

# A Guided Tour of the SAP Java Persistence Framework — Achieving Scalable Persistence for Your Java Applications

---

Katarzyna Fecht, Adrian Görler, and Jürgen G. Kissner



*Katarzyna Fecht, SAP  
NetWeaver PMO, SAP AG*



*Adrian Görler, Java Server  
Technology Group, SAP AG*



*Jürgen G. Kissner, SAP  
NetWeaver Development,  
SAP AG*

*(complete bios appear on page 140)*

ERP systems, information systems, e-business applications, and Web services differ widely in scope, size, scalability, and resource requirements. But ultimately they all rely on data stores that guarantee the availability and integrity of the stored information. Meeting this expectation requires a stable, powerful, and scalable persistence mechanism that supports application portability and offers features such as database abstraction layers and diagnostic tools.

Starting with Release 6.20, SAP Web Application Server (SAP Web AS) supports runtime environments for two technologies. ABAP applications use the classic ABAP Engine, while Java applications run in the J2EE Engine.<sup>1</sup> Both software stacks communicate via well-defined interfaces (SAP JCo,<sup>2</sup> SAP Java Resource Adapter, and Web services) that make the ABAP persistence layer accessible from Java and vice versa. However, pure Java applications should ideally access the database directly from Java, rather than through ABAP. Naturally, the Java world offers a wide variety of solutions, standards, and APIs for achieving persistence in an application. But how does SAP Web AS support this demand? Is there a Java solution that possesses the qualities of the ABAP persistence layer<sup>3</sup> while adhering to Java standards? As of SAP Web AS 6.30, SAP introduces a comprehensive solution that meets these needs — the Java Persistence Framework.

---

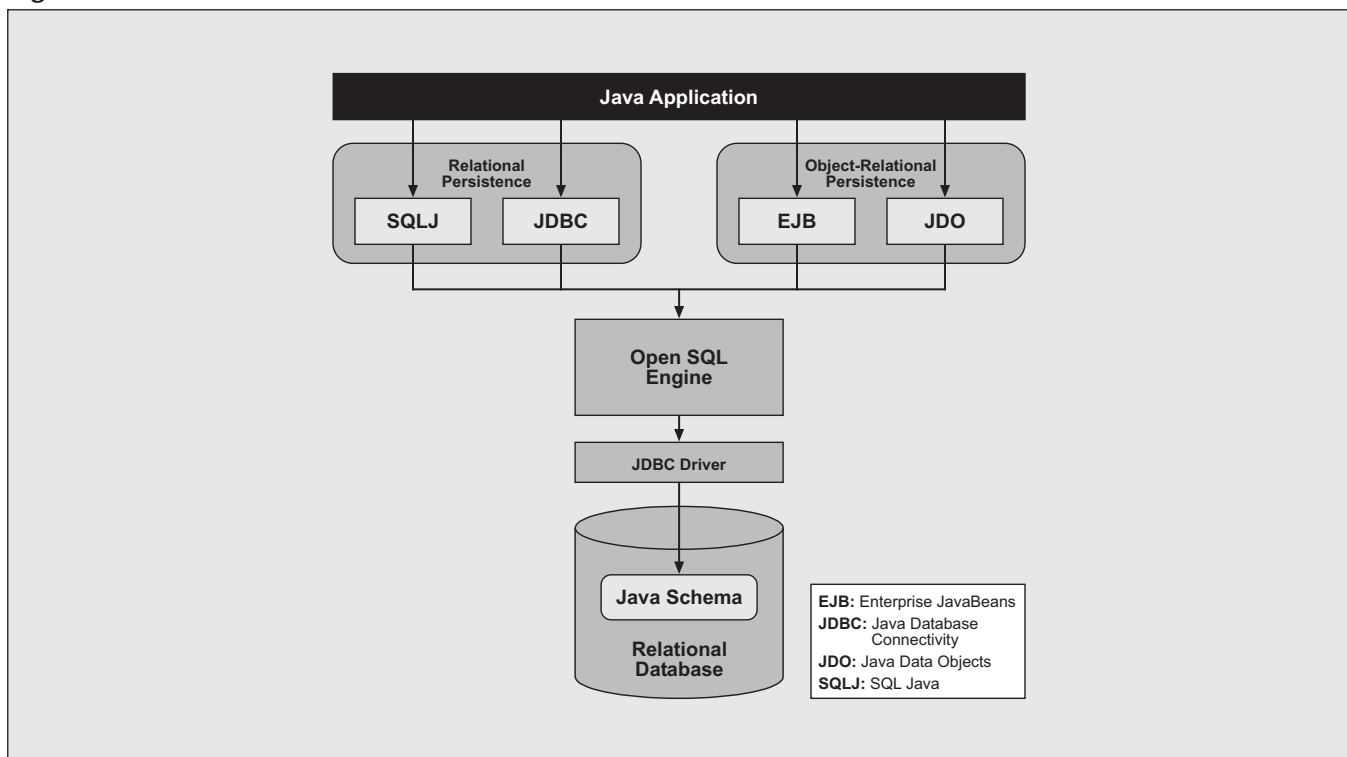
<sup>1</sup> The J2EE Engine is a certified J2EE (Java 2 Platform, Enterprise Edition) application server, fully compliant with the J2EE standard.

<sup>2</sup> The SAP Java Connector (JCo) is a framework that supports calling Java from ABAP and vice versa.

<sup>3</sup> The ABAP persistence layer offers a wide range of solutions, including Open SQL, Dynamic Open SQL, Native SQL, and Object Services. For more information, see the articles “Write Smarter ABAP Programs with Less Effort: Manage Persistent Objects and Transactions with Object Services” (January/February 2002) and “Enhanced ABAP Programming with Dynamic Open SQL” (September/October 2001).

Figure 1

The Java Persistence Framework



As part of a continuing series on the Java Persistence Framework,<sup>4</sup> this article provides an overview of the framework to serve as a foundation for other articles that explore its components in more depth. It discusses how the SAP J2EE environment supports persistence, while providing a glimpse into the world of Java persistence in general. Consequently, this article offers something for Java and ABAP developers alike. For maximum benefit, readers should be familiar with Java, especially Java Database Connectivity (JDBC), and have a basic understanding of the Java 2 Platform, Enterprise Edition (J2EE).

## What Is the Java Persistence Framework?

Let's begin by briefly reviewing the fundamental

building blocks of the Java Persistence Framework (as shown in **Figure 1**) before discussing each of them in more detail.

For maximum flexibility, SAP refrained from defining a proprietary API for the Java Persistence Framework. Instead, you write application code that uses APIs defined by well-established Java and J2EE standards — JDBC and SQL Java (SQLJ) for relational persistence, and Java Data Objects (JDO) and Enterprise JavaBeans (EJBs) for object-relational persistence. Therefore, from an application developer's perspective, the framework is largely invisible and works transparently behind the scenes.

These persistence APIs are built on the Open SQL Engine, which is the heart of the Java Persistence Framework. Regardless of which API is used, the Open SQL Engine handles and mediates all database accesses before the vendor-specific JDBC driver finally accesses the database. In this way, all persistence APIs can take advantage of added features in the Open SQL Engine for:

<sup>4</sup> See also "Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server" (January/February 2004).

- Enhanced performance by caching (a table buffer for table contents, and a statement pool for SQL statements)
- Scalability (a database connection pool for caching database connections)
- Diagnostics (a SQL trace tool)

Moreover, the Open SQL Engine comprises a database abstraction layer that supports writing persistence code once and running it on any database supported by SAP Web AS.

The database is accessed through the vendor's JDBC driver. However, the Java Persistence Framework is not tied to a particular database. You can freely choose the target database, either one that already exists, or one that is newly installed as part of installing SAP Web AS.

Finally, to support the various tasks related to persistence coding, the Java Persistence Framework works seamlessly with the SAP Java Development Infrastructure (JDI) and with SAP NetWeaver Developer Studio.<sup>5</sup>

## Understanding the ABAP and Java Schemas

Regardless of which programming language is used, persistent data in an SAP environment is ultimately stored in a relational database. The ABAP Engine of an R/3 system uses the ABAP Schema, which is the database schema that stores all persistent data for the system. This data includes business data for ABAP applications and internal data (such as ABAP sources, binaries, configuration data, and logging data) of the ABAP Engine. Until Release 6.20, the J2EE Engine stored its internal data (such as configuration data and

Java archives) in the file system, rather than in the database. Because a default database schema did not exist for the J2EE Engine, Java applications could use any schema in any database for storing business data.

Starting with Release 6.30, the J2EE Engine stores its internal data in a dedicated schema — the Java Schema — of an external relational database. This schema is configured during server installation and acts as the system database schema for the J2EE Engine. You typically create the Java Schema in the same database as the ABAP Schema.

Although the ABAP Engine and the J2EE Engine logically operate as one SAP Web AS, they cannot share the same database schema due to several non-trivial technical constraints:

- ABAP-based applications have historically used single-byte code pages for character data, while Java consistently uses Unicode.<sup>6</sup> Converting between these character representations is generally impossible without knowing the semantics of the stored data.
- Altering ABAP tables from Java would bypass the ABAP table buffers and table logging, and vice versa. This operation would require complex synchronization mechanisms between ABAP and Java.
- It is nearly impossible to achieve namespace coordination for database objects between the ABAP and Java worlds without breaking existing code. Historically, strict namespace coordination — even between ABAP projects — exists only rudimentarily.
- Accessing tables from both worlds contradicts the concept of encapsulation — i.e., bundling the data with the methods that operate on it. Even within either the ABAP or J2EE Engine, the preferred table access mechanism is via interfaces of the application that owns the table (rather than via direct table access).

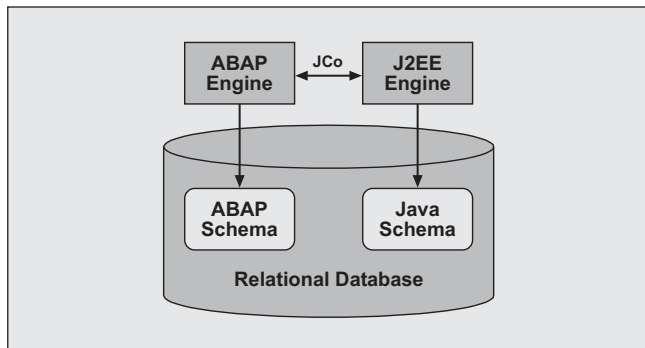
---

<sup>5</sup> Introduced with SAP Web AS 6.30, SAP NetWeaver Developer Studio is SAP's new integrated development environment (IDE) for Java-based applications. For a detailed introduction to using this tool, see the article "Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio" on page 3 of this issue.

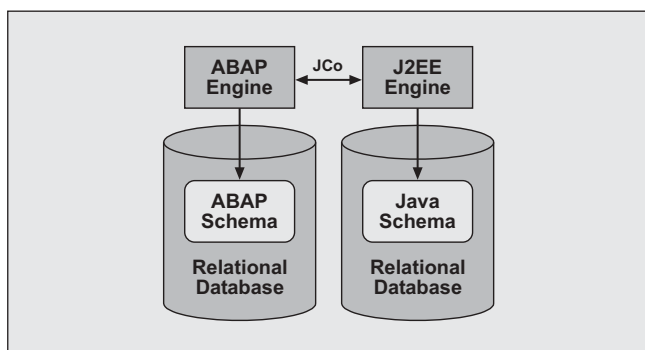
---

<sup>6</sup> Starting with Release 6.10, the SAP Web AS ABAP Engine supports both Unicode and non-Unicode.

**Figure 2** ABAP and Java Schemas in the Same Database



**Figure 3** ABAP and Java Schemas in Different Databases



Consequently, the Java Schema must be distinct from the ABAP Schema. You can install the schemas either in the same database (as shown in **Figure 2**) or in different databases (as shown in **Figure 3**), allowing for the scaling and tuning of each database individually. However, you must use a dedicated technology (such as JCo) for exchanging data between Java and ABAP components.

The J2EE standard does not specify where an application should store its persistent data. In principle, a single Java application could exchange data with several schemas in different databases. Of course, the J2EE Engine supports this flexibility. However, it is not practical for any Java application — whether an SAP application or a custom application — to fully exploit this flexibility.

Obviously you want the ability to assign an application to a specific database schema in order to

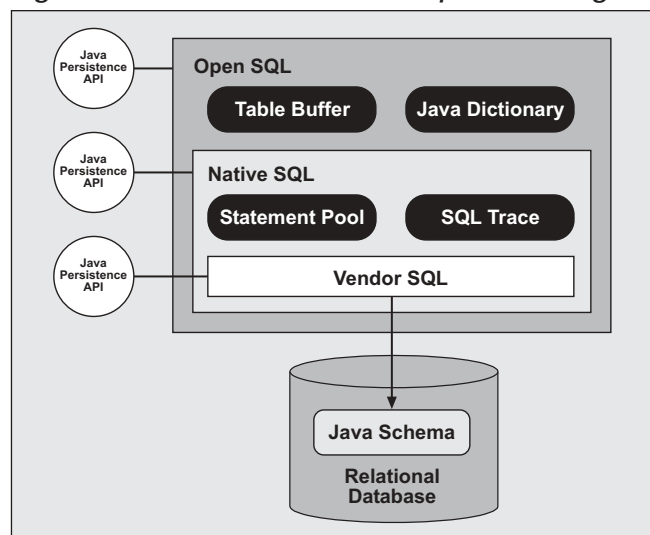
accommodate the needs of different applications or to achieve optimal scalability. In practical terms, a single Java application should use a single database schema (and thus a single database) for storing business data. To simplify database administration, all Java applications should be written so that they are able to share a common database schema. Consequently, all Java applications should use the default Java Schema.

Now that you have a basic understanding of the Java Persistence Framework, we can begin exploring its components — the Open SQL Engine and the Java Persistence APIs — in more detail.

## The Open SQL Engine

We begin our guided tour with the Open SQL Engine, which is the core of the Java Persistence Framework. This component handles every database access from within the J2EE Engine. As shown in **Figure 4**, the Open SQL Engine supports three levels of database access: Vendor SQL, Native SQL, and Open SQL. You access the different levels via JDBC using Open SQL/JDBC, Native SQL/JDBC, and Vendor SQL/JDBC, respectively. Each access level offers incremental features and performance enhancements over basic SQL access. We will examine the

**Figure 4** Access Levels of the Open SQL Engine



advantages of each level after introducing the fundamental concept of connection pools, which applies to all access levels.

### Accessing Data Sources via Connection Pools

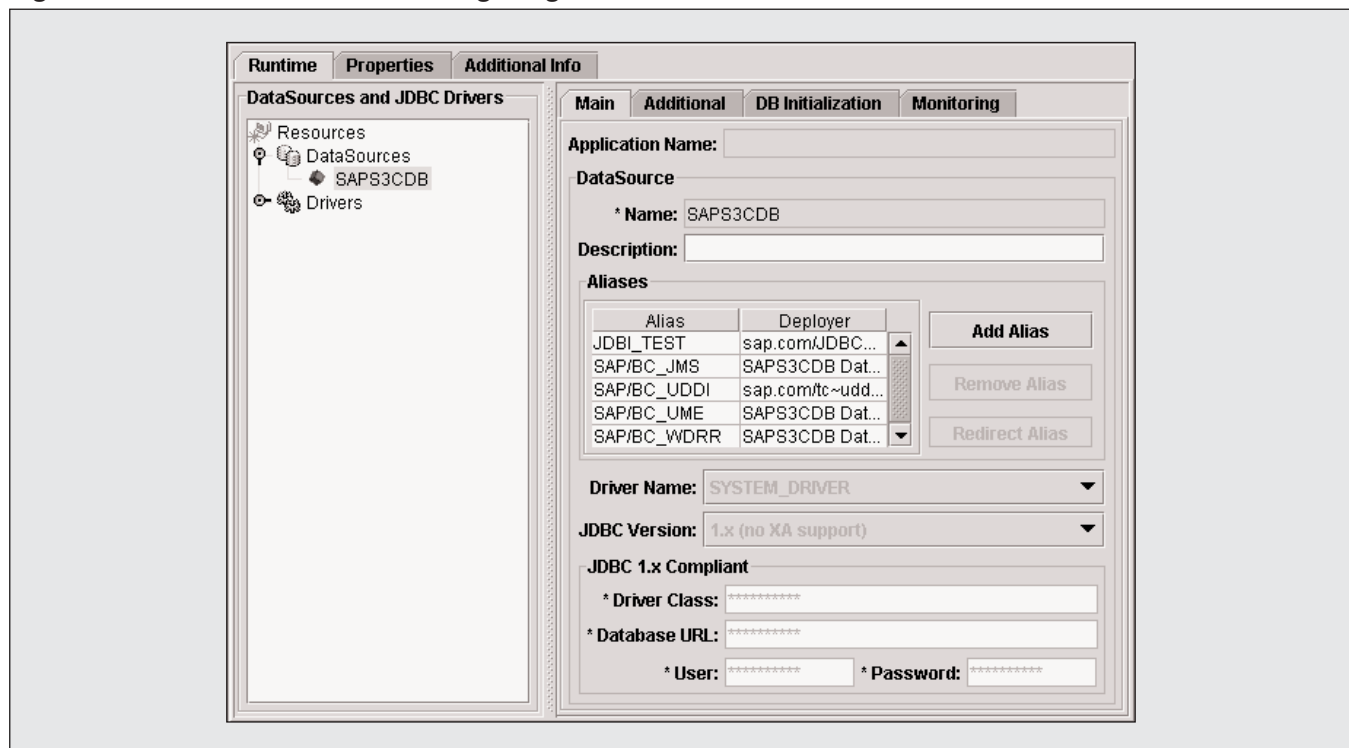
Regardless of the API and access level being used, the J2EE Engine accesses the database via a JDBC connection. Database connections are valuable resources, since a database server can have a limited number of database connections open at one time. An application should therefore hold a JDBC connection only for as long as needed and close it as soon as possible. However, opening a new JDBC connection is slow on any database. Not surprisingly, the J2EE Engine is capable of pooling open database connections. To take advantage of this feature, applications must request a JDBC connection from a connection pool (rather than directly from the database). Instead of being physically closed, connections are automatically returned to this pool for subsequent reuse.

As specified by the J2EE standard, the J2EE Engine exposes connection pools to applications through the JNDI (Java Naming and Directory Interface) Registry Service. To access a connection pool, an application looks up the name of the connection pool via JNDI. This lookup returns a *DataSource* object that allows it to obtain a JDBC connection via the *getConnection()* method. The following example shows how you would implement a JNDI lookup in your application:

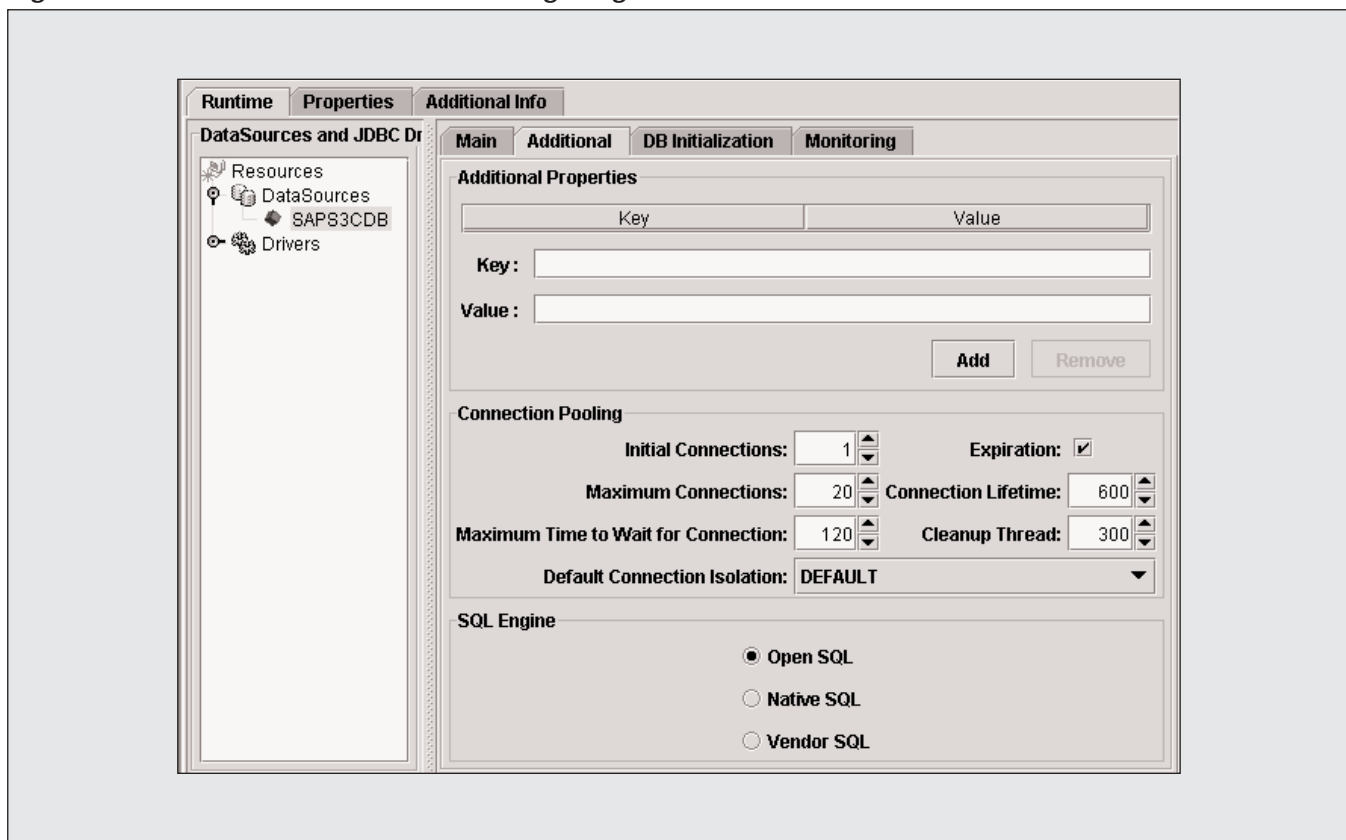
```
Context ctx = new InitialContext();
DataSource ds =
    (DataSource) ctx.lookup(
        "java:comp/env/jdbc/DEMO");
Connection conn = ds.getConnection();
```

The J2EE Engine can manage multiple connection pools for accessing the same or different schemas in one or more databases, even from different database vendors. The JDBC Connector Service creates and manages all connection pools. You use the SAP J2EE Engine Administrator (see **Figure 5**) to access this

**Figure 5** Configuring the JDBC Connector Service



**Figure 6** *Configuring a Connection Pool*



service. To create a connection pool, you specify the JNDI name, the database URL, the JDBC driver, and the database user for the schema. A connection pool for accessing the Java Schema is automatically configured during SAP Web AS installation.

### ***Configuring the Access Level of a Connection Pool***

The connection pool from which you obtain a connection determines its access level. The preconfigured connection pool for the Java Schema uses Open SQL. You must not change the setting for the preconfigured connection pool, which would affect all applications that use the pool. If you need to access a database schema other than the Java Schema, or for some reason you need to use Native or Vendor SQL, configure a separate connection pool for the desired database schema and set the access level accordingly.

By default, every connection pool created in the JDBC Connector Service uses Open SQL. You can change this setting in the SAP J2EE Engine Administrator (see **Figure 6**). Select *Services* → *JDBC Connector* → *Additional* and choose the intended access level in the *SQL Engine* frame. We'll look at the different access levels in detail next.

### ***Using the Vendor SQL Access Level***

For connection pools that are configured to use Vendor SQL, the JDBC Connector Service directly accesses the database vendor's JDBC driver. This access level adds no features to the JDBC driver other than connection pooling, and therefore delivers the lowest quality of service. In this mode, the SQL statements are passed to the database entirely unchanged. However, the application cannot access proprietary classes of the vendor's JDBC driver and



must use the classes and interfaces of the JDBC API. Vendor SQL is generally not recommended, because it lacks the features and enhancements offered by the other access levels.

### ***Using the Native SQL Access Level***

Like Vendor SQL, Native SQL allows SQL statements to be passed to the database unchecked. It therefore supports the entire proprietary SQL dialect of the database vendor. However, Native SQL offers the following advantages over Vendor SQL:

- A SQL statement pool that increases overall performance by caching prepared SQL statements
- The ability to trace SQL statements
- Workarounds for known JDBC driver bugs
- Added features to address acknowledged shortcomings of the JDBC interface

In contrast to Vendor SQL, Native SQL allows the application to access only the set of JDBC drivers supported by SAP Web AS.<sup>7</sup> Native SQL is recommended if you need to use features of a specific database vendor's SQL dialect or the vendor's JDBC driver. However, be aware that portability cannot be guaranteed. You cannot expect an application built on Native SQL/JDBC to run unchanged on a different database platform. In some situations, Native SQL might be useful for accessing an existing database containing non-SAP data.

Let's take a closer look at the advantages offered by Native SQL.

### **Statement Pool for Caching SQL Statements**

Before executing a SQL statement, the database engine must first prepare it. The database engine analyzes the statement, checks it for correctness, and determines a suitable execution plan based on current database statistics. These steps are time-consuming.

---

<sup>7</sup> SAP Web AS supports Microsoft SQL Server, Oracle, MySQL MaxDB, DB2 UDB, DB2 UDB for iSeries, and DB2 UDB for z/OS.

For a quickly executed statement, the preparation time could exceed the execution time. To avoid this expense, JDBC provides a *PreparedStatement* object, which is an abstraction for a prepared statement on the database. The database engine can execute this prepared statement multiple times without repeating the preparation steps.

Using prepared statements offers the advantage of increased application performance. However, remember that an application must release database resources such as connections, statements, and result sets as quickly as possible. So, using prepared statements also makes it difficult to simultaneously save on both resources and computational expenses. To address this conflict, Native SQL provides a statement pool that allows an application to save on statement preparation costs without keeping *PreparedStatement* objects open longer than absolutely necessary.

The statement pool is a cache for *PreparedStatement* objects. Caching is completely transparent to the application. The first time a statement is used, it is prepared and executed on the database as usual. Upon invoking the *close()* method, the Open SQL Engine puts the prepared statement object in the statement pool (rather than closing the statement). Subsequently, for a statement with equal statement text that needs to be prepared on the same database connection, the Open SQL Engine simply takes the statement from the statement pool. This approach prevents costly repeated preparation on the database.

### **SQL Trace for Performance Analysis**

Persistence coding typically consumes a considerable share of the overall execution time of an application. Consequently, the database code is a reasonable starting point when you are searching for performance bottlenecks. Especially in the world of object-relational persistence, you cannot spot problematic statements simply by analyzing the source code. Instead, you need a tool that inspects the SQL statements that are sent to the database. Unfortunately, the JDBC standard lacks an API for this type of performance analysis.

Figure 7

## SQL Trace Summary

**SAP SQLTrace Evaluation:**  
List for trace id 20030820132024015 from node 7564951.

Go to Trace Evaluation Refresh

Time	Duration [µs]	Jdbc method Id	No.	Result	Statement
13:20:37,072	5	Connection.setAutoCommit(boolean)			<a href="#">setAutoCommit(boolean) ( false )</a>
13:20:37,073	1345	PreparedStatement.executeQuery STMT:1462673996, RS:1452466764			SELECT DISTINCT "PID" FROM "UME_STRINGS" WHERE "PID" LIKE ? ESCAPE '\ ' AND "NAME...
13:20:37,075	346	ResultSet.next() RS:1452466764	1	true	<a href="#">next()</a>
13:20:37,076	4	ResultSet.next() RS:1452466764	1	false	<a href="#">next()</a>
13:20:37,077	2	ResultSet.close() RS:1452466764			<a href="#">close</a>
13:20:37,077	416	Connection.rollback()			<a href="#">rollback()</a>
13:20:37,078	5	Connection.setAutoCommit(boolean)			<a href="#">setAutoCommit(boolean) ( true )</a>
13:20:37,078	10	Connection.getMetaData()			<a href="#">getMetaData()</a>
13:20:37,079	3	Connection.setTransactionIsolation(int)			<a href="#">setTransactionIsolation(int) ( 1 )</a>
13:20:37,080	5	Connection.setAutoCommit(boolean)			<a href="#">setAutoCommit(boolean) ( false )</a>
13:20:37,083	1255	PreparedStatement.executeQuery STMT:889446988, RS:1018864204			SELECT "NAME", "ATTR", "VAL" FROM "UME_STRINGS" WHERE "PID" = ? AND "PIDH" = ? A...
13:20:37,085	731	ResultSet.next() RS:1018864204	9	true	<a href="#">next()</a>

To fill this gap, Native SQL provides the SQL Trace tool. It logs all SQL statements and JDBC method calls that access the underlying database, including their parameters and return values. The SQL Trace tool also records the execution time and duration of each statement, together with useful context information, such as the executing thread and the connection on which the statement was executed.

SAP provides a Web application for administering and activating the SQL Trace tool. From an HTML browser, enter the URL `http://<host>:<port>/SQLTrace`, where `<host>` and `<port>` are the host name and the HTTP port of the J2EE Engine. You also use this application for viewing the trace results, as shown in **Figure 7**. The summary shows the sequence of JDBC methods, their execution time and duration, as well as their associated SQL statement texts and return values.

Clicking on any SQL statement displays additional details, including context information, as shown in **Figure 8**. For example, for a JDBC statement executed by SQLJ, the trace results include the name of the Java program, the line number of the SQLJ statement, and the timestamp of the SQLJ source.


### Using the Open SQL Access Level

Built on top of Native SQL, Open SQL offers the highest quality of service in the Open SQL Engine. In contrast to the lower levels of Native SQL and Vendor SQL, Open SQL guarantees platform independence and ensures the portability of database access. To reduce the load on the database, Open SQL also contains a local cache called the “table buffer” for storing database contents.



Figure 8

## SQL Trace Details



**SQLTrace Evaluation: Trace Record Detail**  
 Displaying record 1 (trace id 20030820132024015, node 7564951).

Location	com.sap.sql.jdbc.direct.DirectPreparedStatement.executeQuery
Time	Wed Aug 20 13:20:37 CEST 2003
Thread	SAPEngine_Application_Thread[impl:3]_11
Database Id	SAPJAVDB?SAPJAVDB
Duration	1345
Prepared Statement Id	1462673996
Statement	<pre> SELECT   DISTINCT "PID" FROM   "UME_STRINGS" WHERE   "PID" LIKE ? ESCAPE '\' AND "NAMESP" = ? AND "NAMESPH" = ? AND "ATTR" = ? AND "ATTRH" = ? AND "UPPERVAL" = ? AND "UPPERVALH" = ? </pre>
Bind Parameters	<pre> P(1) = GRUP% P(2) = com.sap.security.core.usermanagement P(3) = -7697117895534560049 P(4) = uniquename P(5) = 10921812666271185 P(6) = ADMINISTRATORS P(7) = 7470780789779283230 </pre>
ResultSetId	1452466764
Table names	[UME_STRINGS]
Program name	
Line Number	0
SQLJ program generation time	0

## Meeting the Challenge of Portability

The database engines supported by SAP Web AS understand different SQL dialects with many syntactical variants. While the SQL standard defines the semantics for SQL statements, it leaves some room for interpretation. Vendor implementations of SQL can also be inconsistent with the SQL standard. Consequently, some statements (such as outer joins or the *ORDER BY* clause) behave differently on different databases. SQL data types and semantics also differ between database vendors.

In addition to differences and incompatibilities inherent to different database systems, we find a similar unpleasant variety in the behavior of different JDBC drivers, because the JDBC standard does not specify the semantics of a number of methods precisely enough.

These differences make it cumbersome to write a truly portable application that will run without changes on multiple database platforms. You typically face a considerable porting effort to migrate an application to each additional database platform. To remove this burden, the Open SQL layer contains a portability engine that allows you to write entirely platform-independent JDBC and SQLJ code. It consists of three building blocks: Open SQL/JDBC, the Open SQL Grammar, and the logical catalog.

## Open SQL/JDBC

As with the lower levels, Vendor SQL and Native SQL, you access Open SQL via JDBC — in this case, using Open SQL/JDBC. This access level supports truly database-independent JDBC code. The JDBC classes and interfaces that are exposed by Open

**Figure 9** *JDBC Data Types Supported by Open SQL*

Data Type	Value Range	Variable Length	Comparable
VARCHAR <sup>a</sup>	1 ... 127 characters	Yes	Yes
LONGVARCHAR	1 ... 1,333 characters	Yes	No
CLOB	1 ... 357,913,941 characters	Yes	No
BINARY	1 ... 255 bytes	No	Yes
LONGVARBINARY	1 ... 2,000 bytes	Yes	No
BLOB	1 ... 1,073,741,824 bytes	Yes	No
SMALLINT	-32,768 ... 32,767	—	Yes
INTEGER	-2,147,483,648 ... 2,147,483,647	—	Yes
BIGINT	-9,223,372,036,854,775,808 ... 9,223,372,036,854,775,807	—	Yes
FLOAT	±1.175E-37 to ±3.4E+38	—	Yes
DOUBLE	±1.0E-64 to ±9.9E+62	—	Yes
DECIMAL	Precision < 32, scale between 0 and 30	—	Yes
DATE <sup>b</sup>	yyyy-mm-dd	—	Yes
TIME <sup>b</sup>	hh:mm:ss	—	Yes
TIMESTAMP <sup>c</sup>	yyyy-mm-dd hh:mm:ss	—	Yes

<sup>a</sup> Empty strings or strings with trailing spaces are not allowed, although a single space may be stored.  
<sup>b</sup> The DATE and TIME types are not time-zone aware. They behave as if the values were stored in a string representation.  
<sup>c</sup> The TIMESTAMP type is time-zone aware. Its values are stored as UTC timestamps.

SQL/JDBC behave uniformly on all supported database platforms. To achieve this goal, the Open SQL Engine intercepts the JDBC method calls. Unfortunately, some methods cannot be implemented in a database-independent way. For example, the JDBC class *java.sql.DatabaseMetaData* contains many database-specific methods, such as *DatabaseMetaData.dataDefinitionCausesTransactionCommit()*. For these methods, Open SQL/JDBC throws an exception. Fortunately, most of these methods are irrelevant for developing business applications.

## Open SQL Grammar

The Open SQL Grammar defines the SQL statements that are accepted by Open SQL. In contrast to Open SQL for ABAP, it is not a proprietary SQL dialect. Instead, it is a subset of SQL statements specified by SQL 92.<sup>8</sup> It contains only statements that SAP has identified as syntactically recognized and executed

with equal semantics by the database systems of all supported database vendors.

The Open SQL Grammar comprises only queries and DML<sup>9</sup> statements. Because you can create, alter, and delete database tables using the table editor, DDL statements are not supported.<sup>10</sup> We discuss this graphical tool later in this section.

Open SQL supports the set of queries and DML statements defined in SQL 92 Entry Level.<sup>11</sup> It also supports some joined tables (*INNER JOIN* and *LEFT OUTER JOIN*, with some minor restrictions) as defined in SQL 92 Intermediate Level. Furthermore, Open SQL for Java supports a *SELECT ... FOR UPDATE* statement for selecting and exclusively locking a single row in a database table. As compared to Open SQL for ABAP, Open SQL for Java is more

<sup>9</sup> Data Manipulation Language (*INSERT*, *UPDATE*, and *DELETE*).

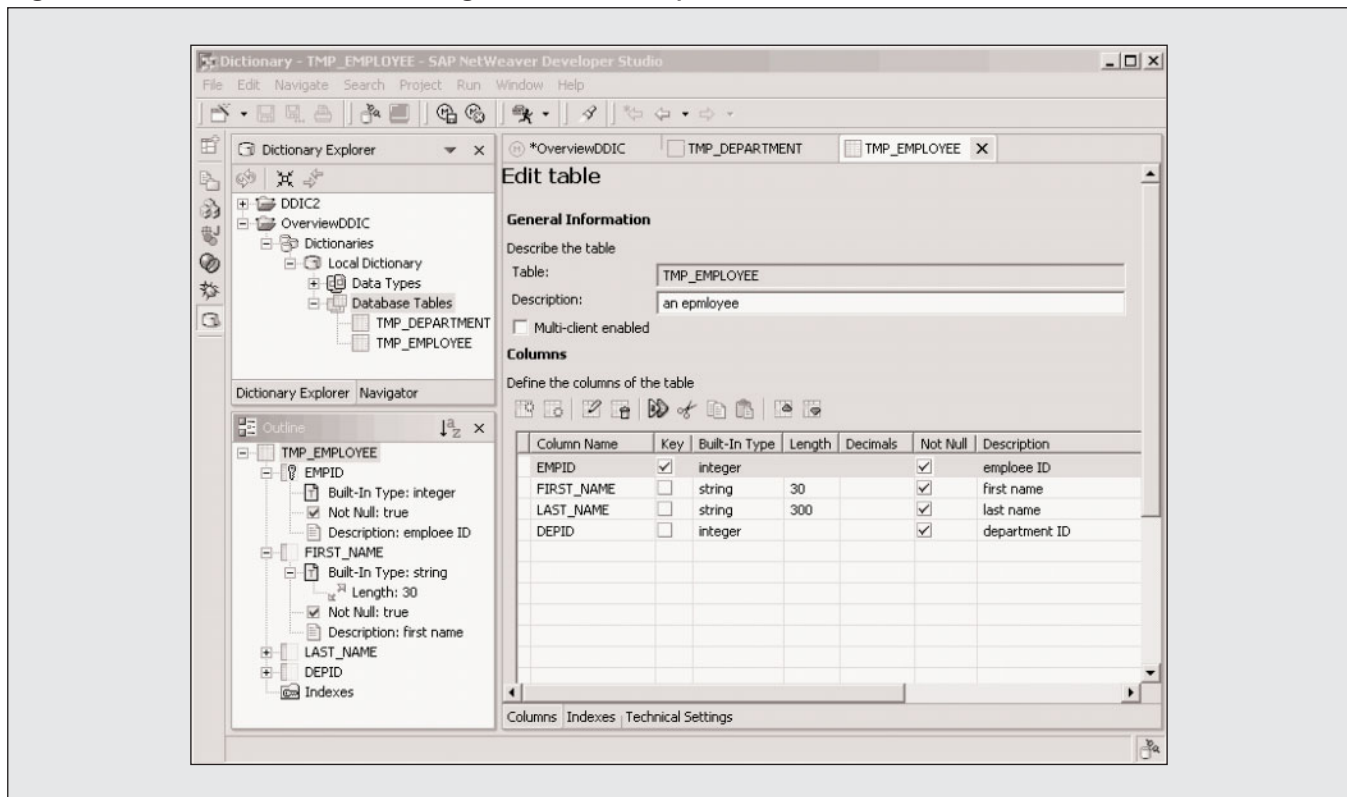
<sup>10</sup> Data Definition Language (for example, *CREATE TABLE* and *ALTER TABLE*).

<sup>11</sup> SQL 92 Entry Level is the lowest of three support levels (Entry Level, Intermediate Level, and Full Level) defined by the ISO/IEC 9075:1992 standard.

<sup>8</sup> SQL 92 refers to the Structured Query Language defined by the ISO/IEC 9075:1992 standard.

Figure 10

Creating a Table Description in the Table Editor



powerful and offers added features such as unions, arithmetic expressions, and using queries in the set clause of *UPDATE* statements.

The documentation that comes with SAP NetWeaver Developer Studio describes the Open SQL Grammar in detail.

## Logical Catalog

Different database systems also have different data types. To abstract from their characteristic properties, the Open SQL Engine contains a logical catalog (the Java Dictionary) for describing database tables using a database-independent type system. In this logical catalog, the JDBC types (as specified by *java.sql.Types*) denote the column types. **Figure 9** summarizes the JDBC types supported by Open SQL. The logical catalog is available at design time to components of SAP NetWeaver Developer Studio (such as the SQLJ Translator and the EJB Query Language Parser) and at runtime to the Open SQL Engine.

At design time, you create a table declaration in the table editor of SAP NetWeaver Developer Studio (see **Figure 10**). During deployment, this table declaration is transferred to the J2EE Engine and the physical database table is created or altered as needed. The database-independent table description is also stored in a metadata repository. At runtime, the Open SQL Engine performs semantic checks against this logical table description. This approach ensures that all SQL statements executed via Open SQL undergo the same checks on all databases.

## Table Buffer

All server nodes of a J2EE Engine cluster share a central database that they access via a local area network with limited bandwidth. Not surprisingly, this central database can easily become a performance bottleneck under the heavy workload of a production system. To increase overall performance, reducing database load and network communication as much as possible is highly desirable. The Open

Figure 11

## How the Table Buffer Works

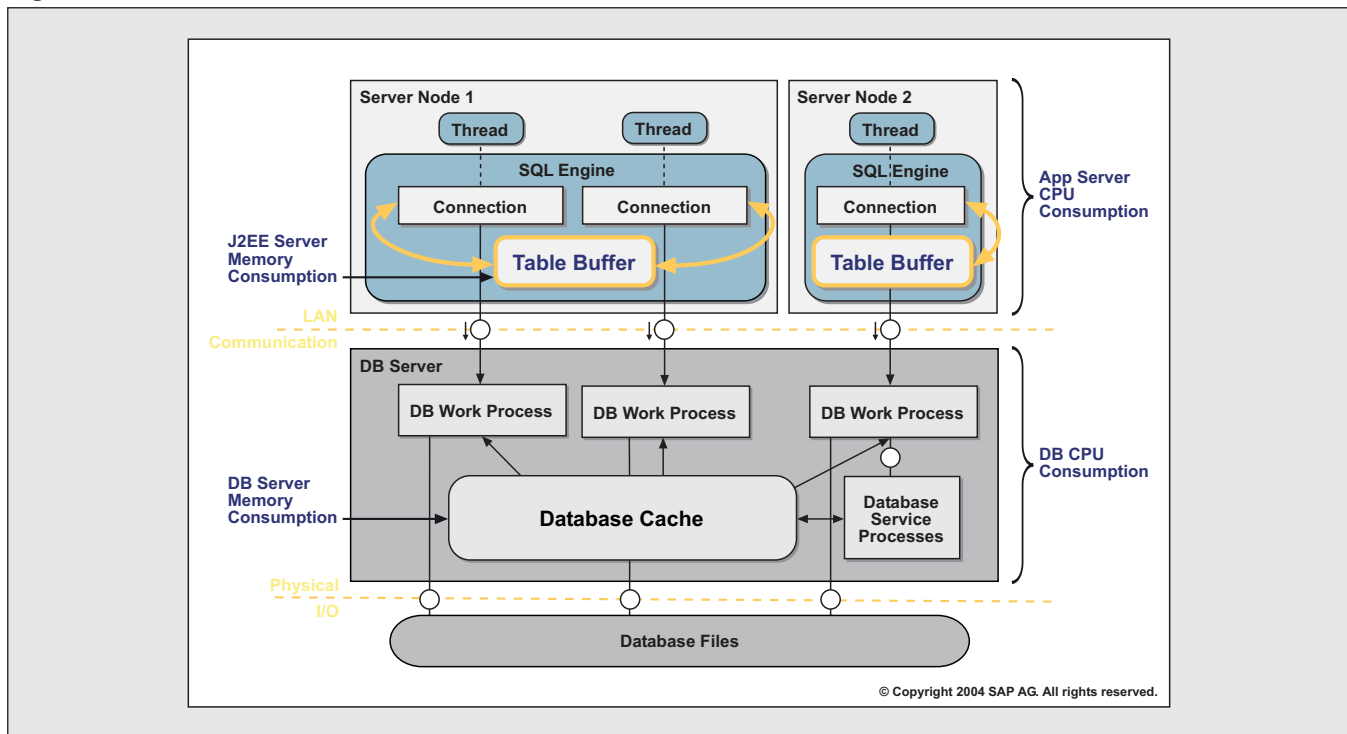


Figure 12

## Setting Table Buffer Properties in the Table Editor

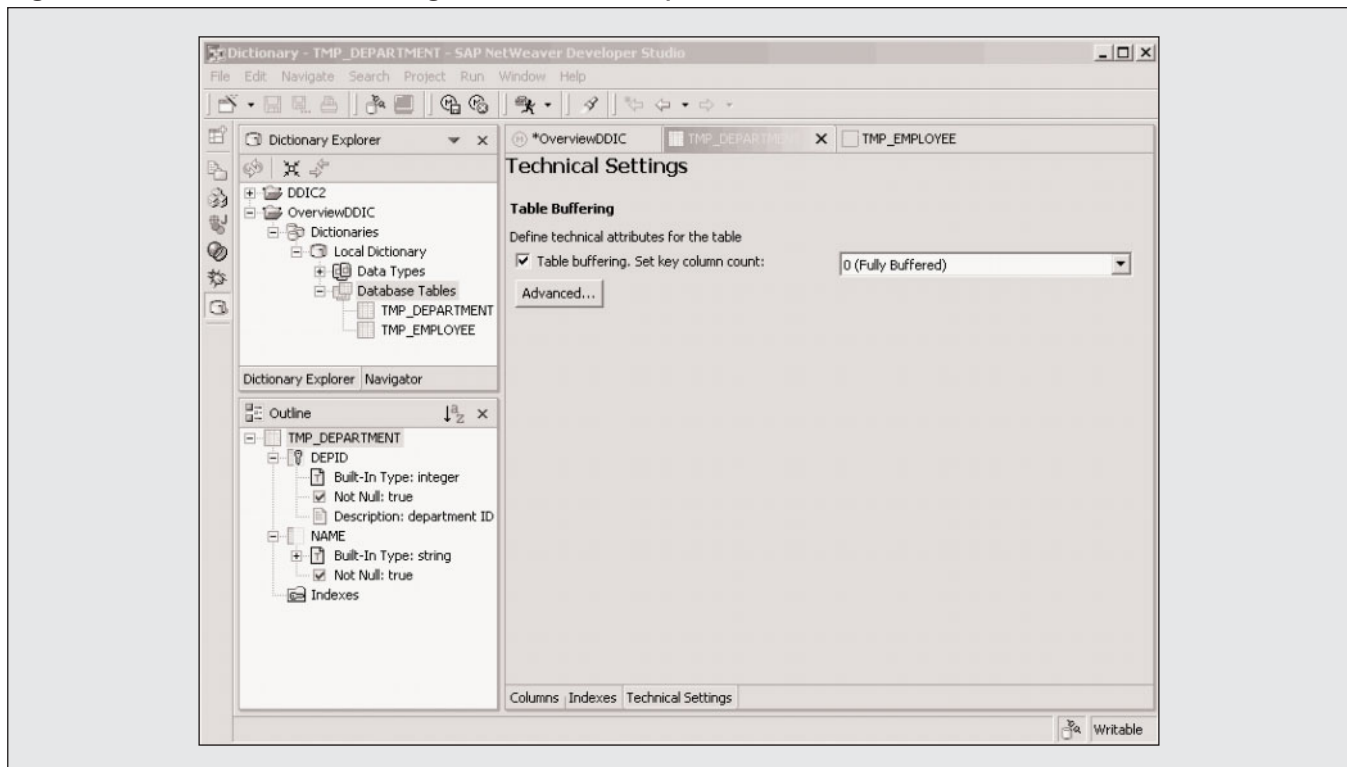


Figure 13

## Defining Buffer Granularity for the Table Buffer

all key fields				two key fields				one key field				no key fields			
key1	key2	key3	data	key1	key2	key3	data	key1	key2	key3	data	key1	key2	key3	data
001	A	2	some data	001	A	2	some data	001	A	2	some data	001	A	2	some data
001	A	4	some data	001	A	4	some data	001	A	4	some data	001	A	4	some data
001	B	1	some data	001	B	1	some data	001	B	1	some data	001	B	1	some data
001	B	3	some data	001	B	3	some data	001	B	3	some data	001	B	3	some data
001	B	5	some data	001	B	5	some data	001	B	5	some data	001	B	5	some data
002	A	1	some data	002	A	1	some data	002	A	1	some data	002	A	1	some data
002	A	3	some data	002	A	3	some data	002	A	3	some data	002	A	3	some data
002	A	6	some data	002	A	6	some data	002	A	6	some data	002	A	6	some data
002	A	8	some data	002	A	8	some data	002	A	8	some data	002	A	8	some data
002	B	1	some data	002	B	1	some data	002	B	1	some data	002	B	1	some data
002	B	2	some data	002	B	2	some data	002	B	2	some data	002	B	2	some data
002	B	3	some data	002	B	3	some data	002	B	3	some data	002	B	3	some data
002	C	0	some data	002	C	0	some data	002	C	0	some data	002	C	0	some data
002	D	1	some data	002	D	1	some data	002	D	1	some data	002	D	1	some data
002	D	5	some data	002	D	5	some data	002	D	5	some data	002	D	5	some data
003	A	2	some data	003	A	2	some data	003	A	2	some data	003	A	2	some data
003	A	3	some data	003	A	3	some data	003	A	3	some data	003	A	3	some data
003	A	6	some data	003	A	6	some data	003	A	6	some data	003	A	6	some data
003	B	2	some data	003	B	2	some data	003	B	2	some data	003	B	2	some data
003	B	4	some data	003	B	4	some data	003	B	4	some data	003	B	4	some data
003	C	5	some data	003	C	5	some data	003	C	5	some data	003	C	5	some data
003	D	2	some data	003	D	2	some data	003	D	2	some data	003	D	2	some data
003	D	6	some data	003	D	6	some data	003	D	6	some data	003	D	6	some data
003	D	8	some data	003	D	8	some data	003	D	8	some data	003	D	8	some data

SQL access level provides the table buffer for this purpose. **Figure 11** illustrates how the table buffer works.

The table buffer is a local cache on each server node of the J2EE Engine. It improves performance by buffering database contents on the application server. The J2EE Engine simply retrieves localized data from the table buffer, instead of fetching it from the database. Buffering is completely transparent to the application, requiring no special code to take advantage of the table buffer. You use SAP NetWeaver Developer Studio (see **Figure 12**) at design time to enable and configure the table buffer for individual tables.

Instead of caching the entire contents of a table, you can choose to store just the portion that is frequently accessed. You specify these portions (referred to as the “buffer granularity”) in the table editor. Granularity refers to the number of primary key columns that determines the key range of the table to be buffered — that is, the set of rows whose first  $n$  key columns have equal values. **Figure 13** shows the options for defining buffer granularity. For  $n = 0$ , the complete table is buffered — that is, its entire contents are loaded into the buffer upon first access to any data in the table. Otherwise, the table is divided into generic areas with equal values for the first  $n$  key columns. When a row in a generic area is accessed, the contents of the entire generic area are loaded into the table buffer.



With these options, you have the flexibility to define the key range in the most suitable way for the database table size and access pattern. For a large table where typical access is via a fully qualified key, you would choose *all key fields* as the buffering granularity. For small tables, you would more commonly specify the coarsest granularity by choosing *no key fields* in order to load the entire table contents.

Every server node has its own table buffer. Any modifications to the local buffer also invalidate the affected key ranges in remote buffers, ensuring data consistency across all server nodes of a J2EE Engine cluster. Consequently, buffer granularity affects buffer synchronization between J2EE Engines, in addition to determining the amount of cached records in the buffer.

Now that you understand the topology of the Open SQL Engine, we will continue our guided tour by focusing on how your application code works with it.

## The Java Persistence Application Programming Interfaces (APIs)

Relational databases are accessed in two fundamentally different ways:

- Via relational persistence, where you access database tables with SQL.
- Via object-relational persistence, where you manipulate persistent objects that are mapped to database tables by object-relational mapping (rather than accessing database tables directly). Changes to persistent objects are translated into database calls by an appropriate runtime component on the application server.

A wide range of Java solutions have come to support both programming paradigms. In keeping with this trend, an important design goal of the Java Persistence Framework is to allow you to choose the appropriate API for your needs. Therefore, the Java Persistence Framework supports the most prominent APIs:

- For relational persistence, Java Database Connectivity (JDBC) and SQL Java (SQLJ)

- For object-relational persistence, EJB entity beans with container-managed persistence (CMP) or bean-managed persistence (BMP), and Java Data Objects (JDO)

The Java Persistence Framework gives you the flexibility to mix APIs as needed in your Java applications. You might use any of the APIs exclusively or you might use several APIs in combination, even within the same database transaction if needed. Choosing the appropriate approach is clearly a complex task based only partly on the technology. You also need to consider factors such as the programming skills of the development team and possible reuse of the developed software components.

**Figure 14** compares the features of these APIs to help you in your assessment. Next, we will introduce each option and examine when and how to use them. While a single article could not possibly cover everything you need to know, these APIs are industry-standard technology. Many printed and online publications are available from SAP and other sources (see the sidebar on page 139 for some suggestions). In addition, look for other *SAP Professional Journal* articles that focus on programming with SQLJ,<sup>12</sup> EJB CMP, and JDO. At the end of this section we will also briefly look at how you can use ABAP as the backend of a Java application and the available APIs for managing local and distributed transactions.

### Relational Persistence APIs

With relational persistence, you access a database using SQL — the most widely used query language. You must manually write the SQL statements. On one hand, you can write highly efficient statements that are tailored for the specific needs of the application. On the other hand, you must possess strong Java and SQL skills. Because the SQL statements address specific database tables and column names, an application using relational persistence is programmatically tied to these names. They cannot be changed without modifying the application code.

<sup>12</sup> "Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server" (January/February 2004).

**Figure 14** *Comparison of Java Persistence APIs*

Feature	SQLJ	JDBC	JDO	EJB CMP
Persistent object type	N/A	N/A	Java class (lightweight)	Entity bean (heavyweight)
Supports dynamic queries?	No	Yes	Yes	No
Query language	SQL 92	SQL 92	JDOQL (Java-like)	EJBQL (SQL-like)
Supports aggregates?	Yes	Yes	No	Yes
Supports inheritance?	N/A	N/A	Yes	No
Performs design-time checks on queries and DML statements?	Yes	No	No	Partially supported*
Supported in SAP NetWeaver Developer Studio?	Yes	No	Planned for SAP NetWeaver 2005	Yes
Can access Open SQL?	Yes	Yes	Yes	Yes
Can access Native SQL?	No	Yes	Yes	Yes
Can access Vendor SQL?	No	Yes	Yes	Yes
Usable outside of EJB container?	Yes	Yes	Yes	No

\* Full support planned for SAP NetWeaver 2005.

Good programming practices dictate separating the database code from the business logic. With relational persistence, SQL statements can be executed anywhere in the Java code. Therefore, the relational persistence programming model does not enforce this separation; this is sometimes considered a drawback of relational persistence.

The Java world offers two well-established APIs for accessing a database with relational persistence — JDBC and SQLJ, both of which are supported by the Java Persistence Framework.

### ✓ **Note!**

*As you learned in the Open SQL section, it is generally very burdensome — if even possible — to write truly portable database code with SQL. However, if your JDBC or SQLJ application is using the Open SQL access level of the J2EE Engine, you can be sure that your database access implementation is portable between the databases and operating systems supported by SAP.*

## The JDBC API

Java Database Connectivity (JDBC) 2.1 is part of the Java 2 Platform, Standard Edition (J2SE), and is also the underlying database technology for J2EE. As the oldest and probably most widely used of the Java persistence APIs, JDBC is a SQL call-level API. When you work with JDBC, your Java program invokes JDBC methods to obtain a database connection, prepare SQL statements, execute statements on the database, and process the result set of queries. The JDBC classes and interfaces are contained in the Java packages `java.sql.*` and `javax.sql.*`.

JDBC methods take SQL statements as Java string arguments, and the SQL statements sent to the database are always constructed at runtime. The advantage of this approach is that it allows the posting of SQL statements that cannot be determined at design time. For example, you might want to construct a *WHERE* clause dynamically at runtime. The disadvantage is that the SQL statements cannot be checked at design time. Faulty statements might be detected only at runtime, possibly in a production system.

You can use JDBC with connection pools

## Example: Using the JDBC API for Relational Persistence

In this example, we use JDBC to insert a new department into the database table TMP\_DEPARTMENT. Assume that we have already obtained a data source (dataSource) using a JNDI lookup. In line 5, we obtain a JDBC connection (conn) from this data source. Next, we prepare an INSERT statement (lines 8-9) that contains two parameters as markers for the department ID and name. Before executing the statement, we bind the values to the parameters (lines 12-13). Then we execute the statement (line 16). Finally, we close the prepared statement object and the connection.

```

1 int depId = ...;
2 String depName = ...;
3
4 // obtain a connection
5 java.sql.Connection conn = dataSource.getConnection();
6
7 // prepare a statement
8 java.sql.PreparedStatement stmt = conn.prepareStatement(
9     "insert into TMP_DEPARTMENT (DEPID, NAME) values (?, ?)");
10
11 // bind the parameters
12 stmt.setInt(1, depId);
13 stmt.setString(2, depName);
14
15 // execute the statement
16 stmt.executeUpdate();
17
18 // free resources
19 stmt.close();
20 conn.close();

```

configured for any Open SQL access level (Open SQL, Native SQL, or Vendor SQL). SAP refers to these options as Open SQL/JDBC, Native SQL/JDBC, and Vendor SQL/JDBC, respectively:

- With Open SQL/JDBC, your SQL statements are checked against the Open SQL Grammar for portability. The JDBC methods are also checked for portability.
- With Native SQL/JDBC and Vendor SQL/JDBC, you can use the proprietary functionality of the database vendor's JDBC driver (at the cost of code portability).

The example above illustrates how to use JDBC for relational persistence.

## The SQLJ API

SQL/OLB,<sup>13</sup> also known as “SQL Java” (SQLJ) is an ISO<sup>14</sup> standard that defines a mechanism for embedding SQL statements directly in Java code. With SQLJ, you still write the SQL code, but you are freed from learning and applying the JDBC methods.

<sup>13</sup> Object Language Binding.

<sup>14</sup> For more information about the International Organization for Standardization (ISO), see [www.iso.org](http://www.iso.org).

In order to illustrate a simple query, we next show you how to determine which employees work in a specific department. As in the previous example, where we inserted a new department into the database table, we first obtain a connection from a data source (line 4). Again, we prepare the query as a SQL statement (lines 6-7). Executing the query returns a `ResultSet` object (line 10), which we process in a loop (lines 12-18). For every iteration, we process a single row of the result set. In lines 13-14, we use the `getString()` method to extract the column values from the result set. From the column values, we create a new `Employee` object and “do something” with the data.

```

1 int depId = ...;
2
3 // obtain a database connection
4 java.sql.Connection conn = dataSource.getConnection();
5
6 java.sql.PreparedStatement stmt = conn.prepareStatement(
7     "select FIRST_NAME, LAST_NAME from TMP_EMPLOYEE where DEPID = ?");
8
9 // execute the query
10 java.sql.ResultSet rs = stmt.executeQuery();
11
12 while (rs.next()) {
13     String firstName = rs.getString("FIRST_NAME");
14     String lastName = rs.getString("LAST_NAME");
15     Employee emp =
16         new Employee(depId, firstName, lastName);
17     doSomething(emp);
18 }
19 rs.close();
20 stmt.close();
21 conn.close();

```

A SQLJ framework generates the corresponding JDBC code.

You can use SQLJ only with connection pools configured for the Open SQL access level. Consequently, the SAP SQLJ solution is called “Open SQL/SQLJ.”<sup>15</sup>

In SAP NetWeaver Developer Studio, you can use the SQLJ Editor to implement Java classes that contain SQLJ clauses. A SQLJ clause starts with the

keyword `#sql` and contains the SQL statement enclosed in curly brackets `{... }`. The example on the next page illustrates the look and feel of SQLJ. If you are familiar with Open SQL for ABAP, you will notice that it is very similar.

The major advantage of SQLJ is that the SQLJ Translator checks the embedded SQL clauses at design time. The task list in SAP NetWeaver Developer Studio displays any detected errors so you can correct them without having to test your application on the server, which shortens the application development cycle. The SQLJ Translator checks the syntax of the SQL statements for conformance

<sup>15</sup> For more information on Open SQL/SQLJ, see the article “Achieving Platform-Independent Database Access with Open SQL/SQLJ — Embedded SQL for Java in the SAP Web Application Server” (January/February 2004).

## Example: Using the SQLJ API for Relational Persistence

In this example, we illustrate how to use SQLJ for the same business scenario used in the JDBC example on page 126 — inserting a new department into the database table TMP\_DEPARTMENT. First, we create a new SQLJ connection context instance, which represents a database connection (line 5). On this connection context, we execute the INSERT statement that contains the values to be inserted as embedded host variables (lines 8-9). Finally, we close the connection context (line 12).

```

1 int depId = ...;
2 String depName = ...;
3
4 // obtain a SQLJ connection context
5 Ctx ctx = new Ctx();
6
7 //
8 #sql [ctx] { insert into TMP_DEPARTMENT (DEPID, NAME)
9             values (:depId, :depName) };
10
11 // free the resources
12 ctx.close();

```

As in the JDBC example, we'll next demonstrate how to select all employees who work in a specific department, only here we will use SQLJ. In lines 4-5, we define a SQLJ result set iterator that serves as the typed handle to the result set. Again, we create a connection context (line 8). In lines 12-15, we assign the result of the query to the iterator (iter). We process the result set in a loop, one row per iteration (lines 16-21). Then, we extract the column values from the iterator using the named getter

with the Open SQL Grammar, which guarantees that your SQL code is portable across any database supported by SAP Web AS. The semantics of the SQL statements are checked against the table definitions in the logical catalog to ensure consistency. This check verifies that your application uses proper table and column names, and that the Java types match the defined data types for the database.

Another advantage of SQLJ over JDBC is the ability to write more compact code. You can use Java variables directly within SQL statements (as shown in the SQLJ example above) to achieve straightforward data exchange between Java and SQL in both directions.

However, SQLJ does not replace JDBC completely:

- Open SQL/SQLJ requires you to use the Open SQL access level. If you want to access the database using a Native SQL or Vendor SQL connection pool, you must use JDBC.
- SQLJ only supports static SQL statements. Generating SQL statements at runtime is not possible; you must use JDBC for dynamic SQL.

Fortunately, combining SQLJ and JDBC is straightforward, because the SQLJ framework is actually using JDBC behind the scenes. Open SQL/SQLJ and Open SQL/JDBC both use the same SQL checker and execute statements using the same runtime environment. Consequently, these two APIs can seamlessly interact with each other. SQLJ and JDBC can share a database connection and transaction, and work on the same statements and result sets.



methods `FIRST_NAME()` and `LAST_NAME()`. With these values, we create a new `Employee` object and “do something” with the data.

```

1 int depId = ...;
2
3 // define a iterator class
4 #sql iterator EmpIter
5     (int DEPID, String FIRST_NAME, String LAST_NAME);
6
7 // obtain a SQLJ connection context
8 Ctx ctx = new Ctx();
9 EmpIter iter = null;
10
11 // execute the query
12 #sql [ctx] iter = {
13     select FIRST_NAME, LAST_NAME
14     from TMP_EMPLOYEE
15     where DEPID = :depId};
16 while (iter.next()) {
17     // do something with the data
18     Employee emp =
19         new Employee(depId, iter.FIRST_NAME(), iter.LAST_NAME());
20     doSomething(emp);
21 }
22
23 // free the resources
24 iter.close();
25 ctx.close();

```

### ***The Object-Relational Persistence APIs***

Java is an object-oriented programming language. Business applications, however, typically store persistent data in relational databases. Object-relational persistence bridges the gap between the object world of Java and the relational world of the database. In object-relational persistence, you do not access database tables directly. Instead, the application works with persistent objects. Therefore, with object-relational persistence, development of persistent objects and the business application are decoupled.

For each class of persistent objects, you declare the object-relational mapping to the data store in an XML file. The object-relational mapping designates

the Java class fields that will be persistent, including the database table and column for storing the specified attributes. An underlying server-side persistence framework component (which is commonly called a “persistence manager”) uses the provided object-relational mapping rules and synchronizes the persistent objects with the data store. This operation is transparent to your application, which does not contain any persistence coding.

Although the persistence manager automatically generates the SQL statements, remember that you must understand the underlying database model in order to correctly specify the object-relational mapping for the persistent object. Furthermore, the current specifications do not offer the flexibility of JDBC and SQLJ. Notably, the JDO and EJB CMP

## Example: Using the EJB CMP API for Object-Relational Persistence

In this example, we illustrate how to use an EJB CMP entity bean to insert a new department into the database. Assume that instances of an entity bean named `DepartmentBean` represent department entities. In order to work with the bean, we first obtain the local home interface of the bean `DepartmentLocalHome` using a JNDI lookup (lines 5-6). In line 9, we invoke the `create()` method on the home interface to create a new bean instance.

```
1 int depId = ...;
2 String depName = ...;
3
4 // Look up the bean's home interface inside Web AS Java's namespace.
5 DepartmentLocalHome departmentHome = (DepartmentLocalHome)
6     ctx.lookup("java:comp/env/ejb/DepartmentBean");
7
8 // create a new department bean
9 departmentHome.create(depId, depName);
```

As with the JDBC and SQLJ examples on pages 126 and 128, we next determine which employees work in a department with a given ID. Assume that the employee entities are represented by the `EmployeeBean` with the local home interface `EmployeeLocalHome` and the business interface `Employee`. In lines 4-5, we obtain the local home interface of the `EmployeeBean` using a JNDI lookup. In order to execute the query that retrieves the desired set of employees, we simply execute the `findEmployeesByDepId()` method on the home interface (line 7), which returns a collection of `Employee` entities. In lines 9-11, we iterate over this collection and “do something” with the data.

query languages lack the power and expressiveness of SQL.

SAP Web AS supports the two most prominent APIs for relational persistence — EJB CMP and JDO. SAP Web AS 6.30 supports EJB CMP version 2.0 and JDO version 1.0. We’ll look at these APIs next (upcoming *SAP Professional Journal* articles will address these APIs and how they work with SAP Web AS in detail).

### The EJB CMP API

Enterprise JavaBeans (EJBs) are the building blocks of J2EE applications. In the J2EE architecture, they are the reusable components of which all applications are composed. Entity beans are specialized EJBs that represent data in a persistent data store.

To reduce the complexity of developing stable and scalable applications with EJBs, the J2EE server offers additional services such as transaction management and security. These services allow you to concentrate on developing the application’s business logic. The concept of EJB entity beans with container-managed persistence (CMP) offers an additional advantage. The J2EE server also takes control of all persistence-related activities, such as database connectivity and database queries or modifications, which relieves the entity beans of responsibility for relational database access.

In other words, EJB CMP is the J2EE approach for object-relational persistence. The EJB container and the underlying persistence manager use JDBC to synchronize the attributes of the entity bean with the database at runtime. Synchronization also applies to persistent relationships between entity bean instances.

```

1 int depId = ...;
2
3 // look up the home interface
4 EmployeeLocalHome employeeHome = (EmployeeLocalHome)
5     ctx.lookup("java:comp/env/ejb/EmployeeBean");
6
7 Collection col = employeeHome.findEmployeesByDepId(depId);
8
9 while (col.hasNext()) {
10     doSomething((Employee) col.next());
11 }

```

In order to run the entity bean, we provide the EJBQL statements with the method `findEmployeesByDepId()` inside the deployment descriptor of the `EmployeeBean`:

```

1 <query>
2     <description/>
3     <query-method>
4         <method-name>findEmployeesByDepId</method-name>
5         <method-params>
6             <method-param>int</method-param>
7         </method-params>
8     </query-method>
9     <ejb-ql>
10         SELECT Object(e) FROM Employee AS e WHERE e.depid = ?1
11     </ejb-ql>
12 </query>

```

Upon deployment of CMP entity beans, the necessary JDBC code is generated based on the information provided by the bean's deployment descriptor and object-relational mapping.

Proper implementation can be tedious and error-prone, but SAP NetWeaver Developer Studio offers wizard-driven entity bean definition. It generates the client interfaces and a template for the bean class, and then you simply edit the XML files and implement the bean class.<sup>16</sup> Once you have defined the entity bean, you can benefit from a pure Java approach when working with it in your applications. The example above illustrates this approach.

To search for specific entity beans or their persistent fields, you use specialized finder and select methods. EJB CMP comes with a query language for this purpose called “EJB Query Language” (EJBQL), which has a SQL-like syntax. It supports static queries only, which are checked by the framework at design time.

## The JDO API

The Java Data Objects (JDO) standard, which emerged from Java Specification Request (JSR) 12, has gained considerable popularity inside the Java community. Since JDO is not part of the J2EE standard, you can use it either inside or outside of EJBs.

The primary benefit of JDO is that you can declare plain Java classes as persistence-capable. Think of JDO as a natural extension of the Java

<sup>16</sup> See the article “Get Started Developing, Debugging, and Deploying Custom J2EE Applications Quickly and Easily with SAP NetWeaver Developer Studio” on page 3 of this issue for a detailed example of defining an entity bean.

## Example: Using JDO for Object-Relational Persistence

In this final example, we demonstrate how to use JDO to insert a new department into the database. Assume that the persistence-capable class `Department` represents a department. First, we create a new instance of the class `Department` (line 5). Next, we obtain a persistence manager instance from a persistence manager factory (line 8). The persistence manager holds the connection to the database. To insert the newly created `Department` object into the database, we simply invoke the `makePersistent()` method on the persistence manager (line 11). Finally, we close the persistence manager to release the associated resources (line 12).

```

1 int depId = ...;
2 String depName = ...;
3
4 // Create a new department instance
5 Department dep = new Department(depId, depName);
6
7 // A JDO Persistence Manager manages life cycle of persistent objects
8 javax.jdo.PersistenceManager pm = pmf.getPersistenceManager();
9
10 // Make the department persistent
11 pm.makePersistent(dep);
12 pm.close();

```

Next, we will determine which employees work in a specific department using a JDOQL query. Again, we first obtain a persistence manager (line 4). In line 6, we create a new JDO Query object using the `newQuery()` method of the persistence manager. It takes as arguments the candidate class `Employee` and a Java-like filter expression that contains a parameter ID. We declare the existence of this parameter (line 7) and execute the query with the value for this parameter (line 8). The result of the query is a collection of `Employee` objects, which we process in a loop and then “do something” with the data.

```

1 int depId = ...;
2
3 // A JDO Persistence Manager manages life cycle of persistent objects
4 javax.jdo.PersistenceManager pm = pmf.getPersistenceManager();
5
6 Query query = pm.newQuery(Employee.class, "department.depId == id");
7 query.declareParameters("int id");
8 Collection col = (Collection)query.execute(new Integer(depId));
9
10 while (col.hasNext()) {
11     doSomething((Employee)col.next());
12 }
13
14 query.close(col);
15 pm.close();

```

In addition to the persistence-capable class, we must also provide a related identity class containing the primary key and the XML-based metadata (not shown in this example).

programming language, adding persistence capability to objects that live inside a Java Virtual Machine. In addition to defining the persistence-capable class, you create an “identity class” that contains the primary key and, as with EJB CMP, the XML-based metadata. After compilation, the class files are modified in a post-processor step inside SAP NetWeaver Developer Studio by a byte-code enhancer. It replaces all references to the persistent attributes with method calls to the runtime component of the SAP JDO implementation. Transparently to the application, this runtime reflects actions performed on object instances, such as creating or deleting instances and reading or writing field values. The application can insert a new record into the database simply by creating an instance of a persistent class. As the example on the previous page illustrates, a call to the JDO method *makePersistent()* makes the particular instance persistent.

The example also shows how to query a database using JDO. In contrast to EJB CMP, JDO comes with a Java-like query language called “JDO Query Language” (JDOQL) that supports both static and dynamic queries. While queries can be created dynamically at runtime, the drawback, of course, is that they cannot be checked at design time.

Choosing between EJB CMP and JDO depends on the nature of the application:

- JDO is a more lightweight approach, because you implement a single persistence-capable class instead of a set of interfaces.
- EJB CMP is more widely used than JDO. If portability across different J2EE servers is an issue, EJB CMP might be a better choice.

In addition, consider the differences in functionality. As we have pointed out, only JDO supports dynamic queries. Furthermore, JDO supports what is referred to as “persistence by reachability.” Objects that can be directly or indirectly reached from a persistent object through a network of references are persisted automatically. Also, only JDO offers Java inheritance for persistent objects. In turn, EJB CMP supports a cascade-delete, which means that deleting a persistent object also causes the EJB CMP framework to delete all dependent objects.

For both APIs, you do not need to deal with transaction management — in other words, you do not directly commit or rollback the database transaction. As with relational persistence, you either delegate this task to an EJB session bean or use the distributed Java Transaction API (JTA) that is provided by the J2EE server.

## The EJB BMP API

EJB CMP entity beans always map to a relational database. If you need to access an alternative data store (such as a file system or an ABAP-based application) and prefer to use entity beans as an abstraction of the persistent objects, you can use EJB entity beans with bean-managed persistence (BMP). The EJB BMP API does not come with a framework that creates the persistence code. Instead, you must explicitly implement the code inside the bean class using JDBC, SQLJ, or JDO. You might also need to write specific code for accessing a proprietary system (such as an ABAP-based application) as the data store.

The persistence type (container-managed or bean-managed) is irrelevant to the client of an entity bean. Therefore, the code in the EJB CMP example on page 130 applies to working with EJB BMP as well.

When considering EJB BMP, you must weigh the increased effort against the achieved object-oriented view of persistent data. As with EJB CMP, you must learn and adhere to the EJB specification, as well as follow J2EE design patterns.

But what if you want to use ABAP as the backend of a Java application, and what APIs are available for managing local and distributed transactions? We’ll take a quick look at these issues next.

## Accessing ABAP Data from Java

As previously mentioned, ABAP and Java programs must not directly access the same database schema. However, Java and ABAP applications do not need to be completely isolated. You can combine ABAP and Java technology in your Web applications as needed.



**Figure 15**      *Accessing the User Transaction Interface via a JNDI Lookup*

```

javax.naming.Context ctx = new javax.naming.InitialContext();

// lookup the UserTransaction interface in JNDI
javax.transaction.UserTransaction ut =
    (UserTransaction) ctx.lookup("java:comp/UserTransaction");

[...]

// start a new user transaction
ut.begin();

// do something transactional

// commit the user transaction
ut.commit();

```

You may, for example, have a Web-based application that combines Java-based Web user interface components with ABAP-based business logic components. The tools of the extensive SAP connectivity technology suite (SAP JCo, SAP Java Resource Adapter, and Web services) support efficient and robust data exchange between Java and ABAP applications.

This architecture introduces an interesting alternative to JDBC-based persistence. A Java application could use ABAP for its data backend. In this scenario, Java components use JCo to access an ABAP function module, instead of accessing the database using persistence APIs. The function module then uses the ABAP persistence framework to communicate with the database. Thus a wide range of data managed by ABAP applications is also readily available to Java applications.

### ***APIs for Transaction Management and Locking***

Transaction management is necessary for achieving atomic and consistent business transactions. JDBC, SQLJ, and JDO provide standard APIs to demarcate

local transactions explicitly within the Java source code. You use them to start, commit, and rollback database transactions for the application using a single transaction resource. For an isolated application that runs entirely in a Web container, this type of transaction demarcation is sufficient.

With J2EE, the recommended approach to transaction management is to use the Java Transaction API (JTA). JTA, which is part of J2EE, allows applications to programmatically demarcate distributed (or global) transactions. A distributed transaction can combine multiple transactions on multiple data stores and other XA (Distributed Transaction) resources such as JMS (Java Message Service) to one logical transaction. A transaction manager, which is external to the data store, manages JTA transactions. SAP Web AS provides a transaction manager, supporting the two-phase commit protocol, as part of the Transaction Service. Application components can start, rollback, and commit distributed transactions through the simple interface *javax.transaction.UserTransaction*.

You access this interface via a JNDI lookup, as illustrated in **Figure 15**. Here, we first perform a

JNDI lookup to obtain a *UserTransaction* object. Then, we start a user transaction, perform some transactional operations, and finally commit the transaction.

To comply with the J2EE component model, larger business applications typically encapsulate the business logic into EJBs. An EJB is a server-side component that runs in the J2EE Engine. It lives in an EJB container, which manages its lifecycle and provides services for transaction management, security, and locking. EJBs allow you to refrain from demarcating transactions explicitly within the Java source code, because they support the superior concept of container-managed transaction demarcation. For each EJB method, you specify one of the standardized transaction attributes inside the bean's deployment descriptor, indicating whether the method can participate in a transaction and if it requires a transaction of its own. At runtime, the EJB container ensures that the method executes in the proper context as specified by these transaction attributes.

The Open SQL Engine only supports distributed transactions if the underlying resource manager — that is, the database vendor's JDBC driver — supports them. Until recently, not all database vendors supported by SAP Web AS supported distributed transactions. This situation implies that SAP applications cannot generally access multiple data stores in the scope of a distributed transaction. Further, you cannot access the ABAP Schema mediated by JCo and the Java Schema via the Java Persistence Framework in the same distributed transaction. Because the ABAP Engine is not capable of a two-phase commit, it cannot participate in a distributed transaction.

To solve the problem of concurrent access to the same data from multiple transactions, the SQL standard specifies several transaction isolation levels to define the required degree of isolation between concurrent transactions.<sup>17</sup> The technical implementation

of transaction isolation levels is usually based on database locks. The databases supported by SAP Web AS do not offer uniform semantics for locks. Therefore, the Java Persistence Framework uses the proven and tested SAP concept of logical locks, instead of physical database locks. The database operates at the lowest possible isolation level<sup>18</sup> and application components acquire locks on database objects via the Table Locking API, which is part of the Locking and Enqueue Service of SAP Web AS.

Now that you have a solid understanding of the Open SQL Engine and the Java Persistence APIs, let's take a look at how SAP helps you ensure the portability of your Java applications and persistence code through its support of the Data Source Alias concept, and how the SAP Java Development Infrastructure enables you to easily deploy Java applications in your SAP environment.

## ***The Data Source Alias***

The Open SQL access level solves the challenge of portable persistence code. But you need other mechanisms in order to achieve full platform independence of both application and data store configuration. The resource reference mechanism of J2EE<sup>19</sup> allows you to redirect servlet and EJB applications to another database connection pool without changing the Java sources. However, you must adjust the deployment descriptor appropriately and redeploy the application. Therefore, the resource reference approach is not feasible for building a ready-to-deploy enterprise application.

To overcome this limitation, SAP supports the concept of a Data Source Alias. Data Source Aliases are logical names that you assign to each configured database connection pool. You can assign multiple Data Source Aliases to each connection pool

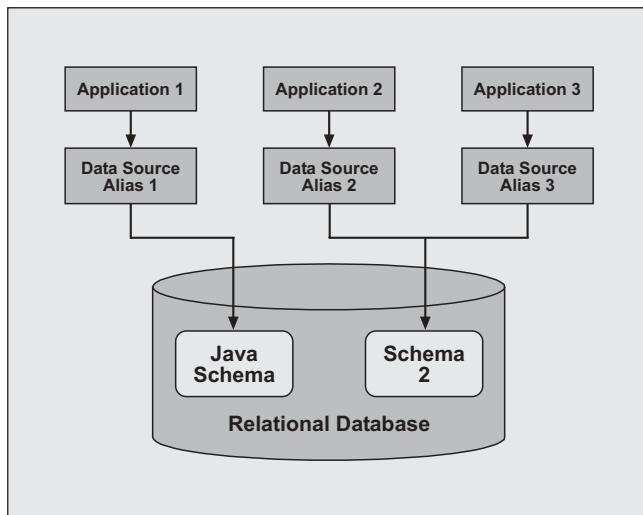
---

<sup>17</sup> Isolation levels provide a mechanism for protecting against undesired behaviors within a transaction. For example, the isolation level "committed read" protects a transaction from seeing uncommitted changes performed by a concurrent transaction.

<sup>18</sup> The lowest possible isolation level is a "consistent read" for Oracle and a "dirty read" for all other databases.

<sup>19</sup> For more information, refer to Section 5.5 of the J2EE 1.3 specification.

**Figure 16** Data Source Aliases Mapped to Different Schemas

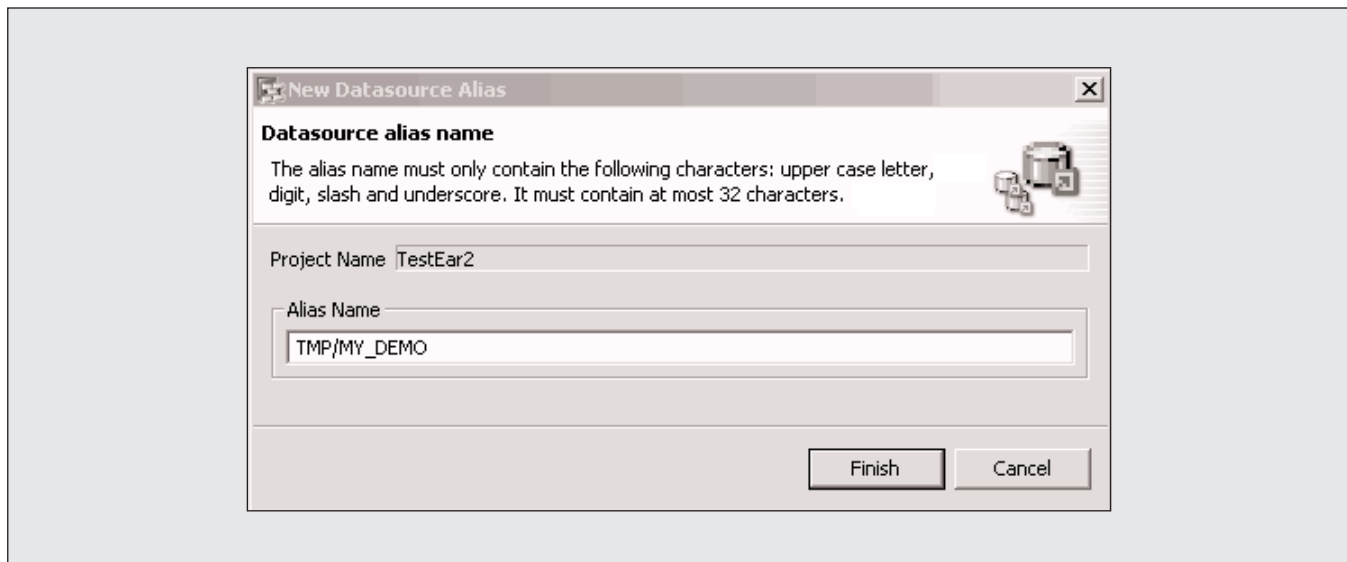


(as shown in **Figure 16**). Any Java-based application component (EJBs, servlets, and other components such as services, libraries, or Web Dynpro) can use the Data Source Alias for referring to the required database connection pool.

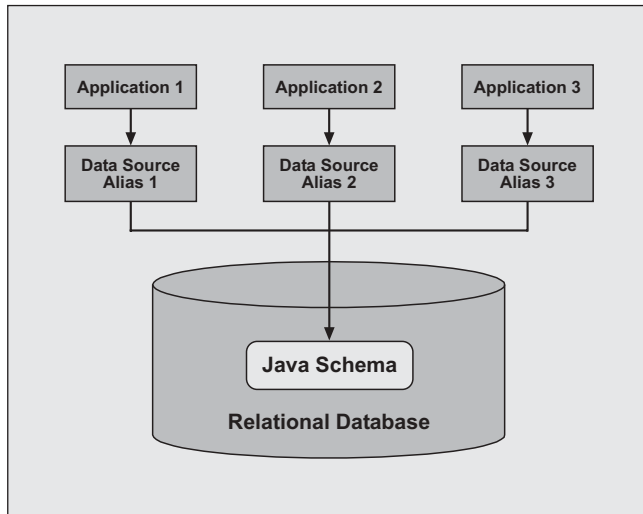
With Data Source Aliases, moving an application to another database connection pool is a straightforward task. The system administrator simply reassigns the Data Source Alias to the new target connection pool in the SAP J2EE Engine Administrator. This reassignment is fully decoupled from application development and deployment, and the application code remains untouched. You do not even need to redeploy it. Data Source Aliases are also unique and application-specific. You use the Name Reservation Service to allocate them, which we will explain later. Therefore, the system administrator can easily identify which application is accessing a particular connection pool.

You create a Data Source Alias in the J2EE Explorer view of SAP NetWeaver Developer Studio (see **Figure 17**). Select *new* → *META\_INF/datasourcealias.xml*. As discussed previously, all applications should store data in the same Java Schema and access it via the default preconfigured connection pool. Therefore, all defined Data Source Aliases are automatically assigned to this default connection pool (as shown in **Figure 18**).

**Figure 17** Defining a Data Source Alias



**Figure 18** *Data Source Aliases Mapped to the Same Schema*



## SAP Java Development Infrastructure

As SAP learned with ABAP, efficient mechanisms for deploying and transporting applications, application metadata, and dictionary data are indispensable. To accommodate these needs for Java applications, SAP Web AS includes the extensive SAP Java Development Infrastructure (JDI), which enables the SAP Software Logistics concepts for Java. The benefits of the JDI become particularly evident with complex business applications that require large developer teams, because it ensures well-structured and consistent software development and maintainability.

The JDI consists of a set of closely interacting tools and services for offline design, implementation, and development on a local PC, as well as online deployment and testing on a central server. In order to support offline development on a local PC, a major design goal was to shift as many decisions as possible to design time. While it is beyond the scope of this article to cover the JDI in detail, here we focus on what the JDI offers in support of persistence. Two tools are of particular importance:

- SAP NetWeaver Developer Studio is the common

frontend that connects you to the JDI services. You can develop, build, configure, deploy, execute, and remotely debug entire applications centrally from SAP NetWeaver Developer Studio.<sup>20</sup>

- The Software Deployment Manager resides inside the target SAP Web AS instance. It installs and updates a consistent set of software components in the target environment.

### The Java Dictionary

The J2EE Engine comes with a global metadata repository called the “Java Dictionary.” It contains definitions of two fundamental types of dictionary objects — data types and database tables. It provides the various design-time and runtime components (such as the table editor and the Open SQL Engine) with a uniform description of data types and database tables (the logical catalog).

As an abstraction of the proprietary database-specific data types, the Java Dictionary offers “build-in types” upon which all data types are ultimately based. You use these build-in types to assemble self-defined semantic data types — “simple types” and “structures.” They may additionally contain a description and a value range to be used for validity checks on the user interface. For consistency, you can also use these self-defined semantic data types in table definitions.

Regardless of which API is used, you must create and maintain all database tables in the Java Dictionary. It enables you to create and alter database tables, columns, and indexes in a database-independent manner. You must specify column types as Java Dictionary types (either build-in types or simple types) that have corresponding JDBC types.

Dictionary objects are organized into “dictionary projects” or “dictionary components.” You create and edit them in SAP NetWeaver Developer Studio

<sup>20</sup> See the article by Karl Kessler on page 3 of this issue for a detailed introduction to SAP NetWeaver Developer Studio.

using the table editor of the Dictionary Explorer view. The table editor does not create or alter the database tables directly. Instead, it generates schema archives that contain XML-based database table descriptions. In order to apply the table descriptions to the database, you must deploy them to SAP Web AS. During deployment (which can be initiated from within SAP NetWeaver Developer Studio), the Software Deployment Manager reads the table descriptions and creates or alters the target database appropriately.

### ***The Name Registration Service***

The JDI supports distributed development, allowing you to develop and test different applications independently. To avoid naming conflicts once applications are installed on a common system, the JDI includes a central Name Registration Service to ensure the worldwide conflict-free naming of database objects such as database tables, indexes, and SAP Data Source Aliases.

When you create a database object, the appropriate editor automatically contacts the Name Registration Service. It guarantees that the allocated names comply with a simple naming convention for database objects, which provides for hierarchical, company-specific, and business-area-specific names. Remember to use the same prefix for all names that are associated with a business area or for your company's data store.

## ***Benefits of Using the Java Persistence Framework***

Although somewhat complex, the Java Persistence Framework is designed to make life easier for Java developers in many ways:

- ☑ **Java standards compliance:** If you are familiar with one of the supported Java APIs, you can start developing or extending applications immediately. You don't need to spend time learning proprietary interfaces.
- ☑ **Object-relational persistence:** While possible, object-relational persistence (via JDO or EJB CMP) is not compulsory. Plus, object-relational persistence works seamlessly with relational persistence. For example, an application that typically uses JDO can also incorporate JDBC calls for database requests that cannot be expressed by the JDO query language.
- ☑ **Flexibility:** A technique or technology that is perfect for a simple servlet could be terribly wrong for a large-scale J2EE application. Therefore, the framework supports a wide spectrum of approaches to persistence in both the underlying technology and the development environment. You can base application design and implementation solely on requirements, rather than lack of support for certain solutions. The framework also enables a smooth transition from one programming model to another, allowing the coexistence of different APIs.
- ☑ **Consistent behavior of persistence components:** You should be able to expect that a particular database query implemented in different persistence approaches will be equally valid and lead to the same results. You can, of course, safely use your SQLJ SQL statement inside a JDBC call, because the syntactical and semantic checks are the same. But consistency and interoperability go far beyond validity checking. For example, any persistence API can reuse a prepared statement created in SQLJ. Likewise, any modification of a buffered table will be visible to all persistence APIs, independent of how the table was modified.
- ☑ **Portability:** The challenge of portability might not be immediately obvious, because Java (with its standards for persistence) seems to promise independence from database systems and vendors. Unfortunately, this premise is not completely valid. To bridge this gap, the Java Persistence Framework enables you to detect dangerous code at development time, instead of discovering discrepancies between database systems in a production system.



## Further Reading

For more information about the Java Persistence Framework, refer to the following offline documentation included with SAP NetWeaver Developer Studio:

- ☑ Open SQL Grammar and data types:

Navigate to *SAP Web AS for J2EE Applications* → *Reference Manual* → *Java Persistence Reference* → *Open SQL Reference*

- ☑ Open SQL/JDBC methods:

Navigate to *SAP Web AS for J2EE Applications* → *Reference Manual* → *Java Persistence Reference* → *Overview of the JDBC API*

- ☑ Java persistence:

Navigate to *SAP Web AS for J2EE Applications* → *Development Manual* → *Developing Business Logic* → *Java Persistence*

- ☑ **Scalability:** The Open SQL Engine provides enhancements that improve the scalability of business applications, most of which are largely invisible to application developers. Applications will continue to benefit from SAP improvements in this area, without requiring explicit code to implement them.

- ☑ **Tight integration with the JDI:** The development cycle of a persistent application — from development to testing, and then through packaging, delivery, and maintenance — is part of the SAP Java Development Infrastructure. Development support ranges from integrated editors (such as those for SQLJ applications) to the final packaging of a deliverable software component.

- ☑ **Simplified database object creation:** Creating or changing database tables or indices for your application is almost trivial with the integrated table editor. All the work, such as assuring database independence, happens transparently behind the scenes.

- ☑ **Faster development:** You test J2EE applications by deploying them to a J2EE application server, where the application is executed. This process can be tedious, especially for large applications. The Java Persistence Framework accelerates development by detecting incorrect or non-portable coding at the earliest possible stage. For example, the SQLJ editor can detect that a SQL statement accesses a nonexistent table column immediately after you enter the statement.

## Summary

The Java Persistence Framework is an essential part of the technological foundation of SAP Web AS 6.30. It complies with established Java standards, and at the same time offers software developers the flexibility to choose the type of persistence that best suits their purposes. Rather than enforcing any particular programming paradigm (such as object-relational persistence), the Java Persistence Framework supports combining different approaches. Interoperability is achieved via a core central component, the Open SQL Engine, which is the foundation of all exposed Java persistence APIs. This engine ensures the consistent behavior of all persistence components and approaches, and includes enhancements that enable the portability and scalability of business applications.

The Java Persistence Framework is firmly embedded in the Java Development Infrastructure, resulting in efficient software design and development. With just a few mouse clicks, you can create and deploy database objects (such as tables and indices) in a database-independent manner. Programming errors and nonportable code are recognized well before the application is deployed. Regardless of the scale or scope of your application, simple servlets and complex J2EE applications alike will benefit from the Java Persistence Framework.

*Katarzyna Fecht studied computer science and mathematics at the Warsaw University in Poland. In 2001, she joined SAP as a member of the J2EE Server group, where she coauthored and taught the internal classroom training for the J2EE and EJB technologies. Katarzyna currently belongs to the SAP Web Application Server product management team, where she focuses on information rollout for the Java Persistence and J2EE technologies. You can contact her at [katarzyna.fecht@sap.com](mailto:katarzyna.fecht@sap.com).*

*Adrian Görler studied physics at the Ruprecht-Karls-University in Heidelberg, Germany, where he specialized in computational biophysics. He received his doctorate at the Max-Planck-Institute for Medical Research, Heidelberg, and did post-doctoral research work in various international laboratories. In 1999, Adrian joined SAP and became a member of the Business Programming Languages group, where he worked as a kernel developer responsible for the implementation and maintenance of Open SQL as well as Native SQL in ABAP. Since 2001, Adrian has been a member of the Java Server Technology group, where he has been working on the implementation of Open SQL for Java, especially Open SQL/SQLJ. He can be reached at [adrian.goerler@sap.com](mailto:adrian.goerler@sap.com).*

*Jürgen G. Kissner received his doctorate in theoretical physics at the University of Manchester (UK). In 1996, he joined the SAP Server Technology development team as a member of the Database Interface group, where he worked on high availability, in particular database reconnect and switchover, and the integration of parallel database systems with R/3. Jürgen is currently responsible for the connection and transaction handling aspects of SAP Web Application Server, including multiple database connections in a heterogeneous database landscape. His second development focus is the ABAP and Java table buffer, especially the buffer manager and the SQL statement processor, and the Native SQL layer of the Open SQL engine. He can be contacted at [juergen.kissner@sap.com](mailto:juergen.kissner@sap.com).*