

Otimização de Sistemas de Banco de Dados

Laboratório 04

Análise de Desempenho: Prepared Statements vs Concatenação

Objetivo do Laboratório

Reproduzir em laboratório a teoria apresentada em sala de aula quanto ao *parse* de comandos e custo gerado no servidor.

O laboratório consiste na implementação de um código-fonte acessando o banco de dados via um driver capaz de processar **prepared statements** (exemplo: Java e JDBC). O aluno deve elaborar duas versões do mesmo código, alternando o modo de acesso aos dados conforme explicado em aula (*hardcoded* e *softcoded*).

O processamento consiste em executar **5.000 vezes ou mais** um mesmo SELECT numa tabela criada pelo aluno. Para cada modo de acesso, o aluno deve computar o tempo gasto para processar o total dos comandos. Ao final, deve-se montar um gráfico comparativo entre as duas versões implementadas e demonstrar a diferença entre elas.

Análise dos Resultados

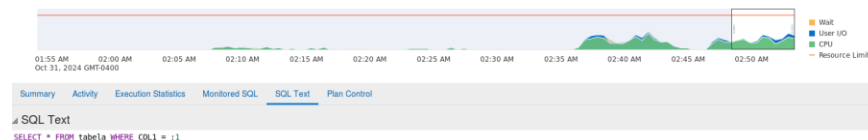
Os testes realizados mostraram um desempenho superior ao utilizar **Prepared Statements** em relação ao método de concatenação de strings. Os tempos de execução das consultas foram os seguintes:

- **Tempo de Execução com Hardcoded Query (Concatenação): 213573 ms**
- **Tempo de Execução com Prepared Statement: 187738 ms**

Interpretação dos Resultados

1. Otimização do Cache de Consultas

Quando um **Prepared Statement** é utilizado, o SGBD armazena o formato da consulta em cache, permitindo que a etapa de *parsing* da query seja pulada em execuções subsequentes. Apenas as variáveis das cláusulas WHERE são processadas, resultando em uma execução mais rápida.



Código utilizado (**Prepared Statement**):

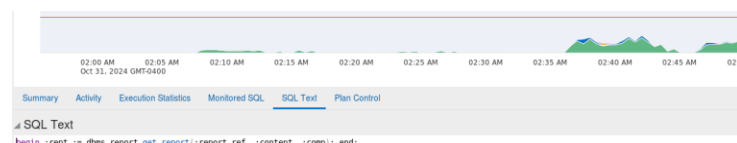
```
37 public long executePreparedStatementQuery() throws SQLException {
38     String query = "SELECT * FROM tabela WHERE COL1 = ?";
39
40     try (PreparedStatement preparedStatement = DatabaseConnection.getConnection().prepareStatement(query)) {
41
42         long startTime = System.currentTimeMillis();
43
44         // Loop para 5.000 execuções com valores diferentes
45         for (int i = 1; i <= 5000; i++) {
46             preparedStatement.setInt(1, i); // Define o valor atual de i no parâmetro
47             ResultSet resultSet = preparedStatement.executeQuery();
48
49             // Processa o resultado
50             while (resultSet.next()) {
51                 int col1 = resultSet.getInt("COL1");
52                 String col2 = resultSet.getString("COL2");
53                 System.out.println("PreparedST (Softcoded) = COL1: " + col1 + ", COL2: " + col2);
54             }
55             resultSet.close();
56         }
57
58         long endTime = System.currentTimeMillis();
59         return endTime - startTime;
60     }
61 }
62 }
```

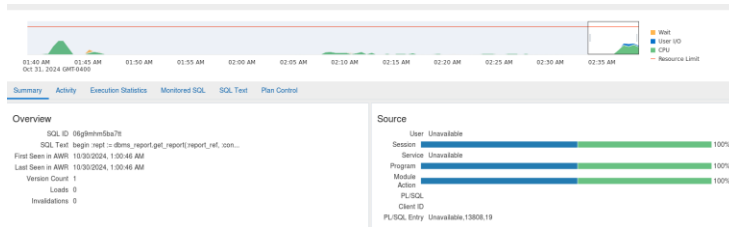
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

PreparedST (Softcoded) = COL1: 237, COL2: text01
PreparedST (Softcoded) = COL1: 238, COL2: text0238
PreparedST (Softcoded) = COL1: 238, COL2: text01
PreparedST (Softcoded) = COL1: 239, COL2: text0239
PreparedST (Softcoded) = COL1: 239, COL2: text01
PreparedST (Softcoded) = COL1: 240, COL2: text0240
PreparedST (Softcoded) = COL1: 240, COL2: text01
PreparedST (Softcoded) = COL1: 241, COL2: text0241
PreparedST (Softcoded) = COL1: 241, COL2: text01

2. Processamento de Consultas Concatenadas

No método de concatenação, o SGBD não consegue identificar o formato da consulta previamente, o que o obriga a realizar o *parsing* da query toda vez que ela é executada. Isso contribui para um aumento significativo no tempo de execução, como evidenciado nos resultados.





Código utilizado (concatenacao):

```
11 public long executeHardcodedQuery() throws SQLException {
12     String queryTemplate = "SELECT * FROM tabela WHERE COL1 = ?"; // template para concatenar o valor de COL1
13
14     try (Statement statement = DatabaseConnection.getConnection().createStatement()) {
15
16         long startTime = System.currentTimeMillis();
17
18         // Loop para 100.000 execuções com valores diferentes
19         for (int i = 1; i <= 100000; i++) {
20             String query = queryTemplate + i; // Monta a query com o valor atual de i
21             ResultSet resultSet = statement.executeQuery(query);
22
23             // Processa o resultado
24             while (resultSet.next()) {
25                 int col1 = resultSet.getInt("COL1");
26                 String col2 = resultSet.getString("COL2");
27                 System.out.println("Concat (Hardcoded) = COL1: " + col1 + ", COL2: " + col2);
28             }
29             resultSet.close();
30         }
31
32         long endTime = System.currentTimeMillis();
33         return endTime - startTime;
34     }
35 }
36
37 public static void main(String[] args) throws SQLException {
38     // ...
39     long result = new HardcodedQuery().executeHardcodedQuery();
40     // ...
41 }
```

3. Benefícios Adicionais do Prepared Statement:

Além da melhoria no desempenho, o uso de **Prepared Statements** também oferece um nível de segurança superior contra SQL injection, protegendo a aplicação contra ataques maliciosos.

4. Importância do prepared statement em Ambientes Corporativos

Em ambientes corporativos que utilizam o SGBD Oracle, a eficiência do **Prepared Statement** pode resultar em melhorias de desempenho de 100% ou mais, dependendo da complexidade da consulta. Embora todos os SGBDs se beneficiem dessa abordagem, a intensidade da melhoria varia.

Conclusão

Este teste simples, que envolveu apenas um parâmetro, demonstrou claramente a vantagem dos **Prepared Statements** em termos de desempenho e segurança. Com a

evidência coletada, é evidente que não há motivos para implementar consultas SQL sem o uso de **Prepared Statements**, preferindo sempre evitar a concatenação de strings.

- **API REST criada usando Spring Boot:**

```
13 @RestController
14 @RequestMapping("/test-performance")
15 public class PerformanceTestController {}
16
17 private final OracleQueryExecutor executor;
18
19 public PerformanceTestController() {
20     this.executor = new OracleQueryExecutor();
21 }
22
23 @GetMapping
24 public Map<String, Long> testPerformance() {
25
26     Map<String, Long> result = new HashMap<>();
27
28     try {
29         // Executa a query hardcoded e armazena o tempo de execução
30         long hardcodedTime = executor.executeHardcodedQuery();
31         result.put("hardcodedQueryTime", hardcodedTime);
32
33         // Executa a query com prepared statement e armazena o tempo de execução
34         long preparedStatementTime = executor.executePreparedStatementQuery();
35         result.put("preparedStatementQueryTime", preparedStatementTime);
36
37     } catch (SQLException e) {
38         e.printStackTrace();
39         result.put("error", -1L); // Indica erro
40     }
41
42     return result;
43 }
```

Retorno da API

Os resultados dos testes podem ser representados na seguinte estrutura JSON:

```
{
  "hardcodedQueryTime": 213573,
  "preparedStatementQueryTime": 187738
}
```