Performance Tuning for the JDBC™ API

What Works, What Doesn't, and Why.

Mark Chamness
Sr. Java Engineer
Cacheeware
Illustrate techniques for optimizing JDBC API-based calls from the Java platform, including examples for an Oracle database implementation.
Learning Objectives

As a result of this presentation, you will:

– Design better JDBC implementations
– Recognize potential performance bottlenecks
– Make more money
Speaker’s Qualifications

- Professional Java developer since 1995, including 3 years at Sun Microsystems.
- Implemented JDBC for production systems using the major databases: Sybase, Informix, and Oracle.
- Worked on more than 3,000 Java bugs at Sun Microsystems.
1) Why optimize?
2) Basic API techniques.
3) Design Strategies.
4) Advanced Driver Tuning methods.
Why Optimize?

- On average, a web request performs 4 database queries.
- Experience has shown that database calls are typical performance bottleneck.
- Bad JDBC can overwhelm the database.
JDBC API

Most Versatile

- SQL: "SELECT * FROM TABLE"
- java.sql.PreparedStatement
- java.sql.CallableStatement
- Cache data on client.

Most Optimized
JDBC API

SQL Statements

- Most flexible
- Least reliable
- Must be recompiled in database for each use
PreparedStatement

- Represents a precompiled SQL statement
- Can be used to efficiently execute statement multiple times
- Somewhat flexible – can create new ones as needed
PreparedStatement pstmt =
    con.prepareStatement("UPDATE EMPLOYEES
    SET SALARY = ? WHERE ID = ?");

pstmt.setBigDecimal(1, 153833.00);
pstmt.setInt(2, 110592);
pstmt.executeUpdate();
java.sql.CallableStatement

- Used to execute SQL stored procedures.
- Same syntax as PreparedStatement.
- Least flexible.
- Most optimized DB call.
Cache

• Keep data within client to reduce the number of round-trips to the database.

• Lesson: The less JDBC the better.
Basic Design Techniques

Use Database Connection Pool

- Don’t use DriverManager.getConnection() often. JDBC connections can take 0.5 to 2 seconds to create.
- Create Pool of Connections and reuse them.
- Necessity for any production system.
Basic Design Techniques

Use multi-threading with Connection Pooling to address network latency:

• Threads can issue queries over separate database connections.
• This improves performance to a point.
Basic Design Techniques

Single-batch Transactions

Collect set of operations and submit transaction in one statement:

BEGIN TRANSACTION
  UPDATE TABLE1...
  INSERT INTO TABLE2...
  DELETE TABLE3
COMMIT
Single-batch Transactions

• DB obtains necessary locks on rows and tables, uses and releases them in one step

• Depending on transaction type, separate statements and commits can result in more DB calls and hold DB locks longer
Basic Design Techniques

Single-batch Transaction Types
Significantly different effects!

java.sql.Connection

• TRANSACTION_READ_COMMITTED
• TRANSACTION_READ_UNCOMMITTED
• TRANSACTION_REPEATABLE_READ
• TRANSACTION_SERIALIZABLE
Don’t have transaction span user input

- Application sends BEGIN TRAN and SQL, locking rows or tables for update
- Application waits for user to press key before committing transaction
Solution: Optimistic locking

- Optimistic locking employs timestamps and triggers in queries and updates
- Queries select data with timestamp values
- Prepare a transaction based on that data, without locking data in a transaction
Basic Design Techniques

Smart Queries

- Make queries as specific as possible
- Put more logic into SQL statements
- DB are designed to use SQL efficiently
- Proper use of SQL can avoid performance problems
Basic Design Techniques

Smart Query Ex: get employees in ENG dept

Instead of:
SELECT * FROM employees;
SELECT * FROM dept;
(and joining on Java application side)

Use database join:
SELECT employees.* FROM employees E, dept D WHERE E.DEPTNO = D.DEPTNO AND D.DEPTTYPE = 'ENG';
Basic Design Techniques

Smart Queries

• Minimize ResultSet before crossing network
• Many performance problems come from moving raw data around needlessly
Basic Design Techniques

Smart Query Guidelines

- Use DB for filtering
- Use Java for business logic
- DB does filtering very well
- DB business logic is poor
- (At least very inconsistent between database vendors.)
Basic Design Techniques

Keep operational data set small as possible

• Move non-current data to other tables and do joins for rarer historical queries
• Otherwise, index and cluster so frequently used data is logically and physically localized
Advanced Driver Tuning

- Special options for each JDBC driver
- No common standard
- Improve performance by reducing round trips to the database.
- Ex. Oracle driver performance extensions
Advanced Driver Tuning

Oracle Performance Extensions

1) Row Prefetch
2) Batch Updates
3) Suppress DatabaseMetaData
   TABLE_REMARKS Columns
4) Specify Column Types
1. Row Prefetch

- Use client-side buffers
- Replace round trips by local manipulation of rows returned by query
- Use `OracleStatement.setRowPrefetch()`
2. Batch Updates

• Reverse Prefetch

• Does for data going to DB what prefetch does for data coming from it

• OraclePreparedStatement.executeUpdateBatch
2. Batch Updates

• Standard JDBC makes a trip to DB for each PreparedStatement.executeUpdate

• Batching: When number of queued requests reaches batch size, sends them to DB
Advanced Driver Tuning

Suppress DatabaseMetaData
TABLE_REMARKS Columns

• Avoids expensive outer join operation
• DatabaseMetaData.getTables() &
getColumns() are slow
• OracleConnection.setRemarksReporting()
4. Specify Column Types

- Standard JDBC: 1st trip to DB used to determine column types of ResultSet
- Then converts data to requested return type (if necessary)
4. Specify Column Types

- Specify column types, JDBC makes one fewer trip to DB
- The server performs any necessary type conversions (which is optimized)
4. Specify Column Types

Must specify data type for each column of expected ResultSet

1. `OracleStatement.clearDefines()`
2. `OracleStatement.defineColumnType(..)`
3. `OracleStatement.executeQuery()`
Summary

Optimization Stages

1) Leverage the strengths of the DB
2) Use the full range of java.sql API
3) Design for Performance – Connection Pools, Multi-Threading, etc.
4) Implement driver performance extensions