**Data Handling Best Practices and Tips for Better Java Application Performance**

Many applications end up in performance problem, when those are pushed through testing phase or are in production. If we look at the causes behind performance problems, there are a set of problems that are caused by inferior data handling. These appplication require these data handling best practices. Data handling is very critical in case of applications processing huge amount of data. Here are few practical data handling best practices or tips for better Java application performance.

## **Reduce Amount of Data Transferred**

In any java application, method call is made to either get something done on method side, or get data processing done by passing input data. Both operations involve data exchange between the caller and processor methods. There can be a rule of thumb that the amount of data that is transferred to and from a method call is minimized. Lesser data provides many benefits e.g. reduced amount of processing, lesser objects to clean up, smaller memory footprint and so on. The program design should target to reduce the amount of data getting transferred across methods, layers, applications and even organizations. This can be achieved by exploring possibility of data source side processing explained below.

## **Lazy Loading**

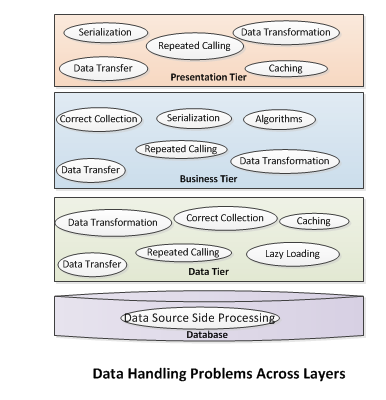
This is delaying data fetching from data store until the last moment when data is actually needed. It is very beneficial in case of heavy objects. E.g. there is a file entity in database which has file contents as blob attribute along with other file characteristics. This blob can be from few Kilobytes to hundreds of Megabytes. Middle tier logic is dependent on attributes other than this heavy blob attributes until it is finally showing the file contents. Lazy loading can be used to delay loading of blob attributes.

## **Repeated/Duplicate Data Calls**

Repeated calling to data provider to get data can hamper performance considerably when it is a remote call of certain type, e.g. database call, web service call, or any call involving marshaling and un-marshaling of data. Use Façade pattern instead; pull all required data in one call and minimize the connection cost and data transfer over network cost.

## **Caching**

Frequently used but less frequently changing data can be cached. Mostly it is static/master data that is cached. Now a days, applications requiring very high performance use transaction data caching also. But as a simple rule, while designing the application, identify such entities that will be frequently required but are not going to change so frequently or not at all going to change, and cache those at appropriate location. Refresh logic can also be employed for such entities.



## **Data Source Side Data Processing**

Good to process data at its source or storage location itself. Transferring large amount of data to the client, and then processing it involves cost of transfer, and sometimes client specific processing logic implementation in changed format. E.g. Filtering certain amount of data in oracle can be as simple as adding a where clause to the select query, while getting all the records and filtering those in java program will require row by row traversing, and individual attribute comparison. This Java implementation may not be very optimal.

## **Minimum Data Transformation and Conversion/ Incorrect Data Types**

Converting data from one format to other may involve cost of conversion. For a single value the performance hit due to conversion may be tiny, but if we combine it for thousands of records, it becomes considerable. E.g. selecting value in String data type and converting it to double or primitive and vice versa.

## **Correct Collection to Hold Data**

It is a very important consideration from performance point of view. Java has provided different collections to fulfill different needs, e.g. there are few kind of raw collections e.g. ArrayList, where you can go on adding data without the collection doing any intelligent operation on the data in collection. Example is Vector, which is synchronized. It is good to ask this question, do I need synchronized data insertion in my data collection? If not, then don’t go for Vector, try ArrayList or any other collection depending on other parameters.

## **Sub Optimal Data Processing Algorithms**

Sometimes it is the data processing algorithm or logic implementation which is having performance problems. This logic implementation should have performance as one of the goals along with other goals, such as memory consumption. Few things that make algorithms sub optimal are –

* ·         Expensive calls, database calls from loops
* ·         Object declaration inside loops
* ·         Unnecessary nested loops
* ·         Objects storage in multiple collections

## **Finalize Heavy Data Objects**

We cannot guaranty immediate garbage collection, but finalizing or setting null is a good practice.

## Utilize Technology Features for Optimal Data Processing

Many technologies have come up with features to help in data processing. This includes ready features of implementation of some of above points. Below are few examples –

–          Prepared statements support from Oracle

–          Framework caching e.g. Hibernate primary and secondary cache

–          Lazy loading support by Hibernate framework

## **Data Serialization and Deserialization**

Design to avoid these expensive operations. If not then it is good to minimize the amount of data that is transformed. E.g. In case of web applications, minimize amount of data being put in session.

## **Parallel Data Processing**

If there is need for huge amount of data processing, then unrelated data can be processed in parallel. This gives reduced total processing time.

## **Object Reuse**

Instead of creating heavy objects from scratch, one can create clone of existing objects and modify only the required attributes reusing whatever possible. Amount of reuse can be controlled by shallow or deep cloning.

JDBC Handling Huge ResultSet

In this article lets discuss about querying huge database tables using JDBC api.

Consider a scenario where we have a huge database table with (say 1 million records), and the requirement is to display those records in your application UI.

Generally, we will use the JDBC api in java to query the database table, iterate the ResultSet object and display the results in UI.

But for querying DB tables with huge set of records it will take more time to iterate the records from DB. This is because the default fetch size for a result set is usually 10. It will be fine if the number of records in the DB is less (say 500). For more records it will take more time to retrieve.

One simple solution is to set the fetchSize attribute in ResultSet to an appropriate value.

For example, to query 1 million records, we can set the fetchSize as 100.

You can easily observe the difference by executing the following program in two ways with setting fetch Size and without setting the fetchSize in ResultSet.

**Output:**

Case 1: Without Setting Fetch Size   
1280653337037  
Time to iterate ResultSet -> 1283  
Case 2: With Setting Fetch Size value ->100  
1280653338321  
Time to iterate ResultSet -> 143

It’s clear that when the fetch size is more, records are retrieved more quickly.  
At the same time setting the fetchSize to a larger value will increase the memory and at sometimes cause out of memory errors. So, while setting fetch Size test your code before using it in the production environment.

# How can I handle extremely large amounts of data coming from a database in J2EE development?

Ok lets assume we have millions of records in a telephone directory and our task is to fetch these records and display it using a J2EE application

1. Simplest way of doing it is a normal fetch operation as a whole (millions of records ) using SQL or ORACLE  in J2EE ,But the problem here is These records  will be cached in memory where your java application might behave so weird ,In here you should not store the data in Java's memory rather write the data fetched to response immediately as the data comes in .As [Senthil Ramasamy](https://www.quora.com/profile/Senthil-Ramasamy-2) pointed out ,You can use pagination to show these records .
2. The second approach would be setting a fetch size using Statement#FetchSize() and limiting the data fetched by result statement. Where you answered it right !

## JDBC performance tips

Java performance tuning tips relevent for tuning JDBC usage. The top tips are:

* Use prepared statements. Use parametrized SQL.
* Tune the SQL to minimize the data returned (e.g. not 'SELECT \*').
* Minimize transaction conflicts (e.g. locked rows).
* Use connection pooling.
* Try to combine queries and batch updates.
* Use stored procedures.
* Cache data to avoid repeated queries.
* Close resources (Connections, Statements, ResultSets) when finished with.
* Select the fastest JDBC driver.

he following detailed tips have been extracted from the [raw tips](http://www.javaperformancetuning.com/tips/rawtips.shtml) page

<http://www.onjava.com/pub/a/onjava/2001/12/05/optimization.html>  
Measuring JDBC performance (Page last updated December 2001, Added 2001-12-26, Author Jack Shirazi, Publisher OnJava). Tips:

* Effectively profiling distributed applications can be difficult. I/O can show up as significant in profiling, simply because of the nature of a distributed application.
* It can be unclear whether threads blocking on reads and writes are part of a significant bottleneck or simply a side issue.
* When profiling, it is usually worthwhile to have separate measurements available for the communication subsystems.
* Wrapping the JDBC classes provides an effective technique for measuring database calls.
* [Article discusses how to create JDBC wrapers to measure the performance of database calls].
* If more than a few rows of a query are being read, then the ResultSet.next() method can spend a significant amount of time fetching rows from the database, and this time should be included in measurements of database access.
* JDBC wrappers are simple and robust, and require very little alteration to the application using them (i.e, are low maintenance), so they are suitable to be retained within a deployed application.

<http://www.as400.ibm.com/developer/java/topics/jdbctips.html>  
JDBC Performance Tips (targeted at AS/400, but generically applicable) (Page last updated February 2001, Added 2001-03-21, Authors Richard Dettinger and Mark Megerian, Publisher IBM). Tips:

* Move to the latest releases of Java as they become available.
* Use prepared statements (PreparedStatement class) [article provides coded example of using Statement vs. PreparedStatement].
* Note that two database calls are made for each row in a ResultSet: one to describe the column, the second to tell the db where to put the data. PreparedStatements make the description calls at construction time, Statements make them on every execution.
* Avoid retrieving unnecessary columns: don't use "SELECT \*".
* If you are not using stored procedures or triggers, turn off autocommit. All transaction levels operate faster with autocommit turned off, and doing this means you must code commits. Coding commits while leaving autocommit on will result in extra commits being done for every db operation.
* Use the appropriate transaction level. Increasing performance costs for transaction levels are: TRANSACTION\_NONE; TRANSACTION\_READ\_UNCOMMITTED; TRANSACTION\_READ\_COMMITTED; TRANSACTION\_REPEATABLE\_READ; TRANSACTION\_SERIALIZABLE. Note that TRANSACTION\_NONE, with autocommit set to true gives access to triggers, stored procedures, and large object columns.
* Store string and char data as Unicode (two-byte characters) in the database.
* Avoid expensive database query functions such as: getBestRowIdentifier; getColumns; getCrossReference; getExportedKeys; getImportedKeys; getPrimaryKeys; getTables; getVersionColumns.
* Use connection pooling, either explicitly with your own implementation, or implicitly via a product that supports connection pooling.
* Use blocked fetchs (fetching table data in blocks), and tailor the block size to reduce calls to the database, according to the amount of data required.
* Use batch updates (sending multiple rows to the database in one call).
* Use stored procedures where appropriate. These benefit by reducing JDBC complexity, are faster as they use static SQL, and move execution to the server and potentially reduce network trips.
* Use the type-correct get() method, rather than getObject().

<http://www.sys-con.com/java/article.cfm?id=671>  
J2EE Application server performance (Page last updated April 2001, Added 2001-04-20, Author Misha Davidson, Publisher Java Developers Journal). Tips:

* Avoid n-way database joins: every join has a multiplicative effect on the amount of work the database has to do. The performance degradation may not be noticeable until large datasets are involved.
* Avoid bringing back thousands of rows of data: this can use a disproportionate amount of resources.
* Cache data when reuse is likely.
* Use optimistic transactions: write to the database while checking that new data is not be overwritten by using WHERE clauses containing the old data. However note that optimistic transactions can lead to worse performance if many transactions fail.
* For read-only queries involving large amounts of data, avoid EJB objects and use JavaBeans as an intermediary to access manipulate and store the data for JSP access.

<http://www.bastie.de/resource/res/mjp.pdf> and <http://www.bastie.de/java/mjperformance/contents.html>  
Performance tuning report in German. Thanks to Peter Kofler for extracting the tips. (Page last updated November 2001, Added 2001-07-20, Author Sebastian Ritter, Publisher Ritter). Tips:

* use the right database driver
* use direct JDBC drivers
* cache (SQL) Statements for DB access
* use PreparedStatements for DB access

<http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf>  
Paper detailing the "Best Practices for Developing High Performance Web and Enterprise Applications" using IBM's WebSphere. All the tips are generally applicable to servlet/EJB development, as well as other types of server development. (Page last updated September 2000, Added 2001-01-19, Author Harvey W. Gunther, Publisher IBM). Tips:

* Use JDBC connection pooling, release JDBC resources when done, and reuse datasources for JDBC connections.

<http://www-4.ibm.com/software/webservers/appserv/3steps_perf_tuning.pdf>  
Tuning IBM's WebSphere product. White paper: "Methodology for Production Performance Tuning". Only non-product specific Java tips have been extracted here. (Page last updated September 2000, Added 2001-01-19, Author Gennaro (Jerry) Cuomo, Publisher IBM). Tips:

* Use connection pools and cached prepared statements for database access.

<http://www.redbooks.ibm.com/abstracts/sg245657.html>  
WebSphere V3 Performance Tuning Guide (Page last updated March 2000, Added 2001-01-19, Authors Ken Ueno, Tom Alcott, Jeff Carlson, Andrew Dunshea, Hajo Kitzh?fer, Yuko Hayakawa, Frank Mogus, Colin D. Wordsworth, Publisher IBM). Tips:

* Run an application server and any database servers on separate server machines.
* One of the most time-consuming procedures of a database application is establishing a connection to the database. Use connection pooling to minimize this overhead.

<http://www.weblogic.com/docs51/techdeploy/jdbcperf.html>  
Weblogic JDBC tuning (Page last updated April 1999, Added 2001-03-21, Author BEA Systems, Publisher BEA). Tips:

* Use connection pools to the database and reuse connections rather than repeatedly opening and closing connections. Optimal pool size is when the connection pool is just large enough to service requests without waits.
* Cache frequently requested data in the JVM and avoid the unnecessary database requests.
* Avoid accessing the database wherever possible.
* Fetch rows in batches rather than one at a time, using the batch as a read-ahead mechanism (i.e. pre-fetch rows in batches). Tune the batch size and the number of rows pre-fetched. Avoid pre-fetching BLOBs.
* Avoid moving data unless absolutely necessary. Process the data and produce results as close to its source as possible. Use stored procedures.
* Streamline data before the result crosses the network.
* Use stored procedures to avoid extra network transfers.
* Use built-in DBMS set-based processing to operate on multiple rows/tables in one request.
* Avoid row at a time processing, process multiple rows together wherever possible.
* Counting entries in a table (e.g. using SELECT count(\*) from myTable, yourTable where ... ) is resource intensive. Try first selecting into temporary tables, returning only the count, and then sending a refined second query to return only a subset of the rows in the temporary table.
* Proper use of SQL can reduce resource requirements. Use queries which return the minimum of data needed: avoid SELECT \* queries. A complex query that returns a small subset of data is more efficient than a simple query that returns more data than is needed.
* Make your queries as smart as possible, i.e. as precise as possible to minimize the data transferred to just that subset that is required.
* Try to batch updates: collect statements together and execute them together in one transaction. Use conditional logic and temporary variables if necessary to achieve statement batching.
* Never let a DBMS transaction span user input.
* Consider using optimistic locking. Optimistic locking employs timestamps to verify that data has not been changed by another user, otherwise the transaction fails.
* Use in-place updates, i.e. change data in rows/tables that already exist rather than adding or deleting rows/tables. Try to avoid moving rows or changing their sizes.
* Store operational data and historic data separately (or more generally store frequently used data separately from infrequently used data).
* Keep your operational data set as small as possible, to avoid having to read through data that is irrelevant.
* DBMSs work well with parallelism. Try to design the application to do other things while interacting with the DBMS.
* Use pipelining and parallelism. Designing applications to support lots of parallel processes working on easily distinguished subsets of the work makes the application faster. If there are multiple steps to processing, try to design your application so that subsequent steps can start working on the portion of data that any prior process has finished, instead of having to wait until the prior process is complete.
* Choose the right driver for your application, i.e. the fastest JDBC driver.

<http://www.sys-con.com/websphere/article.cfm?id=40>  
JDBC optimizing for DB2 (Page last updated April 2002, Added 2002-04-26, Author John Goodson, Publisher WebSphere Developers Journal). Tips:

* Use the same connection to execute multiple statements.
* Keep connection objects open, and reuse them, rather than repeatedly connecting and disconnecting.
* Turn off autocommit, but don't leave transactions open for too long.
* Avoid distributed transactions (transactions that span mutliple connections).
* Minimize the data retrieved from the database, both columns and rows. Use setMaxRows, setMaxFieldSize, and SetFetchSize.
* Use the most efficiently handled data type: character strings are faster than integers, which are in turn more efficient than floating-point and timestamps.
* Use programmatic updates: updateXXX() calls on updatable resultsets. The resultset is already postioned at a row, so eliminating the usual overhead of finding the row to be updated when using an UPDATE statement.
* Cache any required metadata and use metadata methods as rarely as possible as they are quite slow.
* Avoid using null parameters in metadata queries.
* Use a dummy query to get the metadata for a column, rather than use the getcolumns()
* Use parameter markers with stored procedures, rather than embedding data literally in the statement, to minimize parsing overheads.
* Use prepared statements for repeatedly executing SQL statements
* Choose the optimal cursor: forward-only for sequential reads; insensitive for two-way scrolling. Avoid insenstive cursors for queries that only return one row.

<http://www.sys-con.com/java/article.cfm?id=1171>  
J2EE Performance tuning (Page last updated October 2001, Added 2001-10-22, Author James McGovern, Publisher Java Developers Journal). Tips:

* If only using an entity bean for data access, use JDBC directly instead.
* Use a dedicated remote object to generate unique primary keys.
* Follow standard JDBC optimizations: use connection pools; prefer stored procedures or direct SQL; use type 4 drivers; remove extra columns from the result set; use prepared statements when practical; have your DBA tune the query; choose the appropriate transaction levels.
* Consider storing all database character data in Unicode to eliminate conversion overheads. But beware: this step will cause your database size to grow, as Unicode requires 2 bytes per character.
* Use block fetches when the query will give a large ResultSet and all rows are needed. Use the Page-by-Page Iterator pattern when only some of the rows may be needed.
* Consider using an in-memory database (product) for data that doesn't need to be persisted.

<http://portals.devx.com/datadirect/Article/6338>  
JDBC Drivers (Page last updated March 2002, Added 2002-04-26, Author Barrie Sosinsky, Publisher DevX). Tips:

* Type 1 drivers are JDBC-ODBC bridges, plus an ODBC driver. Recommended only for prototyping, not for production. Not suitable for high-transaction environments. Not well supported, and limited in functionality.
* Type 2 drivers use a native API, and are part-Java drivers. Have a binary-code client loading overhead, and may not be fully-featured.
* Type 3 drivers are a pure Java driver which connects to database middleware. Can be server-based which is frequently faster than types 1 and 2.
* Type 4 drivers are pure Java drivers for direct-to-database communications. This can minimize overheads, and generally provides the fastest driver.
* JDBC 3.0 has additional features to improve performance such as advancements in connection pooling, statement pooling, RowSet objects.
* Opening a connection is the most resource-expensive step in database transactions. Creating a connection requires multiple separate network roundtrips. However, once the connection object has been created, there is little penalty in leaving the connection object in place and reusing it for future connections.
* Connection pooling, keeps open a cache of database connection objects, making them available for immediate use. Instead of performing expensive network roundtrips to the database server to open a connection, a connection attempt results in the re-assignment of a connection from the local cache.
* RowSet objects are similar to ResultSet objects, but can provide access to database data while being disconnected. This allows data to be efficiently cached in its simplest form.
* Prepared statement pooling (available from JDBC 3.0) caches SQL queries that have been previously optimized and run so that, should they be needed again, they do not have to go through optimization pre-processing again (avoiding optimization steps, such as checking syntax, validating addresses, and optimizing access paths and execution plans). Statement pooling can be a significant performance booster.
* Statement pooling and connection pooling in JDBC 3.0 can cooperate to share statement pools, so that connections that can use a cached statement from another connection, thus incurring statement preparation overheads only once on the first execution of some SQL by any connection.
* Database drivers developed by vendors other than the the database vendor can be better performing and more feature full. (Driver vendors concentrate on the driver, database vendors have many other things to consider).
* Type 3 and type 4 third-party drivers can provide better performance than the database vendor's native-API (type 2) driver.
* Try to use a driver that supports JDBC 3.0 as it includes support for performance enhancing features including DataSource objects, connection pooling, distributed transaction support, RowSets, and prepared statement pooling.
* Type 3 and Type 4 drivers are the drivers to use when performance is important.

<http://win-www.uia.ac.be/~s985218/professional/thesis/archief/documenten/Marktoverzicht.doc>  
Overview of common application servers. (Announced at <http://www.theserverside.com/home/thread.jsp?thread_id=9581>). I've extracted the performance related features (Page last updated October 2001, Added 2001-10-22, Author Pieter Van Gorp, Publisher Van Gorp). Tips:

* Connection pooling.

<http://www.onjava.com/pub/a/onjava/2001/11/07/atomic.html>  
Atomic File Transactions. (Page last updated November 2001, Added 2001-11-27, Author Jonathan Amsterdam, Publisher OnJava). Tips:

* If you don't require powerful search capabilities, using flat files may be faster than dealing with a database.

<http://www.sys-con.com/java/article.cfm?id=1268>  
EJB design (Page last updated January 2002, Added 2002-01-25, Author Boris Lublinsky, Publisher Java Developers Journal). Tips:

* Entity beans should not be simply mapped to database tables. Treating entity beans as such fine-grained objects which are effectively wrappers on table rows leads to increased network communications and heavier database communications than if entity beans are treated as coarse-grained components.
* For optimal performance, entity beans should be designed to: have large granularity, which usually means they should contain multiple Java classes and support multiple database tables; be associated with a certain amount of persistent data, typically multiple database tables, one of which should define the primary key for the whole bean; support meaningful business methods and encapsulate business rules to access the data.
* Entity beans are transactional resources due to their stateful nature, but application server vendors often rely on the underlying database to lock and resolve access appropriately. Although this approach greatly improves performance, it provides the potential for database lockup.

<http://www.jguru.com/jguru/faq/view.jsp?EID=131579>  
Discussion on JDBC performance (Page last updated August 2000, Added 2001-02-21, Author , Publisher JGuru). Tips:

* Use a connection pool mechanism whenever possible.
* Use prepared statements.
* Use stored procedures.
* Select only required columns rather than using select \* from Table xyz.
* Always close Statement and ResultSet objects as soon as possible.
* Work with DatabaseMetaData to get information about database functionality.
* Always catch *and* handle database warnings and exceptions.
* Time DB queries.
* Use the most appropriate datatype specific kinds of data, e.g. store dates as a date type rather than varchar.
* Use scrollable ResultSet (JDBC 2.0).
* Stay away from the JDBC-ODBC and other Type 1 drivers where possible.

<http://www.theserverside.com/resources/article.jsp?l=Building-Sclable-Recoverable-Applications>  
Scalable recoverable applications (Page last updated May 2002, Added 2002-07-24, Author Billy Newport, Publisher The Server Side). Tips:

* Oracle 9i includes queryable snapshots of the main database which can offload the query to run against the clients local snapshot.
* An in-memory database (such as TimesTen) is very, very fast and can act as a queryable cache for a back end database.
* Database instances on each machine, with replication increases reliability and access speed. But updates now need to be handled differently. Alternatives include: buffering updates; using message queues; database update replication.

<http://www.onjava.com/pub/a/onjava/2001/12/19/oraclejdbc.html>  
Oracle JDBC tips (Page last updated December 2001, Added 2001-12-26, Author Donald Bales, Publisher OnJava). Tips:

* Although Oracle recommend using the OCI driver for optimal client side access, the writer finds the Thin driver to have have better performance.
* Turn off autocommit, Connection.setAutoCommit(false).
* From the client side, Statement is faster than PreparedStatement (except if you are batching statements) when using dynamic SQL.
* Use PreparedStatements for all, except dynamic, SQL statements.
* Use PreparedStatements for batching repetitive inserts or updates.
* OraclePreparedStatement.setExecuteBatch() (proprietary method) is the fastest way to execute batch statements.
* Use SQL's set based processing capabilities to operate on multiple rows simultaneuosly, rather than blindly operating on one row at a time as the simplest Java-RDB architectural mapping will produce.

<http://www.oreilly.com/catalog/jorajdbc/chapter/ch19.html>  
Chapter 19, "Performance" of Java Programming with Oracle JDBC (Page last updated December 2001, Added 2001-12-26, Author Donald Bales, Publisher O'Reilly). Tips:

* Performance should be considered at the start of a project.
* Use the EXPLAIN PLAN facility to explain how the database's optimizer plans to execute your SQL statements, to identify performance improvements such as additional indexes.
* If more than one SQL statement is executed by your program, you can gain a small performance increase by turning off auto-commit.
* It takes about 65 iterations of a prepared statement before its total time for execution catches up with a statement, because of prepared statement initialization overheads.
* Use PreparedStatements to batch statements for optimal performance.
* The Thin driver is faster than the OCI driver. This is contrary to Oracle's recommendation.
* A SELECT statement makes two round trips to the database, the first for metadata, the second for data. Use OracleStatement.defineColumnType() to predefine the SELECT statement, thus providing the JDBC driver with the column metadata which then doesn't require the first database trip.
* Given a simple SQL statement and a stored procedure call that accomplishes the same task, the simple SQL statement will always execute faster because the stored procedure executes the same SQL statement but also has the overhead of the procedure call itself. On the other hand complex tasks requiring several SQL statements can be faster using stored procedures as fewer network trips and data transfers will be needed.

<http://www.fawcette.com/javapro/2002_01/magazine/columns/weblication/>  
Database performance (Page last updated December 2001, Added 2001-12-26, Author Peter Varhol, Publisher JavaPro). Tips:

* Thoughtful page design makes for a better user experience by enabling the application to seem faster than it really is.
* Use the flush method associated with the out object to display static text and graphics on the browser page before the database query returns, to prevent the user from having to look at a blank page for a long time.
* ResultSet types affect updates. TYPE\_FORWARD\_ONLY: no updating allowed; TYPE\_SCROLL-SENSITIVE: update immediately; TYPE\_SCROLL\_INSENSITIVE: update when the connection is closed. (Concurrency type must be set to CONCUR-UPDATABLE to allow the table to be updated.)
* Performance can be better if changes to the database are batched: turn off autocommit; add multiple SQL statements using the Statement.addBatch() method; execute Statement.executeBatch().
* Scaled systems need optimized SQL calls, querying the right amount of data, and displaying pages before the query is complete.
* Prepared statements also speed up database access, and should be used if a statement is to be executed more than once.

<http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/56A6275393F0BE5786256AFD004DBBB4?OpenDocument>  
JDBC tutorial (requires free registration) (Page last updated November 2001, Added 2001-12-26, Author Robert J. Brunner, Publisher IBM). Tips:

* Type 1 (JDBC-ODBC-DB) drivers incur a performance penalty because of the bridging needed to reach the database.
* [Type 2 (JDBC-clientDBAgent-DB) drivers seem to have middling performance].
* Type 3 (JDBC-Middleware-DB) drivers incur a performance penalty because of the bridging needed to reach the database, but does introduce optimization potential from the location of the middleware.
* Type 4 (JDBC-DB) drivers typically provide optimum driver performance.
* The higher the level of transaction protection, the higher the performance penalty. Transaction levels in order of increasing level are: TRANSACTION\_NONE, TRANSACTION\_READ\_UNCOMMITTED, TRANSACTION\_READ\_COMMITTED, TRANSACTION\_REPEATABLE\_READ, TRANSACTION\_SERIALIZABLE. Use Connection.setTransactionIsolation() to set the desired tansaction level.
* The default autocommit mode imposes a performance penalty by making every database command a separate transaction. Turn off autocommit (Connection.setAutoCommit(false)), and explicitly specify transactions.
* Batch operations by combining them in one transaction, and in one statement using Statement.addBatch() and Statement.executeBatch().
* Savepoints (from JDBC3.0) require expensive resources. Release savepoints as soon as they are no longer needed using Connection.releaseSavepoint().
* Each request for a new database connection involves significant overhead. This can impact performance if obtaining new connections occurs frequently. Reuse connections from connection pools to limit the cost of creating connections. [The tutorial lists all the overheads involved in creating a database connection].
* The ConnectionPoolDataSource (from JDBC3.0) and PooledConnection interfaces provide built-in support for connection pools.
* Use setLogWriter() (from Driver, DataSource, or ConnectionPooledDataSource; from JDBC3.0) to help trace JDBC flow.
* Use Connection.setReadOnly(true) to optimize read-only database interactions.
* Use Connection.nativeSQL() to see how the SQL query will execute in the database to help ensure that the SQL is optimized.

<http://www-105.ibm.com/developerworks/education.nsf/java-onlinecourse-bytitle/975BFD2C367CFFD686256B0500581B3B?OpenDocument>  
Advanced JDBC tutorial (requires free registration). (Page last updated November 2001, Added 2001-12-26, Author Robert J. Brunner, Publisher IBM). Tips:

* PreparedStatement objects are compiled (prepared) by the JDBC driver or database for faster performance, and accept input parameters so they can be reused with different data.
* Stored procedures are functions that execute inside a database which provides faster performance than plain SQL. Java supports stored procedures from CallableStatement objects.

<http://www.javaworld.com/jw-06-1998/jw-06-object-pool.html>  
Article on building an object pool for improved performance. (Page last updated June 1998, Added 2000-12-20, Author Thomas E. Davis, Publisher JavaWorld). Tips:

* Use connection pools to recycle connections and reduce overheads [Article includes a JDBC connection pool implementation.]

<http://www.sys-con.com/java/article.cfm?id=725>  
Optimizing JDBC (Page last updated August 2001, Added 2001-08-20, Author John Goodson, Publisher Java Developers Journal). Tips:

* Minimize the use of Metadata: Cache all metadata as they will not change; Avoid using null arguments in metadata methods; Use a dummy query with getMetadata() rather than getColumns().
* Retrieve data as efficiently as possible: Minimize the amount of data returned by the query; Don't make average users pay the same query cost of the users with extensive query requirements; Remember that users seldom want to see too much data in one go; Use setMaxRows(), setMaxFieldSize(), and SetFetchSize(); Decrease the column size; Use the smallest packet size that will meet your needs (if the driver supports packet sizing).
* Use a parametrized remote procedure call (RPC) rather than passing parameters as part of the RPC call, e.g. use Connection.prepareCall("Call getCustName (?)").setLong (1,12345) rather than Connection.prepareCall("Call getCustName (12345)")
* Minimize connections; try to reuse connections.
* Turn autocommit off.
* Avoid using distributed transactions.
* Use getBestRowIndentifier() to determine the optimal set of columns to use in the *Where* clause for updating data. (The columns returned could be pseudo-columns that can provide pointers to the exact location of the data, and are not obtained by getColumns().)

<http://www.precisejava.com/javaperf/j2ee/EJB.htm>  
EJB performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

* Transactions should span the minimum time possible as transactions lock database rows.
* Use the lowest cost locking available from the database that is consistent with any transaction.
* Use a dirty flag where supported by the EJB server to avoid writing unchanged EJBs to the database.
* Commit the data after the transaction completes rather than after each method call (where supported by EJB server).
* Do bulk updates to reduce database calls.
* Tune the connection pool size to minimize the creation and destruction of database connections.
* Use JDBC directly rather than using entity beans when dealing with large amounts of data such as searching a large database.

<http://www.precisejava.com/javaperf/j2ee/JDBC.htm>  
JDBC performance tips (Page last updated November 2001, Added 2001-12-26, Authors Ravi Kalidindi and Rohini Datla, Publisher PreciseJava). Tips:

* Use the fastest driver available to the database: normally type 4 (preferably) or type 3.
* Tune the defaultPrefetch and defaultBatchValue settings.
* Get database connections from a connection pool: use javax.sql.DataSource for optimal configurability. Use the vendor's connection pool; or ConnectionPoolDataSource and PooledConnection from JDBC2.0; or a proprietary connection pool.
* Batch your transactions. Turn off autocommit and explicitly commit a set of statements.
* Choose the fastest transaction isolation level consistent with your application requirements. Levels from fastest to slowest are: TRANSACTION\_NONE, TRANSACTION\_READ\_UNCOMMITED, TRANSACTION\_READ\_COMMITED, TRANSACTION\_REPEATABLE\_READ, TRANSACTION\_SERIALIZABLE.
* Close resources (e.g. connections) when finished with them.
* Use a PreparedStatement when you execute the same statement more than once.
* Use CallableStatement to execute stored procedures. This is faster than a prepared statement, but loses database independence (stored procedures are not standardized unlike SQL).
* Batch updates and accesses with Statements and ResultSets (with executeBatch() and setFetchSize()).
* Set up the proper direction for processing rows.
* Use the proper getXXX() methods.
* Write SQL queries that minimize the data returned.
* Cache read-only and read-mostly tables data.
* Use the Page-by-Page Iterator pattern to repeatedly pass small amounts of data rather than huge chunks.

<http://www.theserverside.com/resources/article.jsp?l=Prepared-Statments>  
Optimizing JDBC Prepared Statments. Also a followup discussion at <http://www.theserverside.com/discussion/thread.jsp?thread_id=8013> (Page last updated July 2001, Added 2001-08-20, Author ?, Publisher The Server Side). Tips:

* Databases analyze query statements to decide how to process them most optimally, then cache the resulting query plan, keyed on the full statement. Reusing identical statements reuses the query plan.
* Altering the statement causes a new query plan to be generated for each new statement. However statements with parameters can have the query plan reused, so use parameters rather than regenerating the statement with different values.
* Using a new connection requires a prepared statement to be recreated. Reusing connections allows a prepared statement to be reused.
* Connection pools should have associated PreparedStatement caches so that the PreparedStatements are automatically reused.

<http://www.onjava.com/pub/a/onjava/2002/01/09/dataexp1.html>  
Expiring cached data (Page last updated January 2001, Added 2002-01-25, Author William Grosso, Publisher OnJava). Tips:

* Database connection pools can take one of two strategies: a limited size pool, where attempts to make connections beyond the pool size must wait for a connection to become idle; or a flexible sized pool with a preferred size which removes idle connections as soon as the preferred size is exceeded (i.e. temporarily able to exceed the preferred size). The fixed size pool is generally considered to be the better choice.

<http://www.onjava.com/pub/a/onjava/2001/12/19/eejbs.html>  
EJBs are wonderful (Page last updated December 2001, Added 2001-12-26, Author Tyler Jewell, Publisher OnJava). Tips:

* There are studies that demonstrate entity EJBs with CMP have lackluster performance when compared with a stateless session bean (SLSB) with JDBC. [Author points out however that SLSB/JDBC combination is less robust, less configurable, and less maintainable].

<http://www.fawcette.com/javapro/2001_12/magazine/features/kkothapalli/>  
EJB performance tips (Page last updated December 2001, Added 2001-12-26, Author Krishna Kothapalli and Raghava Kothapalli, Publisher JavaPro). Tips:

* Use session bean wrapper for returning multiple data rows from an entity bean, rather than returning one row at a time.
* Use session beans for database batch operations, entity beans typically operate only one row at a time.
* Use entity beans when only a few rows are required for the entity, and when rows need to be frequently updated.
* Tune the underlying system, e.g. TCP/IP parameters, file limits, connection pool parameters, EJB pools sizes, thread counts, number of JVMs, JVM heap size, shared pool sizes, buffer sizes, indexes, SQL queries, keep/alive parameters, connection backlogs.

<http://www.sys-con.com/java/article.cfm?id=712>  
J2EE challenges (Page last updated June 2001, Added 2001-07-20, Author Chris Kampmeier, Publisher Java Developers Journal). Tips:

* Avoid the following common mistakes: Failure to close JDBC result sets, statements, and connections; Failure to remove unused stateful session beans; Failure to invalidate HttpSession.
* Performance test various options, for example, test both Type 2 and Type 4 JDBC drivers; Use a load-generation tool to simulate moderate loads; monitor the server to identify resource utlization.

<http://www.sys-con.com/java/article.cfm?id=673>  
J2EE Application servers (Page last updated April 2001, Added 2001-04-20, Authors Christopher G. Chelliah and Sudhakar Ramakrishnan, Publisher Java Developers Journal). Tips:

* Application servers should use connection pooling and database caching to minimize connection overheads and round-trips.

<http://java.oreilly.com/news/jsptips_1100.html>  
Hans Bergsten's top ten JSP tips (Page last updated November 2000, Added 2001-01-19, Author Hans Bergsten, Publisher O'Reilly). Tips:

* Database access is typically very expensive in terms of server resources. Use a connection pool to share database connections efficiently between all requests, but don't use the JDBC ResultSet object itself as the cache object.

<http://www.sys-con.com/java/article.cfm?id=713>  
Moving from JSP to EJB (Page last updated June 2001, Added 2001-06-18, Author Patrick Sean Neville, Publisher Java Developers Journal). Tips:

* Under heavy loads, entity beans should do more than merely represent a table in a database. If you are merely retrieving and updating data values, consider using JDBC within session beans instead.
* If you have one large database host but only a small Web and middleware host, consider moving much of your logic into stored procedures and calling them via JDBC in session beans.

<http://www.devx.com/judgingjava/articles/maso/default.asp>  
Judging various aspects of Java, including performance (Page last updated May 2001, Added 2001-06-18, Author Brian Maso, Publisher DevX). Tips:

* J2EE defines component models with high scalability potential. Maximizing scalability requires sticking to stateless session beans and handling all database interactions programmatically (through pooled JDBC connections).

<http://www.javaworld.com/javaworld/jw-02-2001/jw-0202-cachedrow.html>  
Article on using CachedRowSet, a ResultSet that doesn't need continuous connection to the database (Page last updated February 2001, Added 2001-02-21, Author Taylor G. Cowan, Publisher JavaWorld). Tips:

* CachedRowSet provides cached result sets that do not require continuous connection to the database, allowing connections to be reused more efficiently.
* Using CachedRowSet lets you batch updates, and execute them asynchronously.
* CachedRowSet also supports offline work which is later synchronized.
* CachedRowSet is probably not appropriate for managing large datasets.

<http://www.webdevelopersjournal.com/columns/connection_pool.html>  
Article on connection pools (Page last updated September 1999, Added 2001-02-21, Author Hans Bergsten, Publisher Web Developers Journal). Tips:

* Reuse database connections using a connection pool.

<http://www-106.ibm.com/developerworks/java/library/j-jtp0410/?loc=j>  
Java transaction management (JTS) (Page last updated April 2002, Added 2002-04-26, Author Brian Goetz, Publisher IBM). Tips:

* A container managing transactions can identify communications to the same database, and automatically convert a two-phase transaction into a more efficient single-phase commit.

<http://www.javareport.com/html/from_pages/article.asp?id=799&mon=4&yr=2001>  
Various strategies for connecting to databases (Page last updated March 2001, Added 2001-04-20, Author Prakash Malani, Publisher Java Report). Tips:

* Use pooled connections to reduce connection churn overheads.
* javax.sql.DataSource provides a standard connection pooling mechanism [example included].
* Obtain and release pooled conections within each method that requires the resource if the connection is very short (termed "Quick Catch-and-Release Strategy" in the article). However do not release the connection only to use it again almost immediately, instead hold the connection until it will not be immediately needed.
* The performance penalty of obtaining and releasing connections too frequently is quite small in comparison to potential scalability problems or issues raised because EntityBeans are holding on to the connections for too long.
* The "Quick Catch-and-Release Strategy" is the best default strategy to ensure good performance and scalability.

<http://www.eweek.com/article/0,3658,s=708&a=23115,00.asp>  
Database comparison (Page last updated February 2002, Added 2002-04-26, Author Timothy Dyck, Publisher E-Week). Tips:

* SQLServer has driver problems that slow access to it.
* Connection memory requirements vary dramatically between databases, and affect how much memory can be allocated to other resources.
* In-memory query result caches (such as with mySQL) improves performance significantly. (Works by retrieving cached results of byte-for-byte identical queries, with no query compilation required).
* Add extra indexes.
* Arrange the stored order of rows to best satisfy the queries.
* Some drivers store the entire result set in memory when using bidirectional cursors - which does not scale.

<http://www.theserverside.com/resources/article.jsp?l=Tips-On-Performance-Testing-And-Optimization>  
Server performance testing (Page last updated 2000, Added 2001-05-21, Author Floyd Marinescu, Publisher The Server Side). Tips:

* Response time increasing too much when database is over populated probably indicates lack of or inappropriate indexing on the database.
* Response time increasing exponentially as load increases, you need to improve scalability by optimizing the application or adding resources.
* Use SQL clause with EXPLAIN or similar (e.g. "Explain select \* from table where tablefield = somevalue") to ensure that the database is doing an indexed search rather than a linear searches of large datasets.
* Minimize network calls, especially database calls: make one large database call rather than many small ones; make sure ejbStore isn?t storing anything for read only operations; use Details Objects to get entity bean state rather than making many trips for each aspect of state.

<http://developer.java.sun.com/developer/technicalArticles/ebeans/sevenrules/>  
Optimizing entity beans (Page last updated May 2001, Added 2001-05-21, Author Akara Sucharitakul, Publisher Sun). Tips:

* Use container-managed persistence when you can. An efficient container can avoid database writes when no state has changed, and reduce reads by retrieving records at the same time as find() is called.
* Minimize database access in ejbStores. Use a "dirty" flag to avoid writing tee bean unless it has been changed.
* Always prepare your SQL statements.
* Close all database access/update statements properly.
* Avoid deadlocks. Note that the sequence of ejbStore calls is not defined, so the developer has no control over the access/locking sequence to database records.

<http://www.java-pro.com/upload/free/features/Javapro/2001/07jul01/ah0107/ah0107-1.asp>  
Experiences building a servlet (Page last updated June 2001, Added 2001-06-18, Author Asif Habibullah, Jimmy Xu, Publisher Java Pro). Tips:

* Make the servlets thread-safe and use connection pooling.
* Use PreparedStatements rather than plain Statement objects.
* Use database stored procedures.

<http://www.java-zone.com/free/articles/sf0101/sf0101-1.asp>  
Choosing a J2EE application server, emphasizing the importance of performance issues (Page last updated February 2001, Added 2001-02-21, Author Steve Franklin, Publisher DevX). Tips:

* Application server performance is affected by: the JDK version; connection pooling availability; JDBC version and optimized driver support; caching support; transactional efficiency; EJB component pooling mechanisms; efficiency of webserver-appserver connection; efficiency of persistence mechanisms.
* Decide on what is acceptable downtime for your application, and ensure the app server can deliver the required robustness. High availability may require: transparent fail-over; clustering; load balancing; efficient connection pooling; caching; duplicated servers; scalable CPU support.

<http://developer.java.sun.com/developer/community/chat/JavaLive/2001/jl0619.html>  
Sun community discussion on "Optimizing Entity Beans" with Akara Sucharitakul (Page last updated June 2001, Added 2001-07-20, Author Edward Ort, Publisher Sun). Tips:

* Prepared SQL statements get compiled in the database only once, future invocations do not recompile them. The result of this is a decrease in the database load, and an increase in performance of up to 5x.

<http://www.devx.com/premier/mgznarch/javapro/2001/01dec01/sl0112/sl0112-1.asp>  
Creating Web-based, interactive graphics. (Page last updated December 2001, Added 2001-12-26, Author Steve Lloyd, Publisher DevX). Tips:

* Close java.sql.Statements when finished with.

<http://www7b.boulder.ibm.com/dmdd/library/techarticle/0204pooloth/0204pooloth.html>  
High performance inserts with DB2 and JDBC (Page last updated April 2002, Added 2002-07-24, Author Krishnakumar Pooloth, Publisher IBM). Tips:

* Use SQLJ to get the use of buffered inserts, and modify the code generated from SQLJ to reuse the RTStatement object.

<http://www.allaire.com/Handlers/index.cfm?ID=17266&Method=Full&Cache=Off>  
Connection Pooling with JRun (Page last updated June 2001, Added 2001-08-20, Author Karl Moss, Publisher Allaire). Tips:

* Establishing an initial connection is one of the most expensive database operations. Use a pool of connections that are ready and waiting for use to minimize the connection overhead.
* Connection pooling is one of the largest performance improvements available for applications which are database intensive.
* Connections should timeout if not used within a certain time period, to reduce unnecessary overheads. Initial and maximum pool sizes provide further mechanisms for fine-tuning the pool.
* JDBC 2.0 supports connection pooling, though a particular driver may or may not use the support. If pooling is supported by the driver, it is probably more efficient than a proprietary pooling mechanism since it can leverage database specific features.

<http://www.as400.ibm.com/developer/java/faq/perffaq.html>  
Some IBM Java performance tips. Although intended for AS/400 Java, many tips are generally applicable (Page last updated ?, Added 2000-10-23, Author ?, Publisher IBM). Tips:

* Use Prepared Statements.
* Store character data in DB2 as Unicode, numeric data as float.

<http://developer.java.sun.com/developer/Books/J2EETech/ch3.pdf>  
Rambling discussion of building J.Crew website, in Chapter 3 of "J2EE Technology in Practice" (Page last updated September 2001, Added 2001-10-22, Authors Dao Ren, Dr. Rick Cattell and Jim Inscore, Publisher Sun). Tips:

* Use database connection pooling
* Cache Database Requests

<http://www.javaworld.com/javaworld/jw-01-2002/jw-0125-overpower.html>  
Wrapping PreparedStatement (Page last updated January 2002, Added 2002-02-22, Author Bob Byron and Troy Thompson, Publisher JavaWorld). Tips:

* With Statement, the same SQL statement with different parameters must be recompiled by the database each time. But PreparedStatements can be parametrized, and these do not need to be recompiled by the database for use with different parameters.
* [Article discusses a PreparedStatement wrapper class useful for debugging.]