

Chapter 3

DB-Gateways



Outline

- Coupling DBMS and programming languages
 - approaches
 - requirements
- Programming Model (JDBC)
 - overview
 - DB connection model
 - transactions
- DB-Gateways
 - architectures
 - ODBC
 - JDBC
- SQL/OLB – embedded SQL in Java
- Summary



Static Embedded SQL

- Static SQL queries are embedded in the programming language
 - programming language is "extended", using **prefix** for SQL operations
 - cursors to bridge so-called impedance mismatch (**sets** of results)
 - **host variables** for query parameters and results
- Example:
exec sql declare c **cursor for**
 SELECT empno FROM Employees WHERE dept = :deptno_var;
exec sql open c;
exec sql fetch c **into** :empno_var;
- Preprocessor/precompiler converts SQL into function calls of the programming language
 - potential performance advantages (early query compilation)
 - vendor-specific precompiler and target interface
 - function calls invoke DBMS-specific APIs
 - initial source code is (usually) portable, but code resulting from precompilation is not!

Dynamic Embedded SQL

- SQL queries can be created dynamically by the program
 - character strings interpreted as SQL statements by an SQL system

- Example:

```
strcpy(stmt, "SELECT empno FROM Employees WHERE dept = ?");  
exec sql prepare s1 from :stmt;  
exec sql declare c cursor for s1;  
exec sql open c using :deptno_var;  
exec sql fetch c into :empno_var;
```

- Preprocessor is still required
 - only late query compilation
 - same drawbacks regarding portability as for static embedded

Call-Level Interface (CLI)

- Standard library of **functions** that can be linked to the program
- Same capabilities as (dynamic) embedded
 - SQL queries are string parameters of function invocations
- Example:

```
strcpy(stmt, "SELECT empno FROM Employees WHERE dept = ?");  
SQLPrepare(st_handle, stmt, ...);  
SQLBindParam(st_handle, 1, ..., &deptno_var, ...);  
SQLBindCol(st_handle, 1, ..., &empno_var, ...);  
SQLExecute(st_handle);  
SQLFetch(st_handle);
```

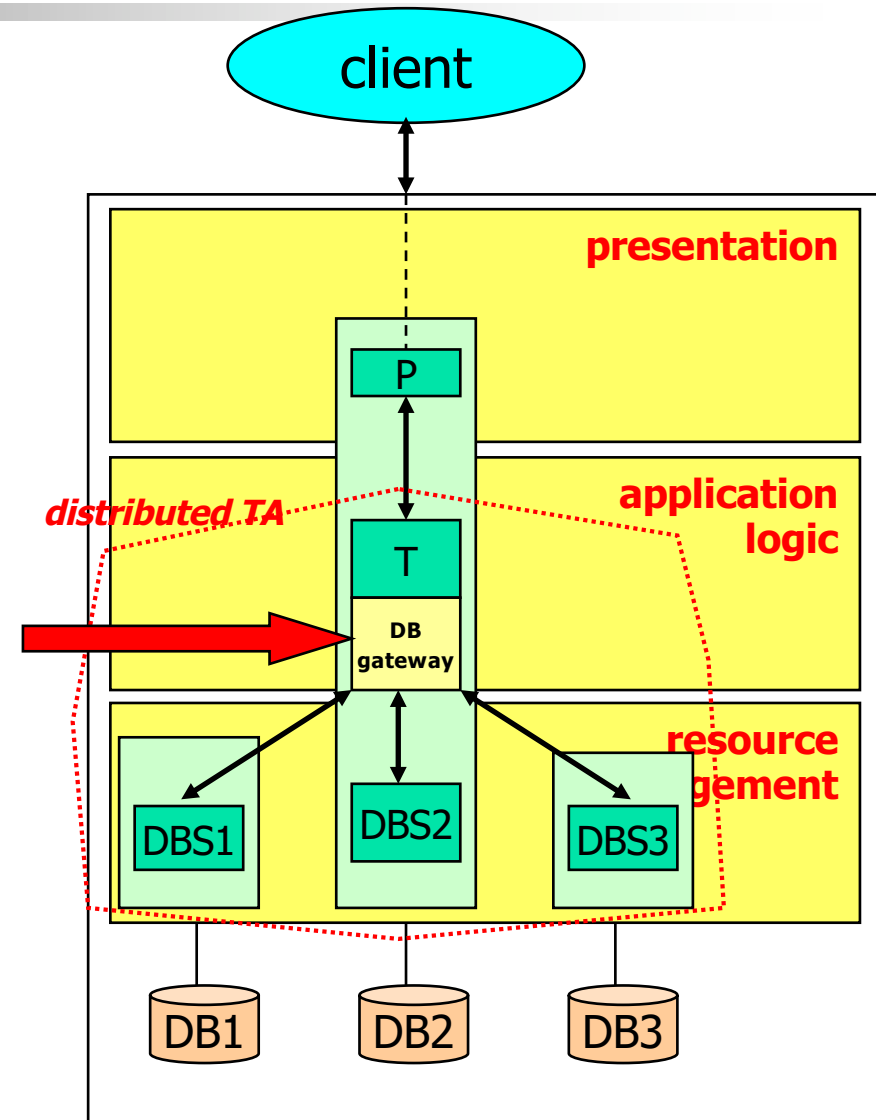
- Avoids vendor-specific precompiler, allows to write/produce binary-portable programs

Standard Call Level Interfaces - Requirements

- Uniform database access
 - query language (SQL)
 - meta data (both query results and DB-schema)
 - Alternative: SQL Information Schema
 - programming interface
- Portability
 - call level interface (CLI)
 - no vendor-specific pre-compiler
 - application binaries are portable
 - but: increased application complexity
 - dynamic binding of vendor-specific run-time libraries
- Dynamic, late binding to specific DB/DBS
 - late query compilation
 - flexibility vs. performance

Additional Requirements for DB-Gateways

- Remote data access
- Multiple simultaneously active DB-connections within the same application thread
 - to the same DB
 - to different DBs
 - within the same (distributed) transaction
- Simultaneous access to multiple DBMS
 - architecture supports use of (multiple) DBMS-specific drivers
 - coordinated by a driver manager
- Support for vendor-specific extensions

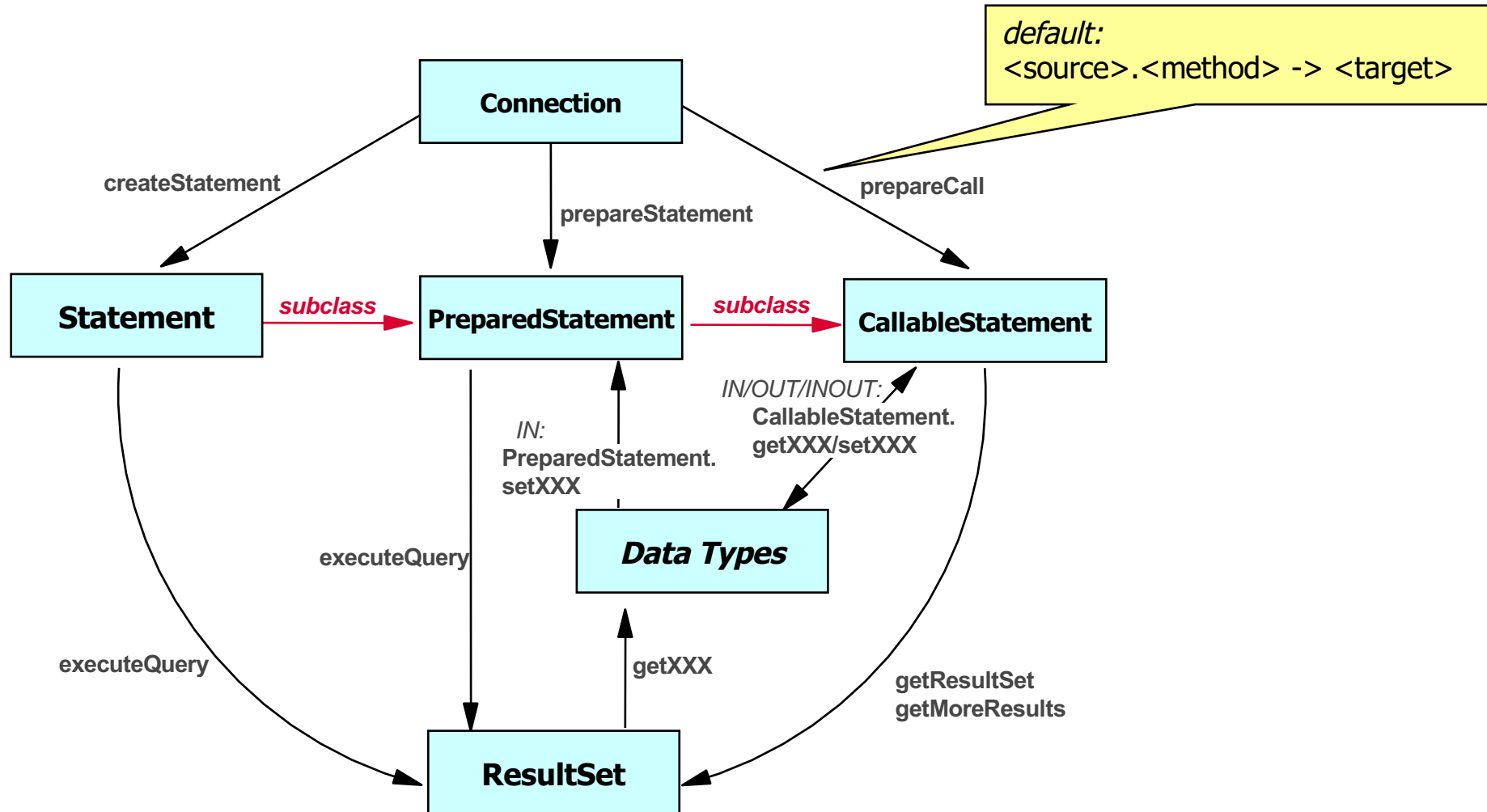


Historical Development

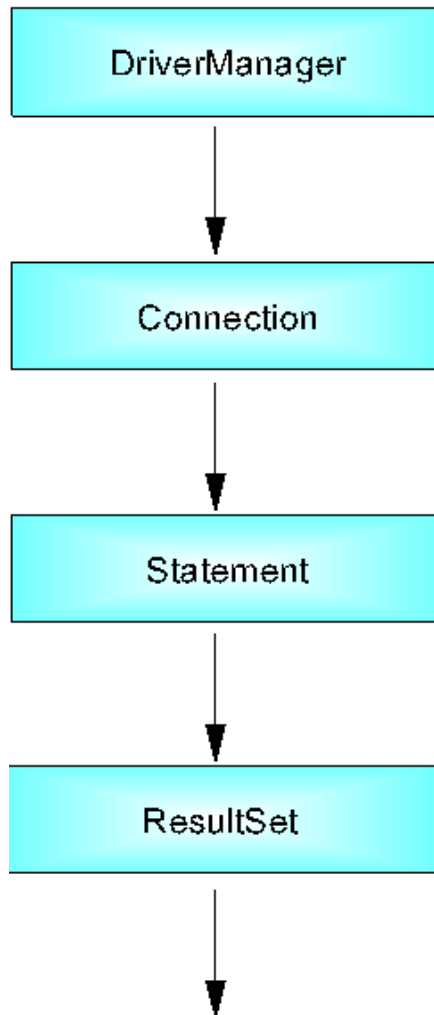
- ODBC: Open Database Connectivity
 - introduced in 1992 by Microsoft
 - quickly became a de-facto standard
 - ODBC drivers available for almost any DBMS
 - "blueprint" for ISO SQL/CLI standard
- JDBC
 - introduced in 1997, initially defined by SUN, based on ODBC approach
 - leverages advantages of Java (compared to C) for the API
 - abstraction layer between Java programs and SQL
 - current version: JDBC 4.1 (July 2011)

| |
|---------------------------------------|
| Java application |
| JDBC 4.1 |
| SQL-92, SQL:1999, SQL:2003 |
| (object-) relational DBS |

JDBC – Core Interfaces



Example: JDBC



```
String url = "jdbc:db2:mydatabase";
...
Connection con = DriverManager.getConnection(url, "desso", "pass");
```

```
String sqlstr = "SELECT * FROM Employees WHERE dept = 1234";
Statement stmt = con.createStatement( );
```

```
ResultSet rs = stmt.executeQuery(sqlstr);
```

```
while (rs.next() ) {
    String a = rs.getString(1);
    String str = rs.getString(2);
    System.out.print(" empno= " + a);
    System.out.print(" firstname= " + str);
    System.out.print("\n");
}
```

JDBC – Processing Query Results

- ResultSet
 - getXXX-methods
 - scrollable ResultSets
 - updatable ResultSets
- Data types
 - conversion functions
 - streams to support large data values
 - with JDBC 2.0 came support of SQL:1999 data types
 - LOBS (BLOBS, CLOBS)
 - arrays
 - user-defined data types
 - references

JDBC – Additional Functionality

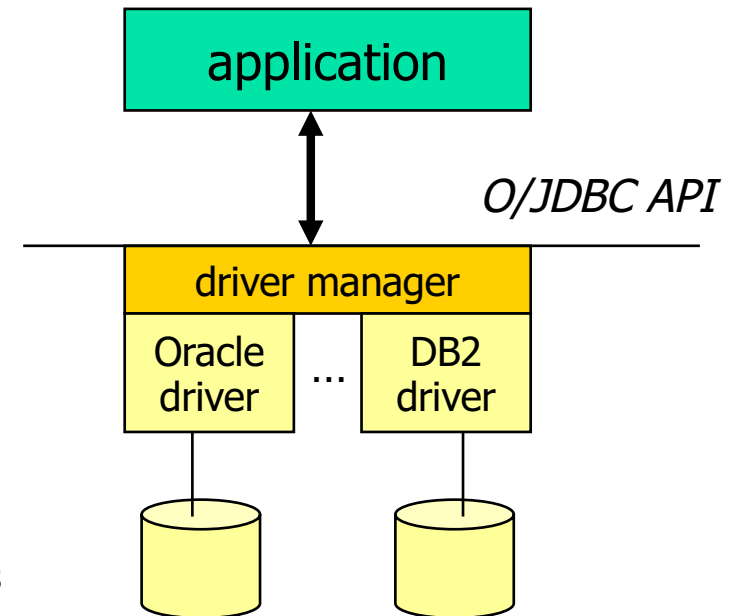
- Metadata
 - methods for metadata lookup
 - important for generic applications
- Exception Handling
 - SQLException class (hierarchy) carries SQL error code, description
 - Integrated with Java (chained) exception handling
- Batch Updates
 - multiple statements can be submitted at once to improve performance
- RowSets
 - Can hold a (disconnected) copy of a result set
 - Modifications can be “buffered” and explicitly synchronized with the database later
- ...

Transactions in JDBC

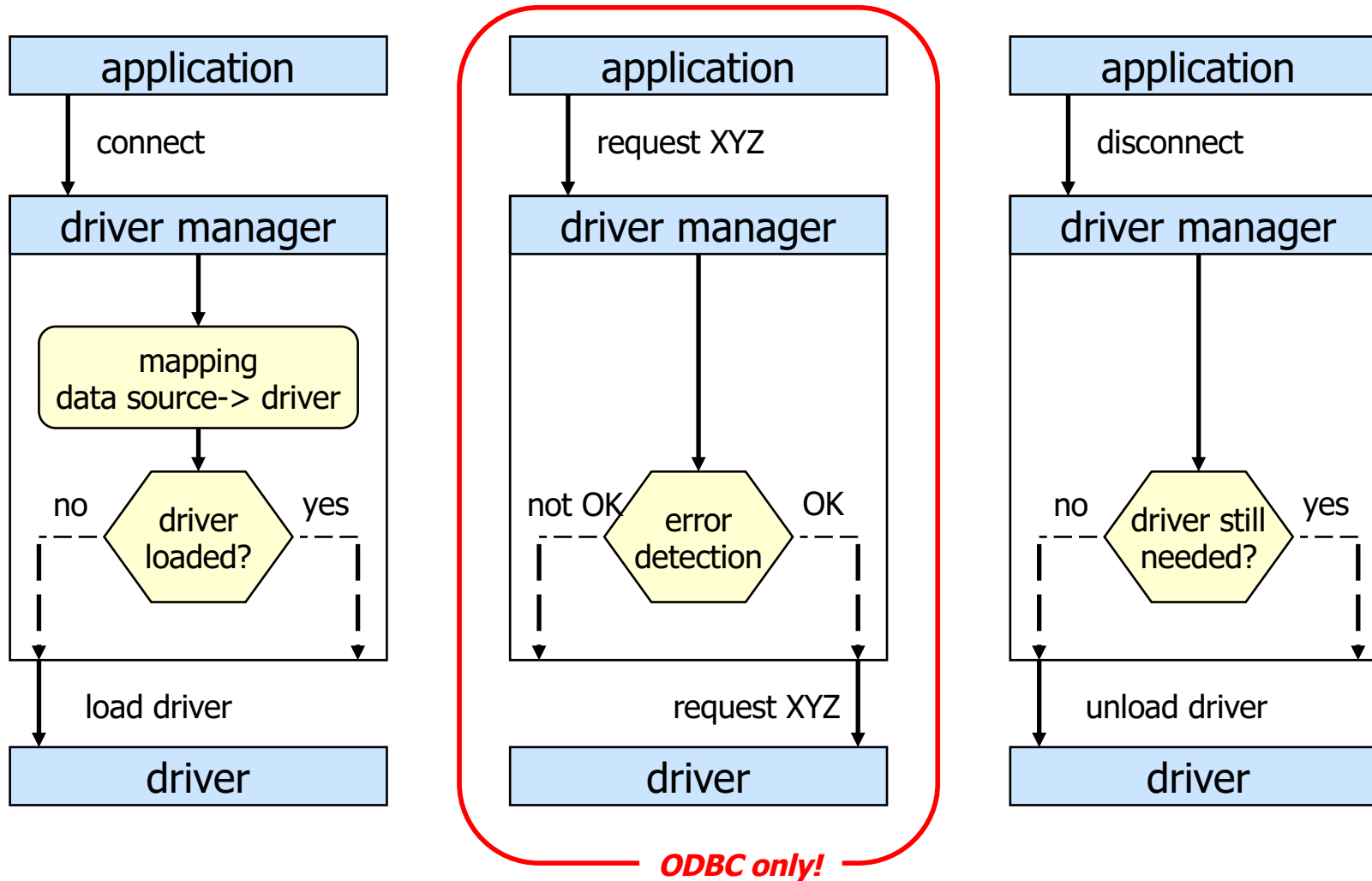
- Connection interface – transaction-oriented methods for local TAs
 - *begin is implicit*
 - commit()
 - rollback()
 - get/setTransactionIsolation()
 - NONE, READ_UNCOMMITTED, READ_COMMITTED, REPEATABLE_READ, SERIALIZABLE
 - get/setAutoCommit()
- Here, the scope of the transaction is a single connection!
 - support for distributed transactions is provided using additional extensions, interactions with a transaction manager (*see subsequent chapters*)
 - same in ODBC

DB Gateway Architecture

- Applications
 - programs using DB-CLI functionality
 - usage
 - connect to data sources
 - execute SQL statements (e.g., queries) over data sources
 - receive (and process) results
- Driver
 - processes CLI calls
 - hides heterogeneity of data sources
 - may translate SQL to vendor-specific dialect
 - communicates SQL requests to DBMS
 - Alternative: does the entire processing of the SQL requests
- Driver Manager
 - manages interactions between applications and drivers
 - realizes (n:m)-relationship between applications and drivers
 - tasks
 - load/unload driver
 - mapping data sources to drivers
 - communication/logging of function/method calls
 - simple error handling



Driver Manager Tasks



Driver – Tasks and Responsibilities

- Connection Management
- Error handling
 - standard error functions/codes/messages, ...
- Translation of SQL requests
 - if syntax of DBMS deviates from standard SQL
- Data type mapping
- Meta data functions
 - access (proprietary) system catalogs
- Information functions
 - provide information about driver (self), data sources, supported data types and DBMS capabilities
- Option functions
 - Parameter for connections and statements (e.g., statement execution timeout)

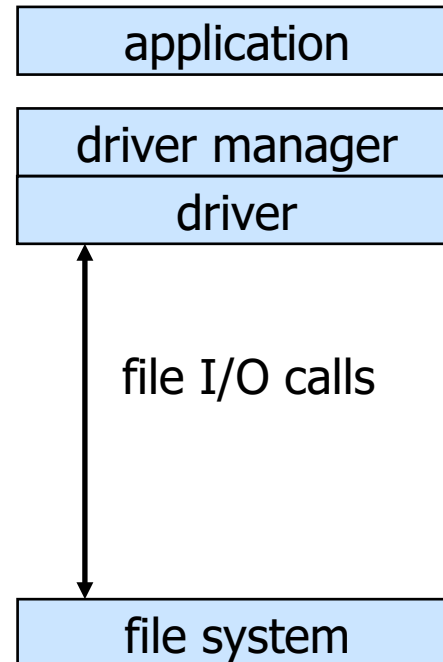
Realization Alternatives

- ODBC driver types
 - one-tier
 - two-tier
 - three-tier
- JDBC driver types
 - Type 1: JDBC-ODBC bridge
 - Type 2: Part Java, Part Native
 - Type 3: Intermediate DB Access Server
 - Type 4: Pure Java
- Application does not "see" realization alternatives!

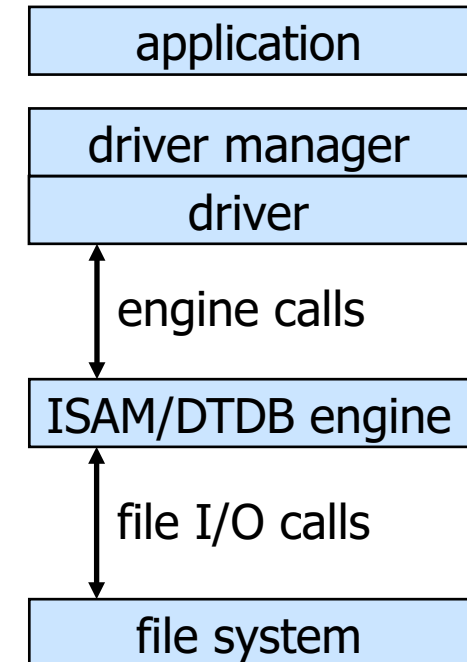
Single-Tier Driver

- Used to access flat files, ISAM files, desktop databases
- Data resides on the same machine as the driver
- Functionality:
 - **complete SQL processing** (parse, optimize, execute)
 - often lacks multi-user and transaction support

accessing flat files

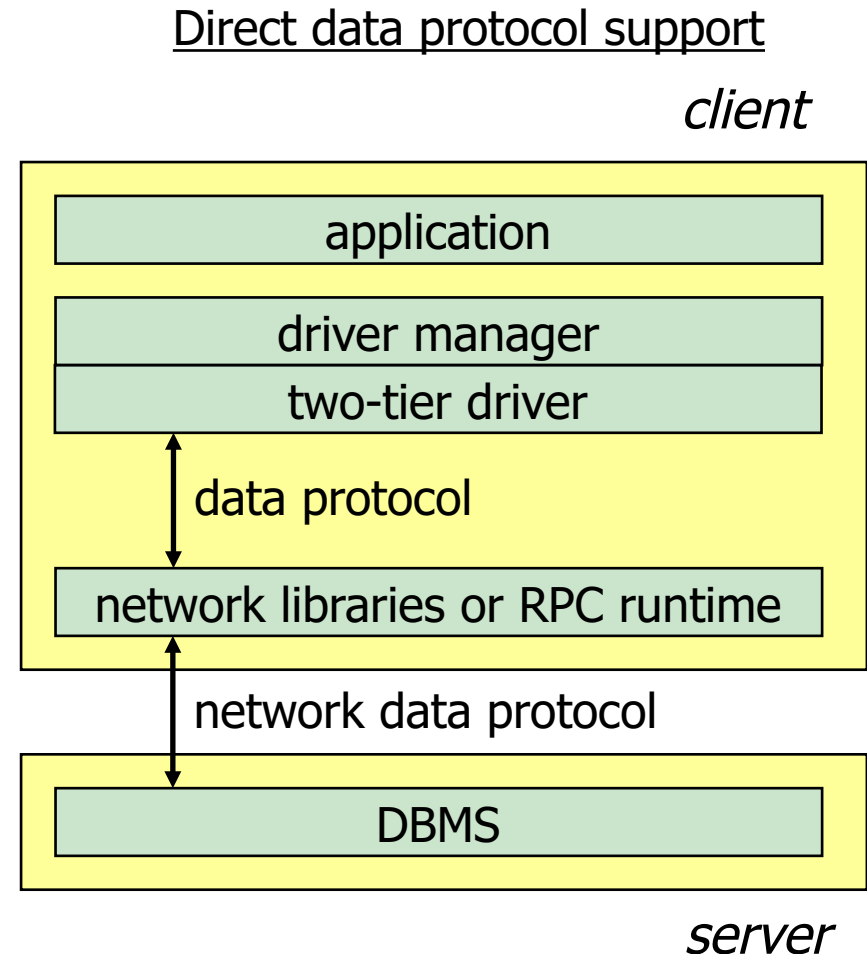


accessing ISAM files or desktop DBs



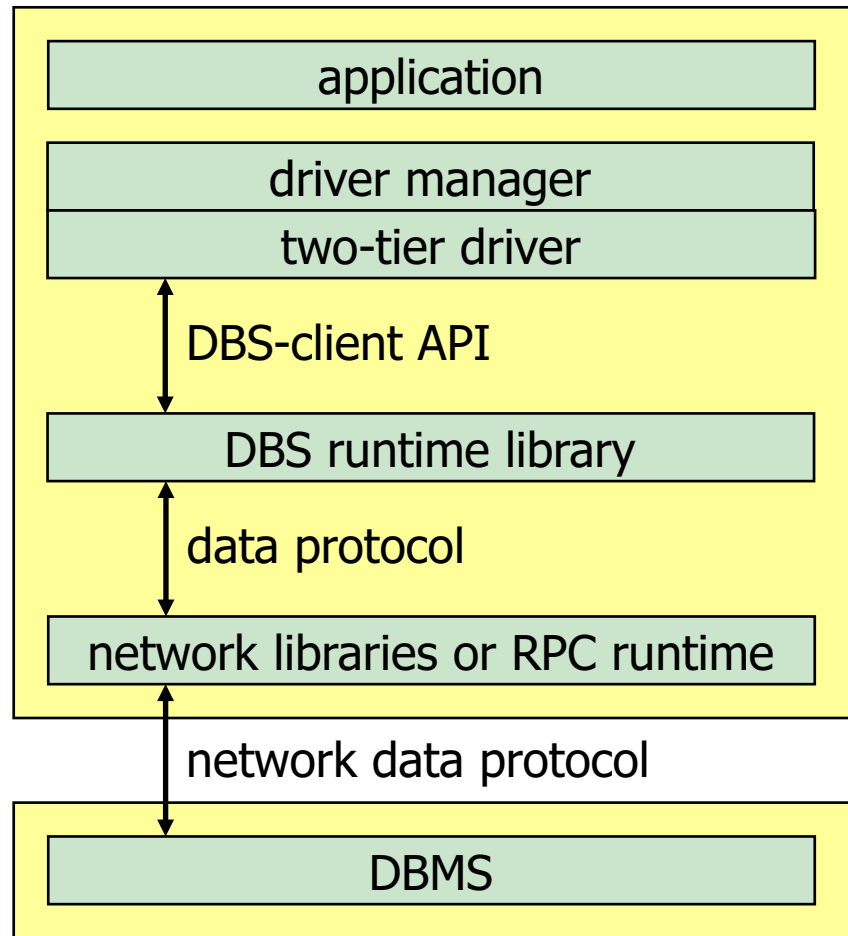
Two-Tier Driver

- Classical client/server support
 - driver acts as a client interacting with DBMS (server) through data protocol
 - Implementation alternatives
 1. direct data protocol support
 2. mapping ODBC to DBMS-client API
 3. middleware solution
- (1) Direct data protocol support
- message-based or RPC-based
 - utilizes DBMS-specific network libraries or RPC runtime



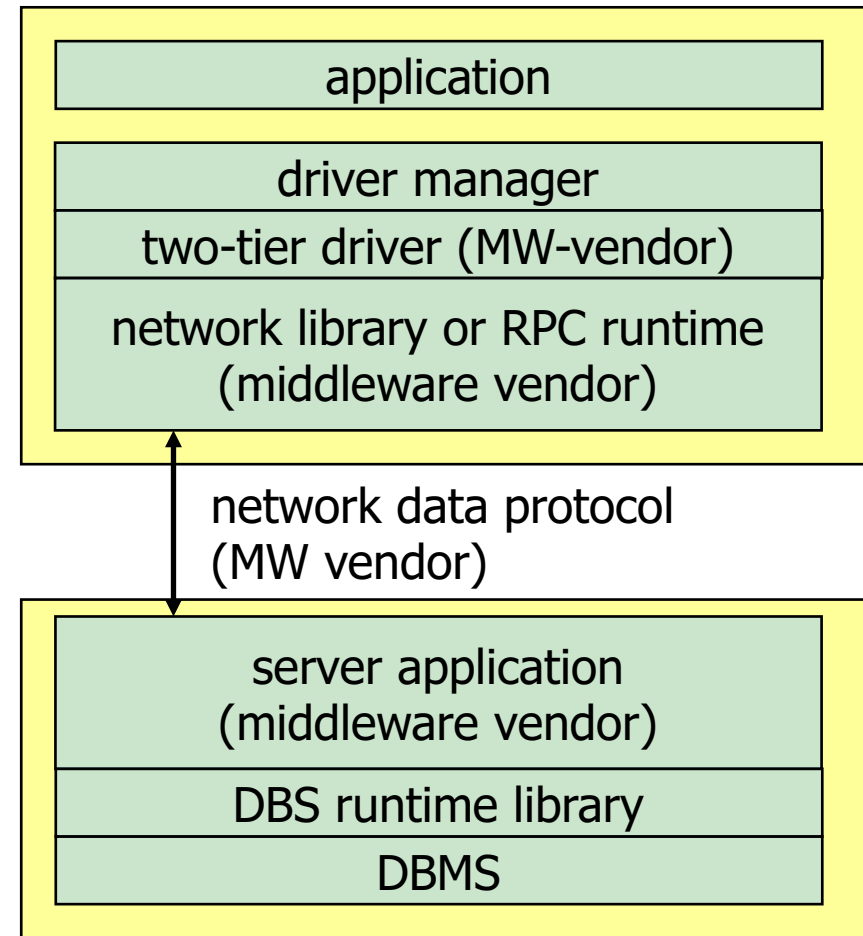
Two-Tier Driver (continued)

(2) Mapping to DBMS-client API *client*



server

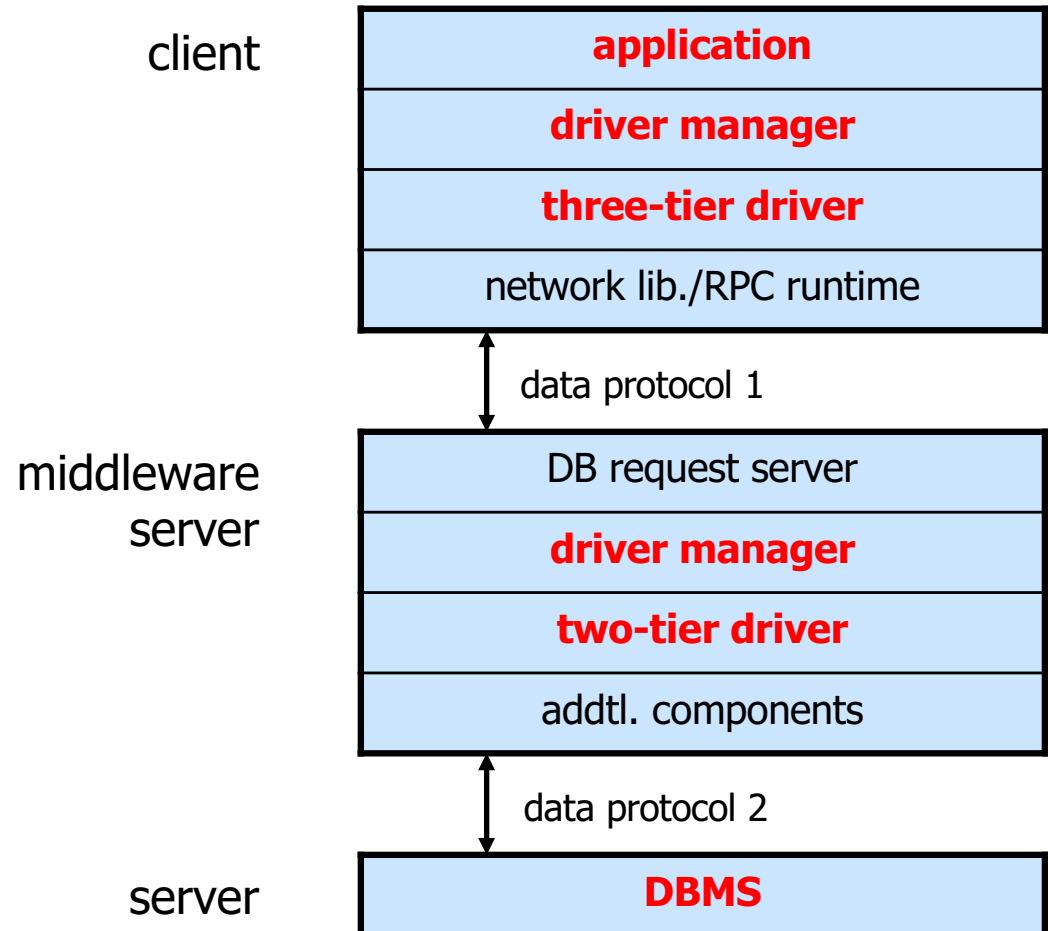
(3) Middleware solution *client*



server

Three-Tier Driver

- Middleware Server
 - “gateway server”
 - connects and relays requests to one or more DBMS servers
- Moves the complexity from the client to the middleware server
 - client requires only a single driver (for the middleware server)
- Arbitrary number of tiers possible



JDBC Driver Types

Partial Java

(requires native binaries on client)

- Type 1: JDBC-ODBC bridge
 - 2-tier
 - mapping to ODBC API
 - uses Java Native Interface (JNI)
- Type 2: Native-API Partial-Java driver
 - 2-tier
 - uses a native DBMS client library

All-Java

(does not require native binaries on client)

- Type 3: Net-Protocol All-Java driver
 - 3-tier
 - driver on client is pure Java
 - communicates with JDBC server/gateway
- Type 4: Native-Protocol All-Java driver
 - 2-tier
 - pure Java
 - implements the network data protocol of the DBMS
 - directly connects to the data source

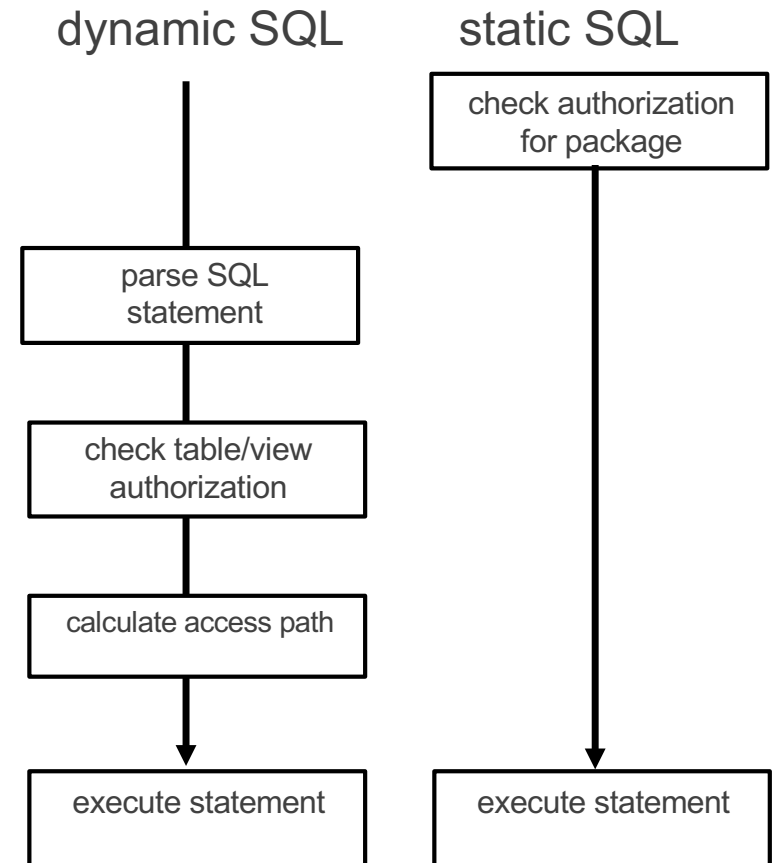


SQL Object Language Bindings (OLB)

- aka SQLJ Part 0
- Static, embedded SQL in Java
 - Development advantages over JDBC
 - more concise, easier to code
 - static type checking, error checking at precompilation time
 - Permits static authorization
 - Can be used in client code and stored procedures
- Goal: SQLJ translator/customizer framework supports binary compatibility (unlike traditional embedded SQL)
 - SQLJ translator implemented using JDBC
 - produces statement profiles
 - vendor-specific customizers
 - can add different implementation, to be used instead of default produced by translator
 - potential performance benefits
 - resulting binary contains default and possibly multiple customized implementations
- Interoperability with JDBC
 - combined use of SQLJ with JDBC for flexibility

SQL/OLB

- Static SQL authorization option
 - Static SQL is associated with "program"
 - Plans/packages identify "programs" to DB
 - Program author's table privileges are used
 - Users are granted EXECUTE on program
 - Dynamic SQL is associated with "user"
 - No notion of "program"
 - End users must have table privileges
 - BIG PROBLEM FOR A LARGE ENTERPRISE !!!
- Static SQL syntax for Java
 - INSERT, UPDATE, DELETE, CREATE, GRANT, etc.
 - Singleton SELECT and cursor-based SELECT
 - Calls to stored procedures (including result sets)
 - COMMIT, ROLLBACK
 - Methods for CONNECT, DISCONNECT



SQL/OLB vs. JDBC: Retrieve Single Row

- SQL OLB

```
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP  
WHERE NAME=:name };
```

- JDBC

```
java.sql.PreparedStatement ps = con.prepareStatement(  
    "SELECT ADDRESS FROM EMP WHERE NAME=?");  
ps.setString(1, name);  
java.sql.ResultSet names = ps.executeQuery();  
names.next();  
name = names.getString(1);  
names.close();
```

Result Set Iterators

- Mechanism for accessing the rows returned by a query
 - Comparable to an SQL cursor
- Iterator declaration clause results in generated iterator class
 - Iterator is a Java object
 - Iterators are strongly typed
 - Generic methods for advancing to next row
- Assignment clause assigns query result to iterator
- Two types of iterators
 - Named iterator
 - Positioned iterator

Named Iterators - Example

- Generated iterator class has accessor methods for each result column

```
#sql iterator Honors ( String name, float grade );  
Honors honor;  
  
...  
#sql [recs] honor =  
    { SELECT SCORE AS "grade", STUDENT AS "name"  
      FROM GRADE_REPORTS  
      WHERE SCORE >= :limit AND ATTENDED >= :days  
      ORDER BY SCORE DESCENDING };  
while (honor.next())  
{  
    System.out.println( honor.name() + " has grade "  
        + honor.grade() );  
}
```

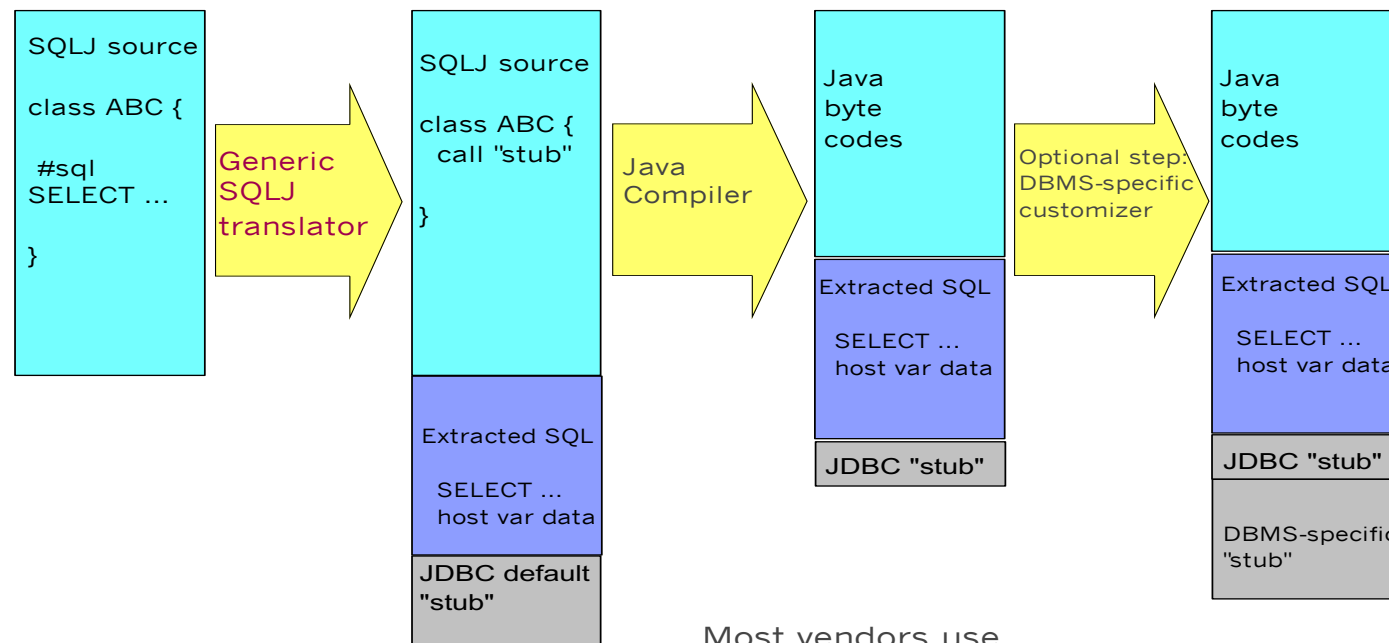
Positioned Iterator

- Use FETCH statement to retrieve result columns into host variables based on position

```
#sql iterator Honors ( String, float );
Honors honor;
String name;
float grade;
#sql [recs] honor =
    { SELECT STUDENT, SCORE FROM GRADE_REPORTS
      WHERE SCORE >= :limit AND ATTENDED >= :days
      ORDER BY SCORE DESCENDING };
while (true) {
    #sql {FETCH :honor INTO :name, :grade };
    if (honor.endFetch()) break;
    System.out.println( name + " has grade " + grade );
}
```

SQLJ - Binary Portability

- Java as a platform-independent language
- Use of generic SQLJ-precompiler/translator (avoids DBMS-specific precompiler technology)
- Generated code uses "standard" JDBC by default
- Compiled SQLJ application (Java byte code) is portable
- Customizer technology allows DBMS-specific optimizations after the compilation



Most vendors use default JDBC "stub"

Summary

- Coupling approaches
 - static and dynamic embedded SQL
 - call-level interface (CLI)
- Gateways
 - ODBC / JDBC
 - support uniform, standardized access to heterogeneous data sources
 - encapsulate/hide vendor-specific aspects
 - multiple, simultaneously active connections to different databases and DBMSs
 - driver/driver manager architecture
 - enabled for distributed transaction processing (*see subsequent chapters*)
 - high acceptance
 - important infrastructure for realizing IS distribution at DB-operation level
 - no support for data source integration
- JDBC
 - 'for Java', 'in Java'
 - important basis for data access in Java-based middleware (e.g., J2EE)
- SQLJ
 - combines advantages of embedded SQL with binary portability, vendor-independence