Scalability and Performance: JDBC Best Practices and Pitfalls

Julia Gutjahr and Andreas Loew
Java Architects

Sun Microsystems GmbH,
Sun Java Center,
Ampèrestraße 6, D-63225 Langen, Germany
{Julia.Gutjahr|Andreas.Loew}@germany.sun.com
http://www.sun.de
http://www.sun.com/javacenter

Abstract. With the recent release of version 3.0, the Java Community Process once again extended JDBC, the Java database API for relational database access, mainly adding some features to support highly scalable enterprise-level applications running within J2EE application server environments even better. Most of these concepts and features being also available for JDBC 2.x environments (at least as part of vendor-specific or add-on solutions), we will present some best practices and common pitfalls regarding Java database access from our project work as members of the Sun Java Center in Germany. We will take a look at how to use resource pooling properly, minimize network roundtrips, chose the appropriate tier for data-intensive operations and do some application level caching of frequently accessed, but fairly static master data.

1 Introduction

The classes and interfaces of the Java Database Connectivity API (JDBC) [1] enable Java applications to access (object-)relational database management systems (DBMS). They are defining a vendor-independent call level interface and can be used to create connections to the database server, send SQL statements to the database and retrieve their results. Using JDBC, a Java developer is able to create database applications that are independent of a specific DBMS.

The most important design goal for the recent version 3.0 of the JDBC API[1] was to improve the suitability and scalability of JDBC with regards to large server-side Java 2 Enterprise Edition (J2EE) applications running on top of Java application servers.

JDBC 3.0 is an integral part of the new Java 2 Standard Edition (J2SE) SDK 1.4 [2]. In contrast to previous versions, the package *javax.sql*, formerly only available as part of the Enterprise Edition (J2EE) [3], has now been integrated into Standard Edition. This package contains for example the classes and interfaces needed for connection pooling and distributed transaction support.

[1] released on February 13, 2002

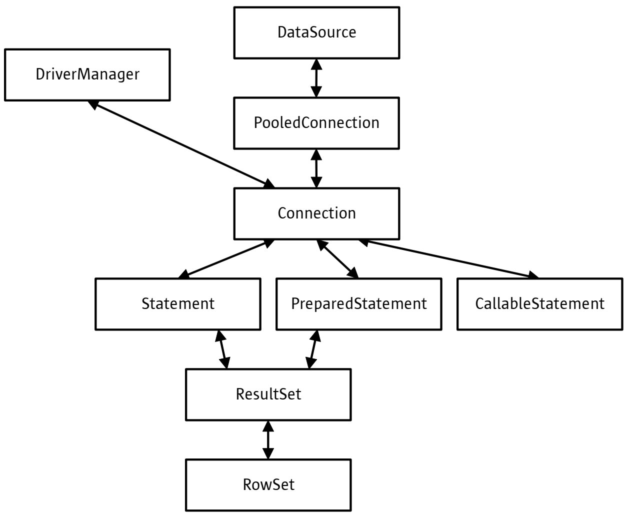

Fig. 1. Relationship among the main JDBC Interfaces [4]

The new JDBC version retains full compatibility with all existing applications and drivers, which means that all software written to use previous JDBC API versions will remain fully functional.

On the Sun Microsystems JDBC product homepage, there is a list of available JDBC drivers from different DBMS vendors and independent providers that registered their drivers at the Sun site [5]. At present, only a few JDBC implementations claim to be implementing the version 3.0 API, but as all major DBMS companies were involved in its definition and are willing to support it, it will be merely a matter of time until choice becomes broader.

As most of the recent additions just standardize some functionality formerly introduced as part of vendor-specific solutions (like connection and statement pooling), they will in most cases even be applicable to “pure” JDBC 2.x environments.

2 Database Access Best Practices: Across the Tiers

Besides meeting the business and functional requirements, it has become a commonly accepted fact that, for a software development project to be successful, the qualities of service (also called systemic qualities or sometimes just the “-ilities”: e.g. availability, reliability, performance, scalability, maintainability etc.) must be a primary design element[2] and not just be addressed as a part of performance tuning activities after development has finished (and, as usual, the application is running too slow).

This is especially true for today’s large-scale, multi-tiered, database-centric applications with high throughput and/or scalability requirements, because with the grow-

[2] Putting systemic qualities back into focus is one of the main concerns of Sun's 3-Dimensional Architectural Framework (3DF) and the SunTone Architectural Methodology (SunTone AM) [6].

ing number of components and tiers involved in the architecture (all being highly configurable and really needing to be configured and used properly), the number of design and development decisions and the risk of getting caught by potential pitfalls is also rapidly increasing.

2.1 Starting Points

When it comes to database access, there are a quite a number of best practices and potential starting points to be considered throughout all phases of the typical software project life cycle, starting from application architecture and design and ending with performance tuning activities, testing and deployment. From our experience with real-life customer projects, the following areas should be analyzed:

- DBMS configuration parameters
- Physical database design
- SQL statements and query execution plans
- JDBC driver setup/configuration (see section 3)
- JDBC usage from within Java application code (see section 4)
- JDBC driver selection (see section 5)
- Application architecture general design issues (see section 6)

As it can be easily seen from this list, just focusing on all kinds of JDBC related activities is simply not enough. Especially with very large databases, the first three (Java-independent) general database configuration tasks are probably even more important for overall database and application performance than the next three items closely related to JDBC.

DBMS configuration parameters. As DBMS themselves are highly sophisticated and complex software systems, they usually have a large number of configuration options. Some of the more important ones include database block size, total size of global in-memory object cache, maximum number of concurrent sessions etc.[3]

Physical database design. The physical design of database storage has an important impact on database overall performance. Issues to be taken into consideration include

- the physical arrangement of tablespaces on storage volumes (hard disks)
- the assignment of database objects (tables, indexes, etc.) to tablespaces
- the storage structure of database objects within a tablespace (for Oracle, all options to be set in a storage clause)
- definition of indexes (which tables/columns should be indexed)
- other performance tuning options, depending on DBMS features (e.g. partitioned tables, materialized views, …).

[3] For an Oracle instance, these parameters are set in a configuration file called `init.ora` in the `oradata/admin` subdirectory.)

SQL statements and query execution plans. Although SQL is a relatively easy language to learn, its non-procedural nature and the complex and highly DBMS-specific query optimizer algorithms tend to produce obscure performance-related issues. As a result, it is much harder to write efficient SQL than it is to write functionally correct SQL. Additionally, there seems to be insufficient awareness of the need to carefully monitor and tune SQL performance, and the DBMS-related tools and techniques needed to tune SQL are not yet known widely enough.[4] It is quite common that massive performance gains can be achieved even by tuning just a small set of really bad SQL statements[5].

While all the aforementioned issues should be addressed by database administrators or database performance specialists, the last four tasks are the genuine responsibility of Java architects and developers. We will cover them in detail in chapters 3 to 6 and close with some remarks on the future role and importance of JDBC in a complex J2EE architecture.

2.2 Best Practices Sample Code and Performance Measurements

Due to space limitations, nearly all coding examples and references to the various performance measurements we carried out comparing bad and best practices using an Oracle 9i database with the most recent Oracle 9.2.0.1 JDBC drivers had to be removed from this experience report. We will try and make most of these code examples and measurements available on the Net.ObjectDays web site[6] along with the session slides.

3 JDBC Driver Setup and Configuration

3.1 Use Connection Pooling

The concept of connection pooling (sometimes also called connection caching) is perhaps the most important single JDBC improvement in terms of performance tuning. It was especially invented for and aimed at the requirements of web-based applications running from within the web container of a J2EE application server.

In this scenario, the extremely resource-intensive opening and closing of database connections is just too costly to be affordable for each and every new HTTP request. Instead, a *ConnectionPoolDataSource* was introduced as a *Connection* factory that opens a certain number of database connections on application server startup and makes them available later on to application server threads serving requests.

[4] Some examples for Oracle include analyzing SQL execution trees using `EXPLAIN PLAN`, using `tkprof` to profile database sessions and investigating global SQL dynamic execution statistics from `V$SQLAREA`.

[5] In some cases, even slight changes to a query's execution plan can improve performance by 100 percent and more.

[6] *http://www.netobjectdays.org* (category "Archive", JaDa 2002 workshop)

JDBC 3.0 even defines a basic set of properties that *ConnectionPoolDataSource* implementations may support[7]. Developers should not modify these properties directly through the API, but rather using their application server or data store instrumentation. The properties are shown in the table below.

Property Name	***Description***
maxStatements	Maximum number of cached *(Prepared)Statements* that should be kept open in the connection pool's statement pool at any given time
initialPoolSize	Number of physical connections the connection pool should contain when being created
minPoolSize	Minimum number of physical connections the pool should keep available at all times
maxPoolSize	Maximum number of physical connections the pool should contain at any given time
maxIdleTime	Maximum number of seconds a physical connection may remain unused in the pool before it gets closed (and possibly removed)
propertyCycle	Interval (in seconds) the pool should wait before enforcing the current policy defined by the values of the above connection pool properties

Table 1. JDBC 3.0 Standard Connection Pool Properties

The actual savings of connection pooling will heavily increase with the growing number of concurrent HTTP requests (rsp. threads within the application server) that will need to access the database.

3.2 Set Connection Properties Appropriately

A JDBC *Connection* instance always represents a database connection (session) to a specific database. SQL statements are executed and their results returned within the context of a connection.

Switch the Auto-commit "Feature" Off. By default, a *Connection* object is in auto-commit mode, which means that it automatically executes a commit operation on the database after executing each SQL statement. This behavior does not only increase network roundtrips, but in most cases also is unwanted from a functional point of view, as a logical unit of work consisting of two or more SQL statements cannot be completely rolled back any more after completion of the first SQL statement. It is therefore important to switch this "feature" explicitly off and let the user application (or the J2EE application server) manage the transaction boundaries itself:

```
conn.setAutoCommit(false); // conn is a Connection
```

[7] Note that, as many current driver implementations, the most recent Oracle 9i 9.2.0.1 JDBC driver does not yet support any of these standard properties.

Transaction Isolation Mode. The isolation level describes the capacity of concurrent transactions to view data that has been updated, but not yet committed, by another transaction. A higher isolation level always means less concurrency and a greater likelihood of performance bottlenecks, but also a decreased chance of reading inconsistent data.

Isolation Level	***Description***
TRANSACTION_NONE	Transactions are not supported. (There is no rollback!)
TRANSACTION_READ_UNCOMMITTED	All update anomalies can possibly happen, including reading dirty data modified by another uncommitted transaction.
TRANSACTION_READ_COMMITTED	It is not allowed to read data from transactions that have not yet committed. Changes done by other transactions become visible immediately after these transactions have committed. Non-repeatable reads can occur for data that has been concurrently changed by another committed transaction.
TRANSACTION_REPEATABLE_READ	It is not allowed to read data from transactions that have not yet committed. In general, all reads are repeatable, but in addition newly inserted data of other committed transactions may become visible ("phantom reads").
TRANSACTION_SERIALIZABLE	Highest level of consistency (standard level according to SQL-99). ACID transaction properties (Atomicity, Consistency, Isolation, Durability) are observed. All reads are repeatable, and no phantom reads can occur.

Table 2. JDBC Transaction Isolation Level

Although it might seem a good rule of thumb to simply choose the highest isolation level that yields acceptable performance using e.g.

```
conn.setTransactionIsolation(TRANSACTION_READ_COMMITTED);
                                                    // conn is a Connection,
```

in most cases this is not a real option, as JDBC drivers normally tend to support just one or two different isolation levels that are most compatible with the way the DBMS server itself handles transactions. We therefore simply propose to adhere to the DBMS's default isolation level[8] unless there are some good reasons not to do so.

3.3 Use Comprehensive (Connection Spanning) Statement Pooling

With all JDBC versions prior to 3.0, the *Connection* instances existing within a connection pool were strictly separated from each other, so that SQL statements sent to the database from one connection were completely unknown to all other connections. In the typical J2EE application server scenario, this meant that for each application server thread, after getting a new *Connection* from the connection pool, each and every *PreparedStatement*[9] had to be re-prepared.

[8] e.g. for Oracle: *TRANSACTION_READ_COMMITTED*

[9] see section 4.2 below for details on the distinction between PreparedStatements vs. Statements

JDBC 3.0 now officially supports the concept of a statement pool that is spanning connections: All the statements sent to the database from all connections within the pool will be cached at one place, so that they can be reused by all connections participating in the pool.

Of course, the size of the statement pool has to be set appropriately. To do this, the connection pool property *maxStatements* (see section 3.1 above) can be used.

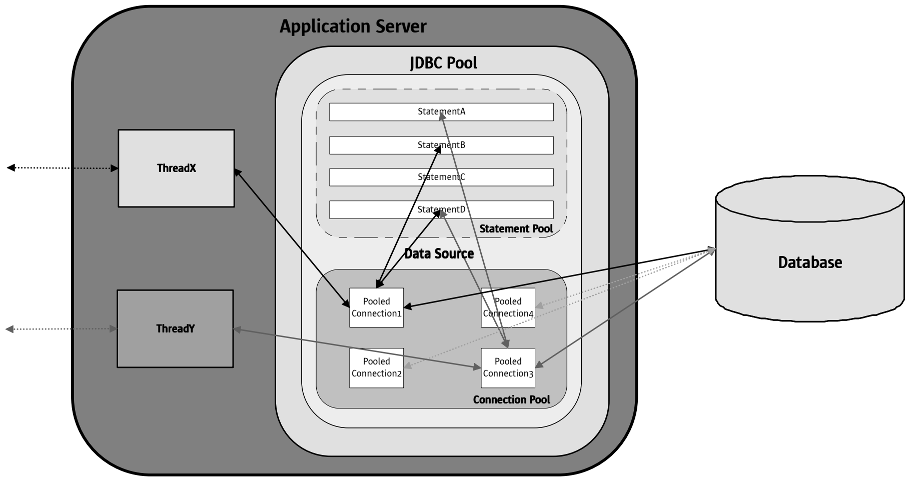

Fig. 2. Connection and Statement Pooling at a Glance

From our performance evaluations, we found the performance gain using the statement pool with the Oracle JDBC drivers not to be as high as we had expected[10]. (This finding will of course vary with the DBMS product used.)

The most probable reason for this is that the Oracle server already manages another sophisticated session spanning statement pool[11]. This database statement cache will be used even if the Java client does no JDBC statement caching, so the actual performance difference we measured was mainly related to network traffic savings: If a SQL statement could already be found in the Java statement pool, then not the lengthy SQL text, but only the internal reference (cursor handle) needs to be passed over the network along with the current bind variable values.

[10] just one and a half seconds for ten identical series of 100 different *PreparedStatements*

[11] This cache is called the library cache and located in the shared pool of the Oracle server's System Global Area (SGA). When a SQL statement is submitted, the server first checks the library cache to see if an identical statement is already present in the cache. If it is, Oracle uses the stored parse tree and execution path for the cached statement, rather than rebuilding these structures from scratch.

4 JDBC Usage from within Java Application Code

4.1 Take Care of Sound Resource Handling

Although the general concept of Java as a programming language doing garbage collections instead of having to free memory explicitly mostly disburdens the application developer from having to worry about memory leaks in his application code[12], this is not the case when it comes to the use of shared resources:

Whenever an application server has assigned a database connection coming from its connection pool (and because of that being a shared resource) to a thread that is serving an HTTP request, the *PooledConnection* object will be marked as used. All other JDBC resources created on top of the associated *Connection* during request processing (*Statements*, *PreparedStatements*, *ResultLists* etc.) will further on hold a reference to this connection. After request processing has finished, the thread should close the JDBC connection and return it to the pool.

But as the *Connection* object itself will just be returned to the pool (and not destroyed or garbage collected), there is no guarantee that any particular JDBC driver implementation will be properly freeing them on the call to *conn.close()*. Although this is intended to work if the JDBC driver is well implemented and thoroughly tested, experience shows that frequently, this is not the case[13], and failing to close and free JDBC resources will indeed lead to severe performance degradations, as soon as connection pool and/or memory limits will be reached. We can therefore just recommend to developers to observe sound coding standards and explicitly close each and every JDBC resource that has been allocated as soon as it is not needed any more.

In particular, this also applies to exception handling: We have seen a lot of code that properly closes result sets, statements and connections in the normal control flow, but fails to do so in case an exception got thrown.

“Finally”, best practice is very easy to follow: simply provide every single *try* and *catch* block with a *finally* clause that closes the appropriate JDBC resources.

4.2 Statements versus Prepared Statements: When to Use What?

While the JDBC *Statement* object is appropriate for sending ad-hoc or one-time only queries to the database server, whenever a SQL statement is likely to be used several times unchanged or just with varying input parameters (be it from the current thread or – in case a connection spanning statement cache pool can be used – another thread running in parallel), it should be sent to the database using a JDBC *PreparedStatement* object.

Using plain *Statement* objects, the SQL command string gets sent to the database each time the *stmt.execute(sqlString)* method gets called. The database server then has to parse the query, calculate an execution plan, execute the query and return a cursor

[12] The only exception is if a reference to some objects that are not needed any more is kept by mistake, they will of course not be garbage collected.

[13] We have seen lots of problems with various JDBC drivers from different sources that were dealing with severe resource handling problems in the past.

handle to the JDBC driver which gets used by the driver implementation in order to access the *ResultSet*.

A *PreparedStatement* introduces the concept of bind variables: Although it is always possible to dynamically create a SQL string using the values of application variables, this would lead to a large number of very similar SQL statements sent to the database. Just imagine several people in a call center accessing customer reference data by searching for the last name – the only difference between their SQL statements is the particular name to search for. With a *PreparedStatement*, there is only one SQL string for this scenario, and a placeholder (the bind variable, represented by a "?" in the SQL string) will be used for the current search value of the last name. Just before executing the statement, this bind variable will be assigned its current value:

```
PreparedStatement pstmt = conn.prepareStatement(
  "SELECT * FROM c_customer WHERE c_last_name LIKE ?");
                                      // prepare the statement
// do something inbetween
pstmt.setString(1, "Sm%");            // assign the bind variable
```

With a *PreparedStatement*, sending of the SQL command string, parsing and determining the execution plan has just to be done for the first time that this particular *PreparedStatement* is needed (which is the job of the *conn.prepareStatement()* factory method). The database server will then return a cursor handle for the given query that the JDBC driver stores internally with the newly created *PreparedStatement* object. Subsequent calls to the same *PreparedStatement* can then just confine to setting current values for the bind variables. When the *pstmt.execute()* method gets called, the JDBC driver will use the cursor handle stored with the *PreparedStatement* instance in order to execute the query and return a handle to its *ResultSet*.

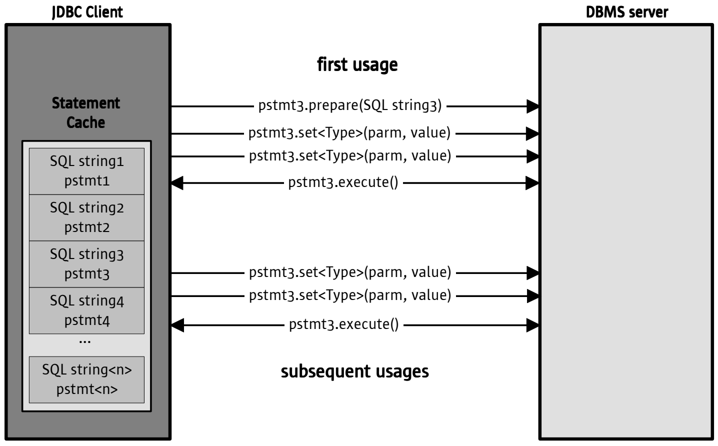

Fig. 3. Prepared Statement Communication Scenario

Another good reason for using *PreparedStatements* is that all statement pooling extensions described in section 3.3 will only work on *PreparedStatements*.

4.3 Minimize the Size of Result Sets

In order to save database server, network and JDBC driver resources, it is also important to try to minimize the size of the *ResultSets* retrieved from the database by trying to describe the queries as specific as possible.

In particular, it is not advisable to use the abbreviated `SELECT *` notation to get back values for all the columns of a given database table. Even if this might be appropriate at the time of writing the query, after the addition of (possibly numerous or large) new columns to the underlying database table, things can change rapidly: The query can become imperformant and waste resources, because it will now also have to send the values for the added columns over the network, although they will not be used at all by the application.

As an example, we changed a query selecting just one attribute of a table to select all its attributes (`SELECT *`). Even though the additional column values were not even retrieved from the *ResultSet*, the execution time more than doubled.

4.4 Avoid the Usage of Metadata Methods Whenever Possible

With the interface *DatabaseMetaData*, the JDBC API offers the possibility to access meta data structures from the underlying DBMS in a vendor-independent, unified way. In order to do that, JDBC driver implementations have to run queries on the DBMS' internal dictionary tables that are sometimes very complex and hence costly. As a rule of thumb, a meta data access method is especially likely to be expensive, if it returns a *ResultSet* (like for example *DatabaseMetaData.getColumns()*) and gets called with some very unspecific or even missing search patterns, so that the size of the *ResultSet* will probably be quite large.

Because of that, generally speaking, accesses to database meta data should be avoided. In most cases, it is perfectly possible to either replace them by using application properties or at least get the meta data just once on application startup and then store it in an application-specific cache.

4.4 Use Statement Batches Where Appropriate

Starting with JDBC 2.x, all *Statement* interfaces allow the definition of SQL batches that accumulate a series of consecutive SQL statements as one unit that gets sent to the DBMS server as a whole and consequently minimizes the number of network roundtrips.

For example, a common usage of a *PreparedStatement* batch is for loading a table with data by executing a series of `INSERT` statements:

```
PreparedStatement pstmt=conn.prepareStatement(
  "INSERT INTO employees (id, name) VALUES (?, ?)");
pstmt.setInt(1, 1234);
pstmt.setString(2, "Scott McNealy");
pstmt.addBatch();
pstmt.setInt(1, 6789);
pstmt.setString(2, "James Gosling");
pstmt.addBatch();
int[] updateCounts = pstmt.executeBatch();
```

But be aware, there is one issue about this: The JDBC specification currently does not define what the driver implementation should do in case of an exception being thrown during the execution of a batch (e.g. abort the whole batch immediately or continue executing the remaining statements), so just be sure to implement some sound exception handling yourself.

5 Selecting A JDBC Driver Implementation

Besides all best practices mentioned in chapters 3 and 4 that are applicable to any JDBC driver, it is also very important to notice that the JDBC driver implementation itself also has a crucial influence on performance.

Vendor Supplied vs. Third Party Drivers. One of the main advantages of the Java platform in general and JDBC in particular is about not being stuck with only one implementation of a single vendor, but instead having a choice between several different implementations of one common standard API. Besides vendor-supplied JDBC drivers, for all major commercial databases there are also a number of alternative implementations by independent software companies that are specifically addressing and focusing on the JDBC driver market.

Always Do Your Own Evaluations. Because there is no single "best of breed" JDBC driver implementation, when doing performance testing of your application or just running into some subtle problems, always have a try and replace the JDBC driver implementation (either with a newer or older revision of the same vendor's drivers or a different implementation by another vendor) and see if it does make a change.

Check Frequently for Updates of JDBC Drivers. As long as the list of features to fully support the latest version of the JDBC standard has not been dealt with and the update frequency of the JDBC driver implementations is as high as it currently is, always try to use (or at least evaluate) the most recent versions of your vendor's drivers[14]. The list of optimizations integrated in their recent driver versions is still growing for nearly all of the vendors, and the code might just have noticeably improved in terms of performance and resource usage with the latest revision, so just give it a try.

Type-2 vs. Type-4 Drivers. Some database vendors (like Oracle) even offer two different kinds of JDBC drivers: a type-2 and a type-4 driver implementation. While a type-2 driver implementation uses the Java Native Interface (JNI) to translate JDBC method calls into calls to a vendor-specific native code DBMS client API (e.g. Oracle's OCI) to communicate with the database, and hence requires the DBMS

[14] A special remark on Oracle's JDBC drivers: It is not at all required – as people often think - that the JDBC driver revision matches the database server revision that is being used. Generally speaking, a later Oracle JDBC revision will as well be able to connect to previous Oracle server releases, unless stated otherwise in the JDBC driver's release notes.

native client libraries to be installed, a type-4 (thin) driver is a 100% Java client-side implementation of the DBMS' client communication protocol (e.g. Oracle's SQL*Net) that does not need any additional DBMS client installation. Whether the type-2 or the type-4 driver performs better overall, mainly depends on whether

- the overhead for doing JNI calls on the particular operating system/JVM implementation with the type-2 (OCI) driver, or
- the performance penalty caused of having to do a lot more garbage collections because of the much larger number of intermediate Java objects being created with the type-4 (thin) driver

is more grave on the particular deployment platform[15]. Again, this can only be decided doing a performance comparison of both drivers in the particular environment.

6 Application Architecture General Design Issues

6.1 Selecting the Right Tier for a Given Functionality

A very important issue that should already be considered at design time for a given functionality is to deploy it to the right tier.

For example, in a database warehousing scenario where a small number of highly aggregated and consolidated values has to be calculated based on large amounts of data, it is not advisable to do these calculations on the middle tier, because all data needed to calculate the result values would have to be sent from the database tier over the network (in this scenario, the probable performance bottleneck) to the middle tier. Instead, it will make much more sense to extract this functionality into a separate component providing a well-defined API and put this component as near as possible to the data it needs to read.

Being able to use a DBMS that has a built-in Java Virtual Machine (JVM) within the database server (like Oracle, IBM's DB2 and Sybase do) is a major advantage here, because the Java class for this component then just can be deployed to the database JVM as a Java stored procedure. Without Java support in the DBMS, you can just migrate the application logic to a conventional SQL stored procedure. In any case, both approaches will lead to significant performance improvements just from avoiding data transfer over the network.

As an example, we deployed a method calculating the maximum cross sum of a numeric column for all rows on a table alternatively to the middle tier and the Oracle 9i built-in database server JVM[16].

[15] This again is dependent on influence factors like memory available for Java heap space, processor architecture and operating system.

[16] The Oracle JServer JVM is a complete, Java 2-compliant Java execution environment running in the same process space and address space as the DBMS kernel, sharing its memory heaps and hence directly accessing its relational data. In addition, the JVM is tightly integrated with the scalable shared memory architecture of the RDBMS (see fig. 4).

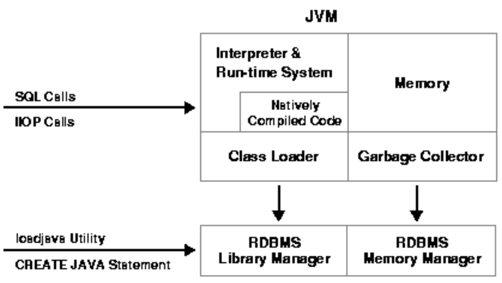

Fig. 4. Main Components of the Oracle JVM [7]

In order to run our Java class from within the Oracle JServer JVM, we just had to change the JDBC connection URL to point to the server-side internal JDBC driver[17], load the compiled Java class file into the database and define a PL/SQL wrapper function so that it can be called from ordinary SQL:

```
call loadjava –user ecperf/ecperf@mydb
  com\sun\germany\sjc\jdbcsamples\DataBaseUtils.class

SQL> create or replace function getmaxcrossum(wherestr varchar2)
       return number as language java name
       ‘com.sun.germany.sjc.jdbcsamples.DataBaseUtils.getMaxCrossSum
         (java.lang.String) return int’;

SQL> select getmaxcrossum('1 = 1') from dual;

GETMAXCROSSUM('1=1')
--------------------
                 107

1 row selected.
```

The possible savings of using the database server’s JVM due to avoiding context switches and data transfer through the networking protocol layers between server and client are the more noticeable the larger the amount of data is that has to be accessed from inside the stored procedure.

6.2 Consider the Use of Application Level Caching

Another example of a performance precaution that can already be addressed at design time is application level caching. Of course, the caching approach is only feasible for a very special kind of data (e.g. master data): data that gets frequently read, but rarely (or even never) updated. If necessary, a cache update event mechanism in order to refresh the cache and enable it to reflect database updates should be designed at the same time as the caching layer.

The first and quite obvious approach is to define an application-dependent cache and store the application’s particular business objects in e.g. standard Java Hashtables. While being a feasible solution for applications that are just in the design

[17] This works by just replacing `"jdbc:oracle:thin:@localhost:1521:MYDB"` (JDBC thin driver) with `"jdbc:oracle:kprb:"` (server-side internal JDBC driver).

phase, this approach has its downsides when a caching layer has to be added to an existing application, causing as little change to existing components as possible.

We have gained some positive experience with a caching layer we suggested for a customer project that is able to seamlessly integrate with existing JDBC applications. In order to achieve this, we did our own implementations of the JDBC interfaces *Connection, PreparedStatement* and *Statement* that are wrapping the standard JDBC driver's implementation classes for the particular DBMS and maintaining a cache of disconnected *ResultSet* objects.

While the conventional *Statement* implementation returns a standard JDBC *ResultSet* from the database, when its *execute(sqlString)* method is called, our own implementation class *CachedStatement* will intercept these calls and redirect them to the cache to see if there already is a *ResultSet* in the cache with the current SQL string as cache access key. If there is a such a cached *ResultSet*, it will simply be returned to the caller. Otherwise, our implementation will chain the call to the wrapped JDBC driver *Statement* implementation class, call its *execute()* method and get a standard JDBC *ResultSet* back. This *ResultSet* then will be put into the cache (using the SQL string that was used to retrieve it as a key), and then returned to the caller.

While this should be sufficient to describe the general approach, two additional implementation details have to be mentioned briefly:

- The standard JDBC *ResultSet* implementations are connected, which means they need a reference to an open database connection in order to be able to read additional data, support updates etc. It is therefore not appropriate to store a connected *ResultSet* in a global cache that will be shared by several threads running in parallel, each of them using a different connection. The solution here is to create disconnected *CachedRowSets*[18] [8] (still implementing the *ResultSet* interface and therefore not causing any changes in the surrounding application code) from the original ones and store them in the cache.
- In order to manage the cache, we chose to use the quite sophisticated Oracle Object Caching Service for Java (OCS4J)[19] implementation [9].

Using a very basic example implementation of the JDBC wrapping layer, we managed to get cached *ResultSet* objects back from the cache in less than 20 milliseconds, whereas executing a database query would need at least several hundreds of milliseconds.

[18] A *CachedRowSet* is an implementation of the *javax.sql.RowSet* interface first defined with the JDBC 2.x Optional API specification [8]. A disconnected *RowSet* object can be thought of as a set of rows that are being cached outside of a data source. Because they are lightweight and serializable, disconnected row sets can also be passed between different components of a distributed application.

[19] The Oracle Object Caching Service for Java (OCS4J) are an implementation of JSR 107 defining an API for in-memory caching of Java objects. Currently, they are part of the Oracle 9iAS product, but also separately available at [9].

7 JDBC – And Beyond?

In a complex J2EE architecture, JDBC still remains one of the low-level APIs that just establish the fundaments that higher level APIs dealing with persistent object storage - like EJB 2.0 Entity Beans, Java Data Objects (JDO) or commercial object-to-relational mapping tools - are relying on.

Consequently, applying the best practices we presented in this document to application servers and applications in an environment fitted with the appropriate JDBC drivers (providing the new features and extensions of JDBC 3.0) will not produce any ground-breaking news, but hopefully contribute its part to the development of more scalable and performant Java database applications.

References

1. Sun Microsystems: JDBC Data Access API Home Page,
   http://java.sun.com/products/jdbc
   Java Community Process: JSR 54 - JDBC 3.0 Specification,
   http://jcp.org/jsr/detail/054.jsp
2. Sun Microsystems: Java 2 Platform, Standard Edition, Overview,
   http://java.sun.com/j2se/1.4/datasheet.1_4.html
3. Sun Microsystems: Java 2 Platform, Enterprise Edition, Overview,
   http://java.sun.com/j2ee/overview.html
4. van Haecke, B.: JDBC 3.0 - Java Database Connectivity.
   M&T Books, New York, Cleveland, Indianapolis (2002)
5. Sun Microsystems: List of JDBC Drivers Available,
   http://java.sun.com/products/jdbc/drivers
6. Berg, D.: 3-D Architectures for Streamlined Enterprises,
   http://www.sun.com/service/about/features/3d_architectures.html
   SunTone Architecture Methodology: A 3-Dimensional Approach to Architectural Design,
   http://www.sun.com/service/sunps/jdc/suntoneam_wp_5.24.pdf
7. Oracle Corporation: Oracle 9i 9.0.1 Java Stored Procedures Developer's Guide,
   http://tahiti.oracle.com/pls/db901/db901.to_pdf?partno=a90210&remark=docindex
8. Sun Microsystems: JDBC 2.0 Optional package API specification,
   http://java.sun.com/products/jdbc/jdbc20.stdext.pdf
   Sun Microsystems: JDBC Row Set Early Access Page,
   http://developer.java.sun.com/developer/earlyAccess/crs
9. Oracle Corporation: Oracle Object Caching Service for Java Home Page
   http://otn.oracle.com/products/ocs4j/content.html