Applikationsprogramgränsnitt mot relationsdatabaser

Tore Risch, Uppsala Database Laboratory
Uppsala University, Sweden

Tore.Risch@it.uu.se

14 maj 2008

1 Inledning

Det finns i huvudsak tre sätt kommunicera med relationsdatabaser genom SQL:

1. I *textuella frågegränssnitt* matar användaren in SQL satser som textsträngar som sedan sänds till databasservern för exekvering. SQL-satserna kan också läsas från fil. Slutanvändaren kommer vanligtvis inte i kontakt med det textuella gränssnittet som i huvudsak används av databasprogrammerare. Textuella frågegränssnitt används dock av kunniga användare som har behov att ställa avancerade databasfrågor.

2. I *grafiska frågegränssnitt* byggs SQL-satserna upp av systemet baserat på grafiska användargränssnitt. Det mest spridda är MS Access som har ett grafiskt gränssnitt där frågor specificeras m.h.a. formulär, kallat *Query By Example (QBE)*. När en fråga exekveras genereras först SQL-satser som sänds till databashanteraren för exekvering. Grafiska gränssnitt används av databasprogrammerare och användare. Kräver i allmänhet mindre kunskaper än textuellt SQL-gränssitt men kan inte hantera alla möjliga SQL-frågor, och det kan vara mer komplicerat att specificera avancerade frågor.

3. Det vanligaste sättet att använda relationsdatabaser är dock att man kör ett applikationsprogram som internt anropar SQL. För detta behövs *SQL applikationsprogramgränssnitt* mellan olika programmeringsspråk och SQL-databaser.

I huvudsak finns två typer av SQL applikationsprogramgränssnitt, *inbäddad SQL* och *SQL klientgränssnitt*.

Applikationprogramgränssnitten måste uppfylla följande krav:

1. Man måste kunna hantera uppkopplingar mellan applikationsprogrammet och databasen, s.k. *sessioner*.

2. Man måste kunna sända SQL-satser till databashanteraren för exekvering. Ett speciellt problem är att SQL-satserna kan potentiellt returnera mycket stora resultatmängder, eventuellt större än tillgängligt primärminne.

3. Det kan ta lång tid att optimera en SQL-fråga. Gränssnittet måste ha metoder att undvika onödig SQL-optimering.

2 SQL klientgränssnitt

Varje databashanteringssystem (DBHS, eng. DBMS) tillhandahåller ett *tillverkarspecifikt klientgränssnitt* i form av ett subrutin- eller klassbibliotek. Man kan där koppla upp sessioner, sända SQL-satser till databasen och hantera konverteringar av data mellan SQL och applikationsvariabler. Dessa gränssnitt är dock DBHS-specifika och man blir lätt uppbunden mot speciell tillverkare om man gör sig beroende av dem.

För att göra applikationsgränssnitten DBHS-oberoende har man utvecklat de standardiserade klientgränssnitten ODBC och JDBC. Dessa gränssnitt anropar de tillverkarspecifika klientgränssnitten internt.

2.1 ODBC

ODBC, Open DataBase Connectivity är ett DBHS-oberoende SQL klientgränssnitt. ODBC är utvecklat av MicroSoft men är oberoende av operativsystem och har blivit en de facto industristandard för SQL klientgränssnitt. Nära besläktat är *SQL/CLI, SQL Call Level Interface*, som är en ISO standard för SQL klientgränssnitt. SQL/CLI är till största delen identiskt med ODBC. Med ODBC (SQL/CLI) kan man utveckla applikationsprogram som är oberoende av relationsdatabastillverkare, och man kan t.o.m. göra applikationer som anropar mer än ett underliggande DBHS. ODBCs arkitektur illustreras av figur 1.

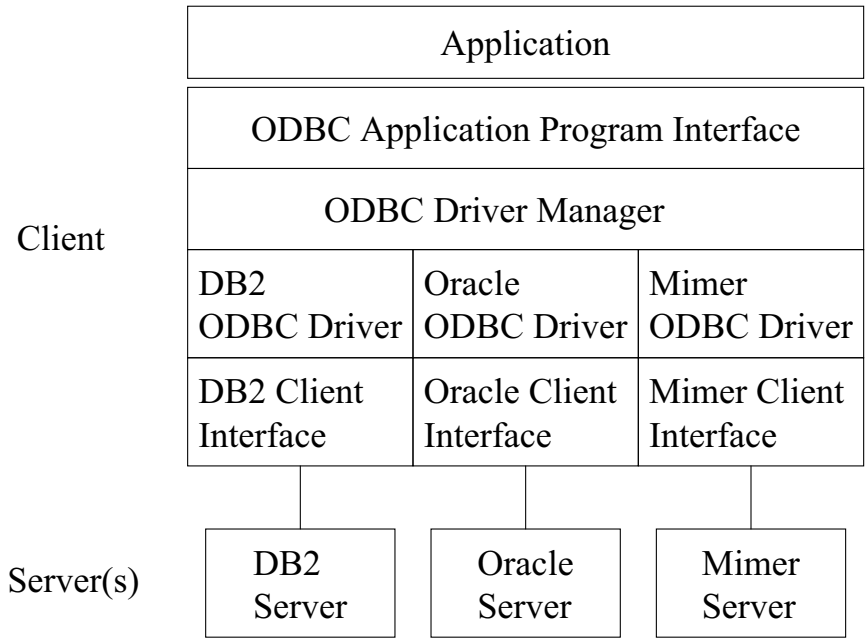
Applikationsprogrammet (vanligtvis i C/C++) länkas ihop med ett bibliotek av ODBC gränssittsfunktioner. Gränssnittsfunktionerna anropar en generell ODBC drivrutinshanterare (driver manager) till vilken inlänkats en eller flera DBHS-specifika ODBC drivrutiner. Varje DBHS-specifik ODBC drivrutin innehåller kod för att översätta mellan ODBCs standardgränssnitt och det klientgränssnitt som resp. tillverkares relationsdatabas använder[1]. Det DBHS-specifika gränssnittet anropas av OBDC drivrutinen för att kommunicera med relationsdatabasen. Normalt körs applikationsprogrammet i en process och databasservarna på separata maskiner[2].

2.2 JDBC

JDBC, Java DataBase Connectivity är ett ODBC-liknande gränssnitt för programmeringsspråket Java. Ursprungligen var funktionaliterna i JDBC och ODBC

[1] I figuren exemplifieras detta för DBHS DB2, Oracle och Mimer.

[2] I MS Access körs allt dock i samma process på en maskin, dvs. MS Access databashanterare är länkad direkt till MS Access ODBC drivrutin.

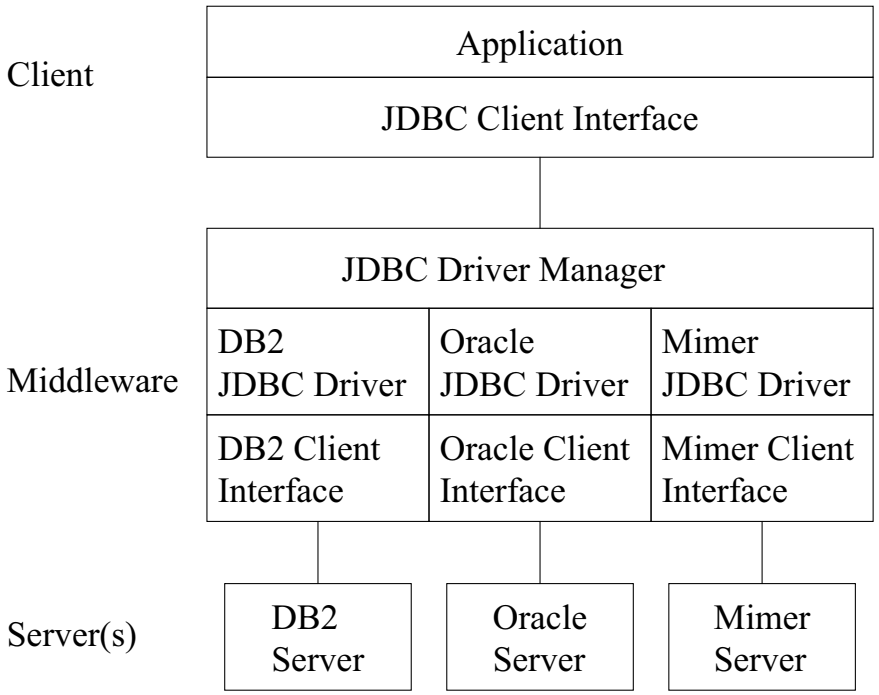


Figur 1: ODBCs arkitektur

identiska, men viss divergering har skett med tiden. En direkt (native) JDBC implementering har arkitektur som illusterat i figur 2. En skillnad mot ODBC är att drivrutinhanteraren för JDBC körs i en separat server, ofta del av en s.k. *applikationsserver*. Till Javaapplikationen länkas ett klassbibliotek, *JDBC klientbiblioteket*, för gränssnitt mot JDBC. JDBC klientbiblioteket kommunicerar över datornätverket med JDBC drivrutinhanteraren som körs i separat process/maskin. Drivrutinhanteraren är uppbyggd analogt som JDBC med specifika drivrutinhanterare som översätter till och anropar DBHS-specifika klientgränssnitt.

En viktig anledning till denna trenivåarkitektur (3-tier architecture) med separat mellanprogramvara (middleware) för JDBC drivrutinhanteringen är att det möjliggör att klienten blir en ren (pure) Javaapplikation som helt är kodad i Java, även om de DBHS-specifika gränssnitten är kodade i C/C++.

Eftersom JDBC och ODBC är så lika är det vanligt att JDBC tillhandahålles genom en *JDBC-ODBC* brygga, illustrerat i figur 3. Där använder man en ODBC drivrutinhanterare som en speciell JDBC drivrutin. Detta gör att man sparar implementeringsarbete att utveckla nya JDBC drivrutiner för olika databashanterare utan kan utnyttja redan implementerade ODBC drivrutiner. Nackdelen är att det blir en extra mjukvarunivå, vilket inte alltid har så stor betydelse med tanke på att kommunikationstiden oftast dominerar.



Figur 2: JDBCs arkitektur

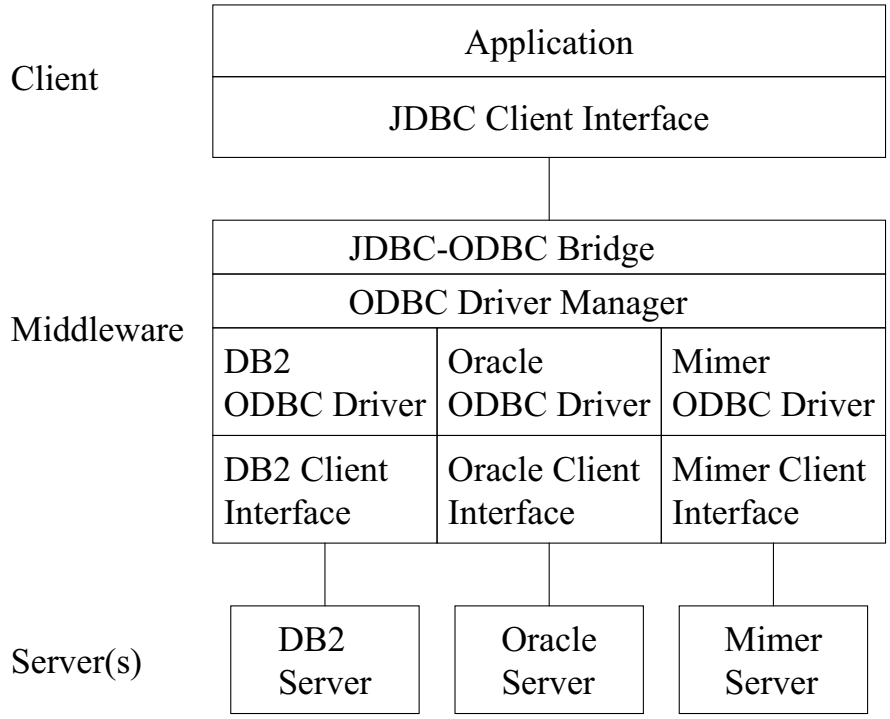
2.3 Programmering av SQL klientgränssnitt

Som exempel på hur SQL klientgränssitten är utformade använder vi JDBC. Både ODBC och DBHS-specifika SQL klientgränssnitt har samma struktur, men andra funktionsnamn etc.

Antag att vi har en tabell

```
EMPLOYEE(SSN,NAME,INCOME)
```

och vi vill skriva ett program som skriver ut namet på alla som tjänar mer än 100000. Följande JDBC program gör detta:



Figur 3: JDBC genom JDBC-ODBC brygga

```
{
                // Load SUN's JDBC-ODBC bridge:
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                // Connect to database server S:
   Connection con = DriverManager.getConnection("jdbc:odbc:S",
                    "myLogin", "myPassword");
               // Create JDBC statement holder:
   Statement stmt = con.createStatement();
   String query = "SELECT NAME FROM EMPLOYEE WHERE INCOME > 10000";
   ResultSet rs = stmt.executeQuery(query);
                // Iterate over query result:
   while (rs.next()) // while tuples in result set do
   {
      String s = rs.getString("NAME");
      System.out.println(s);
   }
}
```

Metoden `executeQuery` tar en SQL-fråga som argument och sänder den till

databasservern för exekvering. Eftersom resultatet från en fråga kan vara mycket större än tillgängligt primärminne behövs speciell metod för att hantera svaret från frågan. Gränssnitt mot relationsdatabaser som JDBC och ODBC är därför *strömmade*. T.ex. får man i JDBC resultatet som en ström av rader representerade av klassen `ResultSet` som man sedan itererar över som i exemplet. Man kan tycka att det vore enklare att returnera en datatyp representerade hela resultattabellen som en datastruktur i Java, men en sådan lösning fungerar inte för stora resultatmändger som inte får plats i minnet. T.ex. är det lätt att av misstag glömma joinvillkor vilket genererar mycket stor resultattabell som ej får plats i Java-minnet.

Ett annat problem med SQL är att optimering av SQL-satser kan vara mycket dyrbart[3]. Man har därför utvecklat mekanismer för att undvika att optimera frågan varje gång den anropas. I vårt exempel kanske man vill göra ett program som i en slinga (loop) läser in undre inkomstgränser från användaren och sedan skriver ut alla som tjänar bättre. Följande kod gör detta:

```
{
                // Load SUN's JDBC-ODBC bridge:
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
                // Connect to database server S:
   Connection con = DriverManager.getConnection("jdbc:odbc:S",
                    "myLogin", "myPassword");
   Statement stmt = con.createStatement();
               // Parameterized query:
   String query = "SELECT NAME FROM EMPLOYEE WHERE INCOME > ?";
   Int incomeLimit;
               // Prepare parameterized query:
   PreparedStatement getPersons = con.prepareStatement(query);

   while(....){
     //  Code to read lower income limit into incomeLimit
     ....
               // Set 1st parameter to incomeLimit:
     getPersons.setInt(1,incomeLimit);
     ResultSet rs = stmt.executeQuery(getPersons);
                // Iterate over query result:
     while (rs.next()) {
        String s = rs.getString("NAME");
        System.out.println(s);
     }
   }
}
```

Metoden `prepareStatement` kompilerar och optimerar, *preparerar*, en parametriserad SQL-fråga. Parametrarna markeras med `?` och numreras internt 1,2,3, etc.

[3]Kostnaden är $O(2^Q)$, där Q är hur många operatorer som finns i frågan.

vartfter de dyker upp i frågan. Bara värden kan parametriseras och `?` kan således inte placeras hur som helst. Med metoden `setInt` sätter man en given parameter i den preparerade frågan till ett heltal[4]. En preparerad fråga anropas på samma sätt som andra frågor. M.h.a. `prepareStatement` sparar man optimeringstid om man vill anropa samma fråga flera gånger.

3 Inbäddad SQL

Om en fråga är komplicerad kan det ta en hel del tid att preparera den. I ODBC/JDBC innebär detta att det tar tid att starta program som först gör en massa `prepareStatement`.

För att spara ytterligare prepareringstid och även för att förenkla gränsnittet mot databasen har man utvecklat *inbäddad SQL*. I inbäddad SQL har man utvidgat ett konventionellt programmeringsspråk (C, C++, Java, etc) med anrop till SQL-satser. SQL-satserna markeras på ett speciellt sätt och en speciell kompilator, en *förprocessor*, översätter uppmärkta SQL-satser till anrop till det DBHS-specifika klientgränssnittet.

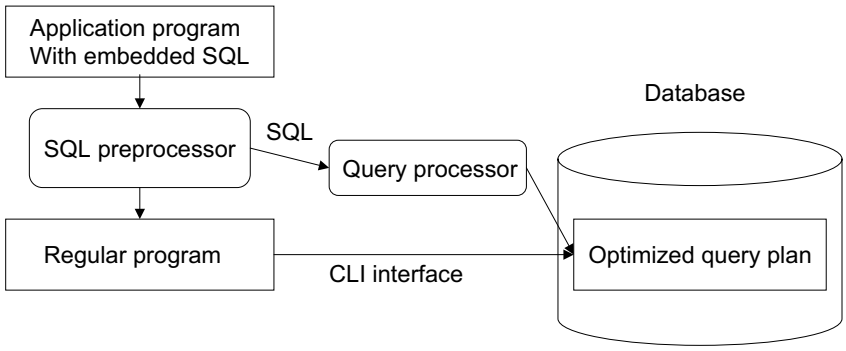
Exempel på inbäddad SQL för Java (SQLJ):

```
// Example of SQLJ code:

import java.sql.*;
import java.io.*;
import sqlj.runtime.*;
import sqlj.runtime.ref.*;

    public static void main(String argv[])
    {   try
        {
            String url = "jdbc:db2:c3421a";     // URL is jdbc:db2:dbname
            Connection con = null;

            // Set the connection with default id/password
            con = DriverManager.getConnection(url);

            // Set the default context for embedded SQL to con
            DefaultContext ctx = new DefaultContext(con);
            DefaultContext.setDefaultContext(ctx);

            int sid = 0;
            String sname = null;
            short age = 0;

            // Add code to read sid here
```

[4]Motsvarande metoder finns för flyttal, strängar, etc.

```
        // Find the name and age of sailor SID:
        #sql { SELECT SNAME, AGE INTO :sname, :age
               FROM TECHSTU.SAILOR
               WHERE SID = :sid } ;

        System.out.println ("Sailor " + sname + "'s age is " + age);
        System.out.println ("Executed Successfuly");
        System.out.println ("Bye");
    }
}
```

Arkitekturen för processering av inbäddad SQL illustreras i figur 4. Förprocessorn kompilerar SQL-satserna inbäddade i applikationen, och de kompilerade SQL-satserna sparas i databasserven. Därefter kompileras det förprocessade programmet som vanligt. När man vid senare tillfälle exekverar programmet sker ingen SQL-kompilering alls, utan det sker direkt parametriserade anrop av förkompilerade SQL-satser i databasservern. Om en exekveringsplan blivit inaktuell p.g.a. databasuppdateringar sker omkompilering dynamiskt när exekveringsplanen anropas nästa gång. Att amortera av optimeringskostnaden är den viktigaste anledningen att använda inbäddad SQL. En mindre viktig anledning är att inbäddad SQL kan vara något enklare att anropa från högnivåspråk än med SQL klientgränssnitt.



Figur 4: Inbäddad SQL

4 Databasprocedurer

Databasprocedurer (eng. *stored procedures*) är program skrivna i en procedurell utvidgning av SQL som körs inuti databasservern. På så vis kan man göra både beräkningar och bearbetningar av data på databasservern. Det finns följande anledningar att använda databasprocedurer:

- Mindre kommunikationstid. Ofta varvas flera datahämtnings- och beräkningsfaser i en applikation så att man hämtar först data, gör sedan vissa beräkningar över hämtade data, gör sedan ytterligare frågor baserade på resultatet av beräkningarna, etc. Om alla beräkningar görs i klienten kan det behövas en hel del kommunikation mellan databasservern och klienten. Detta kan innebära betydande prestandaförluster, speciellt om tiden för tur-och-retur meddelande till databasservern är lång eller om många interaktioner sker mellan klienten och servern.

- Bättre systemstruktur. Generell kod vars syfte är bearbetning och underhåll av data hör naturligt hemma tillsammans med databasen. T.ex. kod för att skapa nya data (t.ex. skapa konto, lägga in order) kan innebära att indata måste kontrolleras och ett antal tabeller uppdateras. Genom skapa data med databasprocedurer får man konsistent uppdatering av databasen.

- Förkompilerade frågor. Som tidigare nämnts är frågeoptimering dyrbart, vilket ju är anledningen till att man infört `prepare` och inbäddad SQL. Ett alternativt till detta är att förkompilera frågorna genom att lägga in dem i anrop från databasprocedurer. Frågeoptimeringen sker då när proceduren defineras för databashanteraren.

Olika databashenteringssystem har tidigare haft olika SQL-utvidgningar för databasprocedurer. Numera är dock databasprocedurer standardiserade genom SQL-PSM som är del av SQL-99. I SQL-PSM defineras databasprocedurer med ett kommando

```
CREATE PROCEDURE namn(deklarationer)
BEGIN
      SQL-PSM satser
END
```

SQL-PSM satserna kan vara vanliga SQL-satser (t.ex. uppdateringar) eller speciella SQL-PSM satser för t.ex. felhantering. För att på ett skalbart sätt kunna anropa SELECT-satser som returnerar stora datamängder har man ett strömorienterat gränssnitt liknande JDBC/ODBC och inbäddad SQL. En speciell markör kallad *cursor* indikerar position i resultattabell från en fråga, liknande *ResultSet* i JDBC. T.ex. följande procedur beräknar medelvärde och varians av lönerna i en avdelning:

```
CREATE PROCEDURE lönestatistik(
       IN  avd CHAR[20],
```

```
       OUT medel REAL,
       OUT varians REAL)
DECLARE räknare INTEGER;
BEGIN
   SET medel = 0.0;
   SET varians = 0.0;
   SET räknare = 0;
   FOR markör AS SELECT lön FROM anställda WHERE avdelning = avd;
   DO
      SET räknare = räknare + 1;
      SET medel = medel + lön;
      SET varians = varians + lön * lön;
   END FOR;
   SET medel = medel / räknare;
   SET varians = varians/räknare - medel * medel;
END;
```

När ovanstående procedur skickas till databasservern komplieras den, inbäddade frågor optimeras, och optimal kompilerad kod läggs i servern. Databasprocedurer anropas med satsen

```
CALL procedur(parametrar);
```

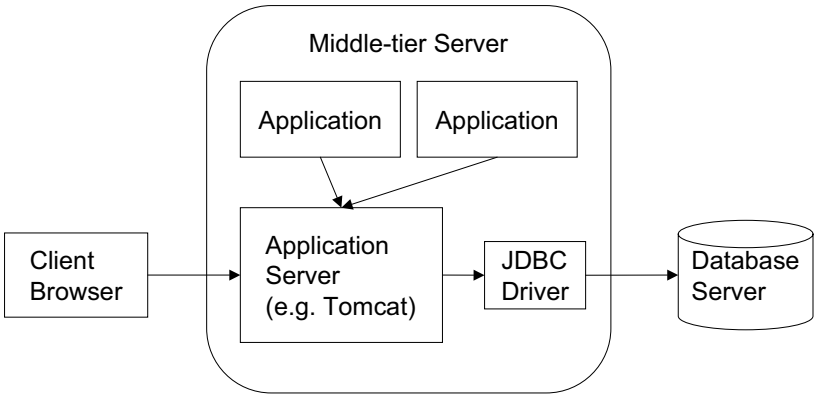
Databasprocedurer anropas från JDBC genom en speciell klass, *CallableStatement*. Liknande konstruktioner finns i ODBC, inbäddad SQL, etc.

5 Applikationsservrar

För att begränsa systemkraven hos klientdatorn vill man ofta göra det möjligt att interagera med applikationen från en vanlig web-browser utan att behöva ladda ner program. Sådana applikationer måste köras på en server, en *mellanserver*, illustrerat av figur 5. Web-browsern interagerar då med ett serverprogram, *applikationsservern*, i mellanservern m.h.a. HTML. På serversidan finns speciella web-sidor som innehåller HTML-kod med inbäddad applikationsprogramkod som anropas från applikationsservern. Applikationskod kommer åt relationsdatabaser genom t.ex. Java och JDBC.

Om programmeringsspråket för applikationerna är Java kallar man websidorna på servern *Java Server Pages, JSP* och programmen som anropas från applikationsserven kallas *servlets*. En känd applikationsserver för Java är *Apache TomCat*. Liknande system finns för bl.a. Microsoft (ASP).

Fördelen med applikationsservrar är att klientkraven hålls begränsade (s.k. *thin client*). Nackdelen är att all interaktion sker genom HTML, att alla beräkningar måste göras i applikationsservern, och att flera lager av servrar kan vara ineffektivt. Speciellt om man har avancerad grafik eller mycket användarinteraktion kan det bli långsamt. Det kan då vara bättre att i stället ladda ner en Java *applet* i klienten som sköter bearbetningar och grafik och som direkt anropar



Figur 5: Applikationsserver

databasen genom JDBC, som beskrivet tidigare. Ett annat alternativ är att lägga bearbetningar i databasprocedurer som anropas från klientprogrammmen. För enkla applikationer som har ett formulärorienterat användargränssnitt är dock applikationsservrar OK.

Kombination av klient och mellanserverprogram används ofta. T.ex. Oracle tillåter att man kör Java i klient, mellanserver, eller t.o.m. i databasservern själv, vilket ger utrymme för intressanta designbeslut.

Eftersom HTML är tillståndslöst är ett speciellt problem för applikationsservrar hur man upprätthåller sessioner mot systemet. Applikationsservrar använder antingen *coockies* eller genererade URLer för att spara tillstånd för en klient mellan olika webbsidor.