>

**Table 2 schedule 2**

In Figure 1, there is the dependency graph of the serializable history for Table 1 schedule 1 and Table 2 schedule 2.

![Figure 1 dependency graph for Table 1 schedule 1 and Table 2 schedule 2](image-url)
An example of a history, which is not conflict serializable:

Transaction T1 and T2 performs some read and write actions, listed in Table 3 schedule 3.

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A)</td>
<td>Write(A)</td>
<td>Read(A)</td>
</tr>
<tr>
<td>Write(A)</td>
<td>Read(B)</td>
<td>Write(B)</td>
</tr>
<tr>
<td>Read(B)</td>
<td>Write(B)</td>
<td></td>
</tr>
</tbody>
</table>

Table 3 schedule 3

There is no equivalent serial history of the history above!

In Figure 2, there is the dependency graph of a non serializable history for Table 3 schedule 3.

Figure 2 dependency graph of the schedule 2

The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.
2.2 ANSI SQL Isolation Levels

The ANSI SQL-92 [1] specification of the isolation levels are on an abstract layer, which admits more than just the locking implementation. The Isolation levels are defined by the anomalies, which are prohibited action subsequences. These are:

Dirty Read:

“SQL-transaction T1 modifies a row. SQL-transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed and that may thus be considered to have never existed.” [1]

A possible sequence for the mentioned anomalies can be seen in Figure 3.

![Dirty Read Diagram]

In Figure 3 transaction T1 writes the value 50 to x and later rolls back the transaction. Therefore transaction T2 should not read the value 50.
Non-repeatable Read:

“SQL-transaction T1 reads a row. SQL-transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.” [1]

A possible sequence for the mentioned anomalies can be seen in Figure 4.

In Figure 4 transaction T1 reads the value of x (15). Transaction T2 writes the value 50 to x and performs a commit. Transaction T1 rereads the value of x (50) and it is different to the first read action. Therefore transaction T2 should not write the value 50 to x.
Phantom

“SQL-transaction T1 reads the set of rows N that satisfy some <search condition>. SQL-transaction T2 then executes SQL-statements that generate one or more rows that satisfy the <search condition> used by SQL-transaction T1. If SQL-transaction T1 then repeats the initial read with the same <search condition>, it obtains a different collection of rows.” [1]

A possible sequence for the mentioned anomalies can be seen in Figure 5.

In Figure 5 transaction T1 reads a list of employees. Transaction T2 inserts a new employee and performs a commit. Transaction T1 reads the list of employees again and gets a different set of employees. Therefore transaction T2 should not insert the new employee.
Difference between Non-repeatable read and Phantom read

Non-repeatable read means, that a row that T1 has queried has a different value the second time (because T2 changed the value of the row in between).

Phantom read means, that all the rows in a query of T1 have the same value before and after, but different rows are being selected by T2 (deleted or inserted some rows)

Isolation Levels

The ANSI SQL defines four levels of isolation by the matrix of Table 4. Each level is defined by the phenomena, which a transaction is forbidden to experience. A special case is the serializable level. It is not defined as the level, on which all of the three phenomena are forbidden. The serializable isolation level provide fully serializable execution (see 2.1 Serializability Concepts). Because of this, the fourth level is called “anomaly serializable”.

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Non-repeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI read uncommitted</td>
<td>possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>ANSI read committed</td>
<td>not possible</td>
<td>possible</td>
<td>possible</td>
</tr>
<tr>
<td>ANSI repeatable read</td>
<td>not possible</td>
<td>not possible</td>
<td>possible</td>
</tr>
<tr>
<td>Anomaly serializable</td>
<td>not possible</td>
<td>not possible</td>
<td>not possible</td>
</tr>
</tbody>
</table>

Table 4 ANSI SQL Isolation Levels defined in terms of the three phenomena

Difference between “serializable” and “anomaly serializable”

The Serializable isolation level is not defined solely in terms of these phenomena. That is the reason to take another name for the Isolation level. The fourth level is called “anomaly serializable, because the level provide fully serializable execution.
3 Analysis of ANSI SQL Isolation Levels

This chapter contains a critique about the definition of the isolation levels of ANSI SQL-92 and gives proposals to redefine it. It shows weaknesses in the anomaly approach to define isolation levels and contains formal redefinition.

3.1 Notation

In the chapters 3.3, 3.4 and 3.5 an abstract language is used to describe the histories. The following notation is necessary to write the histories abstract and short.

T₁, T₂ ... Transaction T₁, T₂ ...
A₁, A₂ ... ANSI SQL Phenomena ("strict interpretation")
P₁, P₂ ... redefinition Phenomena ("broader interpretation")
H₁, H₂ ... Histories H₁, H₂ ...
w₁ [x] write by T₁ on data item x
r₂ [x] read of data item x by T₂
w₁ [P] writing a set of records satisfying predicate P by T₁
c₁ T₁ commit
a₁ T₁ abort

For example, the anomaly "Dirty Read" can be written as:

\[ A₁: w₁[x]... r₂[x]... (a₁ and c₂ in either order) \]

In other words, the ANSI SQL Phenomena (A₁) starts with a write action by transaction 1 on the data item x (w₁[x]). Transaction 2 reads the value of the data item x (r₂[x]). Then transaction 1 performs a rollback (a₁) and transaction 2 a commit (c₂), in either order.

3.2 Simulation of Isolation Levels

To analyse a database system, such as Oracle Java DB [3], about the implementation of the isolation levels, a self-made java-based simulation can be used. The simulation can run the histories mentioned in chapters 3.3, 3.4 and 3.5. In this paper, all of the simulations are made with the Oracle Java DB [3], but it is also possible to use another database instead. The UML class diagram (see 6.1) and the code listings (see 6.2) are listed in the Annex.

3.3 Dirty Read

By Table 4, the histories under READ COMMITTED isolation must not contain the anomaly "Dirty Read" (see Figure 3). This anomaly can be written as A₁:

\[ A₁: w₁[x]... r₂[x]... (a₁ and c₂ in either order). \]

The following history H₁ does not violates A₁:

\[ H₁: r₁[x=50] w₁[x=10] r₂[x=10] r₂[y=50] c₂ r₁[y=50] w₁[y=90] c₁ \]

The paper [2] proposes a broad interpretation of A₁, the phenomenon P₁:

\[ P₁: w₁[x]... r₂[x]... ((c₁ or a₁) and (c₂ or a₂) in any order) \]

H₁ violates P₁, so that we should take the broader interpretation P₁ rather than A₁.
The result of the simulation (see 6.3) shows, that the anomaly A1 (see Figure 3) occurs under READ UNCOMMITTED isolation but is forbidden under READ COMMITTED isolation.

The history H1 (see Figure 6) is similar to a bank account transfer. The result of the simulation (see 6.4) of H1 shows, that the anomaly A1 occurs under READ UNCOMMITTED, but under READ COMMITTED a lock could not be obtained within the time requested. The reason is, that Apache Derby uses locking. Locking is based on P1, the broader interpretation of A1 [2].
Figure 6 History H1
3.4 Non-repeatable or Fuzzy Read

By Table 4, the histories under REPEATABLE READ isolation must not contain the anomaly “Non-repeatable Read” (see Figure 4). This anomaly can be written as A2:

\[ A2: r_1[x] \ldots w_2[x] \ldots c_2 \ldots r_1[x] \ldots c_1 \]

The following history H2 does not violate A2:

\[ H2: r_1[x=50] \ r_2[x=50] \ w_2[x=10] \ r_2[y=50] \ w_2[y=90] \ c_2 \ r_1[y=90] \ c_1 \]

The paper [2] proposes a broad interpretation of A2, the phenomenon P2:

\[ P2: r_1[x] \ldots w_2[x] \ldots ((c_1 \text{ or } a_1) \text{ and } (c_2 \text{ or } a_2) \text{ in any order}) \]

H2 violates P2, so that we should take the broader interpretation P2 rather than A2.

The result of the simulation (see 6.5) shows, that the anomaly A2 (see Figure 4) occurs under READ COMMITTED isolation but is forbidden under REPEATABLE READ isolation.

The result of the simulation (see 6.6) of H2 (see Figure 7) shows, that the anomaly A2 occurs under READ COMMITTED, but under REPEATABLE READ a lock could not be obtained within the time requested. The reason is, that Apache Derby uses locking. Locking is based on P2, the broader interpretation of A2 [2].
Figure 7 History H2
3.5 Phantom

By Table 4, the histories under ANOMALY SERIALIZABLE isolation must not contain the anomaly "Phantom" (see Figure 5). This anomaly can be written as A3:

\[ A3: r1[P]... w2[y in P]... c2 ... r1[P] ... c1 \]

The following history H3 does not violates A3:

\[ H3: r1[P] w2[y in P] r2[z] w2[z] c2 r1[z] c1 (z is the number of items in P) \]

The paper [2] proposes a broad interpretation of A3, the phenomenon P3:

\[ P3: r1[P]... w2[y in P]... ((c1 or a1) and (c2 or a2) in any order) \]

H3 violates P3, so that we should take the broader interpretation P3 rather than A3.

The simulation is not yet implemented for the Histories A3 and H3. The theory shows, that the anomaly A3 (see Figure 5) occurs under REPEATABLE READ isolation but is forbidden under ANOMALY SERIALIZABLE isolation. The history H3 (see Figure 8) shows, that the anomaly A3 occurs under REPEATABLE READ, but under ANOMALY SERIALIZABLE a lock could not be obtained within the time requested. The reason is that, Apache Derby uses locking. Locking is based on P3, the broader interpretation of A3 [2].
Figure 8 History H3
3.6 Summary

In Figure 9 there is a diagram of the isolation levels and their relationships. The edges are annotated with the phenomena that differentiate the isolation levels. Not shown are other isolation types, like cursor stability or snapshot isolation. The other isolation types are discussed in the paper [2].

Figure 9, a diagram of the isolation levels and their relationships
4 Implementation of Isolation Levels

The ANSI SQL 92 [1] standard defines the four isolation levels. The implementation of the isolation levels in the database systems could be different as the standard. In this chapter, there are some database systems with their implementation of the isolation levels.

4.1 PostgreSQL

“In PostgreSQL, you can request any of the four standard transaction isolation levels. But internally, there are only three distinct isolation levels, which correspond to the levels Read Committed, Repeatable Read, and Serializable. When you select the level “Read Uncommitted” you really get “Read Committed”, and phantom reads are not possible in the PostgreSQL implementation of “Repeatable Read”, so the actual isolation level might be stricter than what you select. This is permitted by the SQL standard: the four isolation levels only define which phenomena must not happen, they do not define which phenomena must happen. The reason that PostgreSQL only provides three isolation levels is that this is the only sensible way to map the standard isolation levels to the multiversion concurrency control architecture.” [4]

4.2 Oracle Java DB

Java DB provides all of the four transaction isolation levels (more information about the implementation, see [5]).
5 Bibliography


6 Annex

6.1 UML Class Diagram
6.2 Code Listings

Test.java

```java
package ch.htwchur.mse.sem.test;
import ch.htwchur.mse.sem.Database;
import ch.htwchur.mse.sem.History;
import ch.htwchur.mse.sem.IsolationLevel;
import ch.htwchur.mse.sem.OutputLanguage;
import ch.htwchur.mse.sem.Simulation;
import ch.htwchur.mse.sem.databases.ApacheDerbyDatabase;

public class Test {
    public static void main(String[] args) throws Exception {
        Database database = new ApacheDerbyDatabase();
        database.open();

        // "Dirty Read" histories
        History A1 = HistoryBuilder.create("A1", "w1[x=10] r2[x] a1 c2");
        History H1 = HistoryBuilder.create("H1", "r1[x] w1[x=10] r2[x] r2[y] c2 r1[y] w1[y=90] c1");

        // "Non-repeatable Read" histories
        History A2 = HistoryBuilder.create("A2", "r1[x] w2[x=10] c2 r1[x] c1");
        History H2 = HistoryBuilder.create("H2", "r1[x] r2[x] w2[x=10] r2[y] w2[y=90] c2 r1[y] c1");

        // "Phantom" histories
        // TODO

        Simulation simulation = new Simulation(database);
        // run simulation "Dirty Read"
        simulation.run(A1, IsolationLevel.READ_UNCOMMITTED, OutputLanguage.SQL);
        simulation.run(H1, IsolationLevel.READ_COMMITTED, OutputLanguage.SQL);
        simulation.run(H1, IsolationLevel.READ_UNCOMMITTED, OutputLanguage.SQL);
        simulation.run(H1, IsolationLevel.READ_COMMITTED, OutputLanguage.SQL);

        // run simulation "non-repeatable Read"
        simulation.run(A2, IsolationLevel.READ_COMMITTED, OutputLanguage.SQL);
        simulation.run(A2, IsolationLevel.REPEATABLE_READ, OutputLanguage.SQL);
        simulation.run(H2, IsolationLevel.READ_COMMITTED, OutputLanguage.SQL);
        simulation.run(H2, IsolationLevel.REPEATABLE_READ, OutputLanguage.SQL);
    }
}

ApacheDerbyDatabase.java

```
computeDescription();
}

@Override
public Connection getConnection() throws SQLException {
    return DriverManager.getConnection(connectionURL);
}

private void computeDescription() throws SQLException {
    Connection conn = null;
    try {
        conn = getConnection();
        DatabaseMetaData dbmd = conn.getMetaData();
        String productName = dbmd.getDatabaseProductName();
        String productVersion = dbmd.getDatabaseProductVersion();
        description = productName + "" + productVersion;
    } finally {
        DbUtils.close(conn);
    }
}

@Override
public String getDescription() {
    return description;
}

@Override
public void dropTable(String tableName) throws SQLException {
    Connection conn = null;
    Statement statement = null;
    try {
        conn = getConnection();
        String dropString = "DROP TABLE " + tableName;
        statement = conn.createStatement();
        statement.execute(dropString);
    } catch (SQLSyntaxErrorException ex) {
        if (!"42Y55".equals(ex.getSQLState())) {
            throw ex;
        }
    } finally {
        DbUtils.close(statement);
        DbUtils.close(conn);
    }
}

package ch.htwchur.mse.sem.actions;
import java.sql.*;
import org.apache.commons.dbutils.DbUtils;
import ch.htwchur.mse.Action;
import ch.htwchur.mse.Transaction;
public class WriteAction implements Action {
    private String id;
    private int amount;
    public WriteAction(String id, int amount) {
        this.id = id;
        this.amount = amount;
    }
    public String execute(Transaction transaction) throws SQLException {
        Connection connection = transaction.getConnection();
        PreparedStatement preparedStatement = null;
        try {
            String statementSQL = "UPDATE accounts SET amount = ? WHERE id = ?";
            preparedStatement = connection.prepareStatement(statementSQL);
            preparedStatement.setInt(1, amount);
            preparedStatement.setString(2, id);
            preparedStatement.setQueryTimeout(1);
            preparedStatement.executeUpdate();
            return null;
        } finally {
            }
package ch.htwchur.mse.sem.actions;

import java.sql.*;
import org.apache.commons.dbutils.DbUtils;
import ch.htwchur.mse.sem.Action;
import ch.htwchur.mse.sem.Transaction;

public class CommitAction implements Action {
    public String execute(Transaction transaction) throws SQLException {
        Connection connection = transaction.getConnection();
        String selectStatementSQL = "SELECT amount FROM accounts WHERE id=?";
        PreparedStatement preparedStatement = connection.prepareStatement(selectStatementSQL);
        preparedStatement.setString(1, id);
        ResultSet resultSet = preparedStatement.executeQuery();
        return resultSet.getString("amount");
    }

    @Override
    public String getDescription(Transaction transaction) {
        return "c" + transaction.getId() + "[" + id + "]";
    }

    @Override
    public String getSQLDescription(Transaction transaction) {
        return "SELECT amount FROM accounts WHERE id=" + id;
    }
}

package ch.htwchur.mse.sem.actions;

import java.sql.Connection;
import java.sql.SQLException;
import ch.htwchur.mse.sem.Action;
import ch.htwchur.mse.sem.Transaction;

public class CommitAction implements Action {
    public String execute(Transaction transaction) throws SQLException {
        Connection connection = transaction.getConnection();
        String selectStatementSQL = "SELECT amount FROM accounts WHERE id=?";
        PreparedStatement preparedStatement = connection.prepareStatement(selectStatementSQL);
        preparedStatement.setString(1, id);
        ResultSet resultSet = preparedStatement.executeQuery();
        return resultSet.getString("amount");
    }

    @Override
    public String getDescription(Transaction transaction) {
        return "c" + transaction.getId() + "[" + id + "]";
    }

    @Override
    public String getSQLDescription(Transaction transaction) {
        return "SELECT amount FROM accounts WHERE id=" + id;
    }
}
connection.commit();
return null;
}

@Override
public String getDescription(Transaction transaction) {
return "c" + transaction.getId();
}

@Override
public String getSQLDescription(Transaction transaction) {
return "COMMIT;";
}
}

AbortAction.java

package ch.htwchur.mse.sem.actions;

import java.sql.Connection;
import java.sql.SQLException;
import ch.htwchur.mse.sem.Action;
import ch.htwchur.mse.sem.Transaction;

public class AbortAction implements Action {
public String execute(Transaction transaction) throws SQLException {
    Connection connection = transaction.getConnection();
    connection.rollback();
    return null;
}

@Override
public String getDescription(Transaction transaction) {
    return "a" + transaction.getId();
}

@Override
public String getSQLDescription(Transaction transaction) {
    return "ROLLBACK;";
}
}

Action.java

package ch.htwchur.mse.sem;

import java.sql.SQLException;

public interface Action {
/**
 * @param transaction
 * @return the result or <code>null</code>
 * @throws SQLException
 */
public String execute(Transaction transaction) throws SQLException;
public String getDescription(Transaction transaction);
public String getSQLDescription(Transaction transaction);
}

Database.java

package ch.htwchur.mse.sem;

import java.sql.Connection;
import java.sql.SQLException;

public interface Database {
    public String getDescription();
    public void open() throws ClassNotFoundException, SQLException;
}
public Connection getConnection() throws SQLException;

public void dropTable(String tableName) throws SQLException;

History.java

package ch.htw.chur.mse.sem;
import java.sql.*;
import java.util.*;
import org.apache.commons.dbutils.DbUtils;

public class History {
    private String description;
    private List<HistoryItem> items;

    public History(String description) {
        this.description = description;
        items = new ArrayList<HistoryItem>();
    }

    public String getDescription() {
        return description;
    }

    public void add(Action action, Transaction transaction) {
        HistoryItem item = new HistoryItem(action, transaction);
        items.add(item);
    }

    public void execute(OutputLanguage language) {
        for (HistoryItem item : items) {
            if (item.isTransactionAborted()) {
                continue;
            }
            try {
                for (int index = 1; index < item.getTransaction().getId(); index++) {
                    System.out.print("\t");
                }
                System.out.print(item.getDescription(language));
                String result = item.execute();
                if (result != null) {
                    System.out.println("\t- " + result);
                } else {
                    System.out.println();
                }
            } catch (SQLException ex) {
                System.out.println("\t- " + ex.getMessage());
                item.setTransactionAborted(true);
            }
        }
    }

    public String getItemDescription(OutputLanguage language) {
        StringBuilder builder = new StringBuilder();
        for (HistoryItem item : items) {
            builder.append(item.getDescription(language));
            builder.append(" ");
        }
        return builder.toString();
    }

    public void open(Database database, IsolationLevel level) throws SQLException {
        for (Transaction transaction : getTransactions()) {
            transaction.setConnection(database.getConnection());
            transaction.setIsolationLevel(level.getLevel());
            transaction.setAborted(false);
        }
    }

    private Collection<Transaction> getTransactions() {
        Set<Transaction> transactions = new HashSet<Transaction>();
        for (HistoryItem item : items) {
            transactions.add(item.getTransaction());
        }
        return transactions;
    }
}
public void close() throws SQLException {
    for (Transaction transaction : getTransactions()) {
        Connection connection = transaction.getConnection();
        DbUtils.close(connection);
    }
}

public void addAll(String actionsString) {
    Matcher actionMatcher = actionPattern.matcher(actionsString);
    while (actionMatcher.find()) {
        String actionString = actionMatcher.group();
        add(actionString);
    }
}

public void add(String actionString) {
    Action action = null;
    int transactionNo = -1;

    // read
    Matcher readPatternMatcher = readPattern.matcher(actionString);
    if (readPatternMatcher.matches()) {
        transactionNo = Integer.parseInt(readPatternMatcher.group(1));
        String id = readPatternMatcher.group(2);
        action = new ReadAction(id);
    }

    // write
    Matcher writePatternMatcher = writePattern.matcher(actionString);
    if (writePatternMatcher.matches()) {
        transactionNo = Integer.parseInt(writePatternMatcher.group(1));
        String id = writePatternMatcher.group(2);
        int amount = Integer.parseInt(writePatternMatcher.group(3));
        action = new WriteAction(id, amount);
    }

    // commit
    Matcher commitPatternMatcher = commitPattern.matcher(actionString);
    if (commitPatternMatcher.matches()) {
        transactionNo = Integer.parseInt(commitPatternMatcher.group(1));
    }
}
```java
action = new CommitAction();
}
// rollback
Matcher abortPatternMatcher = abortPattern.matcher(actionString);
if (abortPatternMatcher.matches()) {
    transactionNo = Integer.parseInt(abortPatternMatcher
        .group(1));
    action = new AbortAction();
}
if (action == null) {
    System.out.println("Unknown action");
    return;
}
Transaction transaction = transactions.get(transactionNo);
if (transaction == null) {
    transaction = new Transaction(transactionNo);
    transactions.put(transactionNo, transaction);
}
history.add(action, transaction);
}
public History getHistory() {
    return history;
}
public static History create(String description, String actionsString) {
    HistoryBuilder builder = new HistoryBuilder(description);
    builder.addAll(actionsString);
    return builder.getHistory();
}
```

---

**HistoryItem.java**

```java
package ch.htwchur.mse.sem;
import java.sql.SQLException;

public class HistoryItem {
    private Action action;

    public Action getAction() {
        return action;
    }

    private Transaction transaction;

    public Transaction getTransaction() {
        return transaction;
    }

    public HistoryItem(Action action, Transaction transaction) {
        this.action = action;
        this.transaction = transaction;
    }

    public String execute() throws SQLException {
        return transaction.execute(action);
    }

    public boolean isTransactionAborted() {
        return transaction.isAborted();
    }

    public void setTransactionAborted(boolean aborted) {
        transaction.setAborted(aborted);
    }

    public String getDescription(OutputLanguage language) {
        String description;
        switch (language) {
            case STANDARD:
                description = action.getDescription(transaction);
                break;
            case SQL:
                description = action.getSQLDescription(transaction);
                break;
            default:
```
description=action.getDescription(transaction);
}  
break;
}
return description;
}

IsolationLevel.java

package ch.htwchur.mse.sem;
import java.sql.Connection;
public enum IsolationLevel {
  READ_UNCOMMITTED(Connection.TRANSACTION_READ_UNCOMMITTED),
  READ_COMMITTED(Connection.TRANSACTION_READ_COMMITTED),
  REPEATABLE_READ(Connection.TRANSACTION_REPEATABLE_READ),
  SERIALIZABLE(Connection.TRANSACTION_SERIALIZABLE);
  private int level;
  private IsolationLevel(int level) {
    this.level = level;
  }
  public int getLevel() {
    return level;
  }
}

OutputLanguage.java

package ch.htwchur.mse.sem;
public enum OutputLanguage {
  STANDARD,
  SQL
}

Simulation.java

package ch.htwchur.mse.sem;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import org.apache.commons.dbutils.DbUtils;
public class Simulation {
  public static String createString = "CREATE TABLE accounts (" + "id VARCHAR(1) NOT NULL," + "amount INT NOT NULL "+ ");
  private Database database;
  private Connection connection;
  private Statement statement;
  public Simulation(Database database) {
    this.database = database;
  }
  public void open() throws Exception {
    connection = database.getConnection();
    statement = connection.createStatement();
    dropTable();
    createTable();
    insert("x", 50);
    insert("y", 50);
    insert("z", 2);
```java
public void execute(History history, IsolationLevel level, OutputLanguage language)
    throws SQLException {
    System.out.println("Database: "+ database.getDescription());
    System.out.println("IsolationLevel: "+ level);
    System.out.println("History: "+ history.getDescription());
    System.out.println("History: "+ history.getItemDescription(OutputLanguage.STANDARD));
    System.out.println("-------------");
    selectAll();
    history.open(database, level);
    System.out.println("-------------");
    System.out.println("run:");
    history.execute(language);
    System.out.println("-------------");
    history.close();
    selectAll();
    System.out.println("-------------");
}

public void close() throws SQLException {
    dropTable();
    DbUtils.close(statement);
    DbUtils.close(connection);
}

public void run(History history, IsolationLevel level, OutputLanguage language) throws Exception {
    open();
    execute(history, level, language);
    close();
}

private void createTable() throws SQLException {
    statement.execute(createString);
}

private void dropTable() throws SQLException {
    database.dropTable("accounts");
}

private void insert(String id, int amount) throws SQLException {
    PreparedStatement preparedStatementInsert = connection
        .prepareStatement("INSERT INTO accounts(id, amount) VALUES(?,?)");
    preparedStatementInsert.setString(1, id);
    preparedStatementInsert.setInt(2, amount);
    preparedStatementInsert.executeUpdate();
    DbUtils.closeQuietly(preparedStatementInsert);
}

protected void selectAll() throws SQLException {
    String selectStatementSQL = "SELECT id,amount FROM accounts ORDER BY id";
    ResultSet selectResultSet = statement.executeQuery(selectStatementSQL);
    System.out.println("DB dump accounts:");
    System.out.println("id: amount");
    while (selectResultSet.next()) {
        System.out.println(selectResultSet.getString(1) + ": "+ selectResultSet.getInt(2));
    }
    selectResultSet.close();
    DbUtils.close(selectResultSet);
}
```

Transaction.java

```java
package ch.htw.chur.mse.sem;
import java.sql.*;

public class Transaction {
    private int id;
    public int getId() {
        return id;
    }

    private Connection connection;
    private boolean aborted;
```
public Transaction(int id) {
    this.id = id;
    this.connection = null;
    aborted = false;
}

public Connection getConnection() {
    return connection;
}

public void setConnection(Connection connection) {
    this.connection = connection;
}

public String execute(Action action) throws SQLException {
    return action.execute(this);
}

public boolean isAborted() {
    return aborted;
}

public void setAborted(boolean aborted) {
    this.aborted = aborted;
}

public void rollback() throws SQLException {
    connection.rollback();
}

public void setIsolationLevel(int transactionIsolation) throws SQLException {
    connection.setAutoCommit(false);
    connection.setTransactionIsolation(transactionIsolation);
}
6.3 Simulation of A1

```sql
Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ_UNCOMMITTED
History: A1
History: w1[x=10] r2[x] a1 c2
-----------
DB dump accounts:
  id: amount
  x: 50
  y: 50
  z: 2
-----------
run:
UPDATE accounts SET amount = 10 WHERE id=x
  SELECT amount FROM accounts WHERE id=x  --> 10
ROLLBACK;
COMMIT;
-----------
DB dump accounts:
  id: amount
  x: 50
  y: 50
  z: 2
-----------

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ_COMMITTED
History: A1
History: w1[x=10] r2[x] a1 c2
-----------
DB dump accounts:
  id: amount
  x: 50
  y: 50
  z: 2
-----------
run:
UPDATE accounts SET amount = 10 WHERE id=x
  SELECT amount FROM accounts WHERE id=x  --> A lock could not be obtained within the time requested
ROLLBACK;
-----------
DB dump accounts:
  id: amount
  x: 50
  y: 50
  z: 2
-----------
```
### 6.4 Simulation of H1

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ_UNCOMMITTED
History: H1
History: r1[x] w1[x=10] r2[x] r2[y] c2 r1[y] w1[y=90] c1

<table>
<thead>
<tr>
<th>id</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>50</td>
</tr>
<tr>
<td>y</td>
<td>50</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>

---
run:

```sql
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 10 WHERE id=x
SELECT amount FROM accounts WHERE id=x
SELECT amount FROM accounts WHERE id=y
SELECT amount FROM accounts WHERE id=y
COMMIT;
```

DB dump accounts:

<table>
<thead>
<tr>
<th>id</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>90</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>

---
run:

```sql
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 10 WHERE id=x
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 90 WHERE id=y
COMMIT;
```

DB dump accounts:

<table>
<thead>
<tr>
<th>id</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>90</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>

---

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ_COMMITTED
History: H1
History: r1[x] w1[x=10] r2[x] r2[y] c2 r1[y] w1[y=90] c1

<table>
<thead>
<tr>
<th>id</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>50</td>
</tr>
<tr>
<td>y</td>
<td>50</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>

---
run:

```sql
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 10 WHERE id=x
SELECT amount FROM accounts WHERE id=x
SELECT amount FROM accounts WHERE id=x
SELECT amount FROM accounts WHERE id=y
UPDATE accounts SET amount = 90 WHERE id=y
COMMIT;
```

A lock could not be obtained within the time requested

DB dump accounts:

<table>
<thead>
<tr>
<th>id</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
</tr>
<tr>
<td>y</td>
<td>90</td>
</tr>
<tr>
<td>z</td>
<td>2</td>
</tr>
</tbody>
</table>

---
### 6.5 Simulation of A2

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ COMMITTED
History: A2
History: r1[x] w2[x=10] c2 r1[x] c1

---

DB dump accounts:
id: amount
x: 50
y: 50
z: 2

---

run:
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 10 WHERE id=x
COMMIT;
SELECT amount FROM accounts WHERE id=x

---

DB dump accounts:
id: amount
x: 10
y: 50
z: 2

---

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: REPEATABLE_READ
History: A2
History: r1[x] w2[x=10] c2 r1[x] c1

---

DB dump accounts:
id: amount
x: 50
y: 50
z: 2

---

run:
SELECT amount FROM accounts WHERE id=x
UPDATE accounts SET amount = 10 WHERE id=x
A lock could not be obtained within the time requested

SELECT amount FROM accounts WHERE id=x
COMMIT;

---

DB dump accounts:
id: amount
x: 50
y: 50
z: 2
6.6 Simulation of H2

```
Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: READ_COMMITTED
History: H2
History: r1[x] r2[x] w2[x=10] r2[y] w2[y=90] c2 r1[y] c1

DB dump accounts:
id: amount
x: 50
y: 50
z: 2
---------
run:
SELECT amount FROM accounts WHERE id=x        -> 50
SELECT amount FROM accounts WHERE id=x        -> 50
UPDATE accounts SET amount = 10 WHERE id=x
SELECT amount FROM accounts WHERE id=y        -> 50
UPDATE accounts SET amount = 90 WHERE id=y
COMMIT;
SELECT amount FROM accounts WHERE id=y        -> 90
COMMIT;
---------
DB dump accounts:
id: amount
x: 10
y: 90
z: 2
---------

Database: Apache Derby 10.8.2.2 - (1181258)
IsolationLevel: REPEATABLE_READ
History: H2
History: r1[x] r2[x] w2[x=10] r2[y] w2[y=90] c2 r1[y] c1

DB dump accounts:
id: amount
x: 50
y: 50
z: 2
---------
run:
SELECT amount FROM accounts WHERE id=x        -> 50
SELECT amount FROM accounts WHERE id=x        -> 50
UPDATE accounts SET amount = 10 WHERE id=x
  -> A lock could not be obtained within the time requested
SELECT amount FROM accounts WHERE id=y        -> 50
COMMIT;
---------
DB dump accounts:
id: amount
x: 50
y: 50
z: 2
---------
```