

Examensarbete

Datalagringsmetodik i J2EE

av

Björn Brenander

LiTH-IDA-Ex-01/95

2001-12-28

Handledare: Tomas Byström, Fredrik Samson och Johan Söderqvist

Examinator: Olof Johansson

Sammanfattning

Lagring av data på ett beständigt och effektivt sätt är viktigt i många sammanhang. Syftet med denna rapport är att undersöka de möjligheter som finns att i J2EE-miljö lagra data i en relationsdatabas.

Rapporten ger bakgrundsinformation om relationsdatabaser, SQL och Enterprise JavaBeans. Därefter ges en genomgång av tillgängliga tekniker inom området tillsammans med en rad olika designmönster och optimeringsmetoder. Design och implementation av ett testsystem presenteras, varefter testresultat redovisas. Rapporten avslutas med en kort diskussion om resultaten samt några slutsatser och rekommendationer för vidare arbete.

Abstract

Persistent and efficient storing of data is crucial in many applications. The objective of this thesis is to examine the possibilities available for storing data in a relational database in a J2EE environment.

The thesis gives background information on relational databases, SQL and Enterprise JavaBeans. Available techniques are described along with design patterns and methods of optimization. Design and implementation of a test bench is presented after which the results of the testing are shown. The thesis is concluded by a short discussion on the results along with conclusions and recommendations for further work.

Förord

Detta examensarbete påbörjades i mitten av april 2001, och pågick under hela sommaren och hösten. Till slut blev rapporten klar, och framläggningen hölls torsdagen den 20 december. Jag vill med dessa rader tacka några personer som spelat en stor roll under denna tid:

min fästmö Cecilia Johansson, för kärlek och förståelse

min examiner Olof Johansson från Institutionen för datavetenskap samt mina handledare Tomas Byström, Fredrik Samson och Johan Söderqvist från Ida Infront, för stöd och givande diskussioner

min opponer Adam Skogman, för gott samarbete och många intressanta synpunkter under arbetets gång

mina goda vänner Joel Rosdahl och Fredrik Örvill, för tips och korrekturläsning

Linköping, 28 december 2001

Björn Brenander

Innehåll

1	INLEDNING	1
1.1	OM IDA INFRONT	1
1.2	LÄSANVISNINGAR	1
1.3	OM RAPPORTEN.....	1
1.3.1	<i>Språk</i>	2
1.3.2	<i>Teknik</i>	2
2	PROBLEMFORMULERING.....	3
2.1	BAKGRUND	3
2.1.1	<i>ipax</i>	3
2.2	SYFTE	5
2.3	PROBLEMBESKRIVNING OCH ARBETSMETOD	5
2.3.1	<i>Metod</i>	5
2.4	AVGRÄNSNINGAR	6
3	BAKGRUNDSTEORI	7
3.1	RELATIONSDATABASES	7
3.1.1	<i>En exempeldatabas</i>	7
3.2	SQL	9
3.2.1	<i>INSERT</i>	9
3.2.2	<i>SELECT</i>	10
3.2.3	<i>UPDATE</i>	10
3.2.4	<i>DELETE</i>	11
3.3	ENTERPRISE JAVABEANS	11
3.3.1	<i>Vad är EJB?</i>	11
3.3.2	<i>Vad är en EJB-server?</i>	12
3.3.3	<i>Anatomi hos EJB</i>	13
3.3.4	<i>Sessionsböror</i>	14
3.3.5	<i>Entitetsböror</i>	16
3.3.6	<i>EJB och transaktioner</i>	20
3.4	JDBC	22
3.4.1	<i>Exempel</i>	22
3.4.2	<i>Drivrutiner</i>	24
3.5	SQLJ – INBAKAD SQL.....	27
4	GENOMGÅNG AV TILLGÄNGLIGA TEKNIKER	31
4.1	EEJB.....	31
4.1.1	<i>CMP eller BMP?</i>	32
4.1.2	<i>Finkornighet</i>	32
4.2	JAVA/JDBC	33
4.3	SQLJ.....	34
4.4	JDO	34
4.5	TEKNIKER, DESIGNMÖNSTER OCH OPTIMERING.....	34
4.5.1	<i>Automatisk BMP</i>	34
4.5.2	<i>Förhindrande av onödiga databasoperationer</i>	35
4.5.3	<i>Värdeobjekt</i>	35
4.5.4	<i>Fasadböror</i>	35
4.5.5	<i>Serialiserade attribut i databasen</i>	36
4.5.6	<i>Lagring av relationer</i>	36
4.5.7	<i>Uppkopplingsstrategi</i>	36
4.5.8	<i>DAO</i>	38
4.6	SAMMANFATTNING	39
5	DESIGN OCH IMPLEMENTERING	41
5.1	TESTMODELL	41
5.1.1	<i>Entiteten Författare</i>	41

5.1.2	Entiteten Bok.....	41
5.2	TESTFALL.....	41
5.2.1	Läsning.....	41
5.2.2	Uppdatering.....	41
5.2.3	Listning.....	42
5.2.4	Sökning.....	42
5.2.5	Skapande.....	42
5.2.6	Borttagning.....	42
5.3	URVAL.....	42
5.3.1	Entitetsbönor med CMP.....	42
5.3.2	Entitetsbönor med BMP.....	43
5.3.3	Sessionsböna.....	43
5.3.4	Java/JDBC.....	43
5.4	DESIGN.....	43
5.4.1	Databasdesign.....	45
5.4.2	Installationsbeskrivningar.....	46
5.5	DATABASGENERATOR.....	52
5.6	TESTKLIENT.....	53
5.7	TESTSPECIFIKATION.....	53
5.8	TEKNIK.....	53
5.8.1	Hårdvara.....	53
5.8.2	Mjukvara.....	53
6	RESULTAT.....	55
6.1	TESTFALL OCH TESTADE IMPLEMENTATIONER.....	55
6.2	TESTRESULTAT.....	55
6.2.1	Testfall inbegripande enstaka data.....	57
6.2.2	Testfall inbegripande listningar.....	59
6.2.3	Testfall inbegripande sökningar.....	61
6.3	TEKNIKGENOMGÅNG.....	64
6.3.1	BMPc.....	64
6.3.2	BMPf.....	65
6.3.3	CMP.....	65
6.3.4	SLSB.....	65
6.4	SAMMANSTÄLLNINGAR.....	65
6.4.1	Enstaka poster.....	66
6.4.2	Listningar och sökningar.....	67
6.5	TABELLER.....	68
7	DISKUSSION.....	71
7.1	OBSERVATIONER OCH KOMMENTARER.....	71
7.2	VAD SÄGER ANDRA?.....	71
8	SLUTSATSER.....	73
8.1	VIDARE ARBETE.....	73
8.1.1	Flera samtidiga klienter.....	74
8.1.2	Verklighetsnära testsekvenser.....	74
8.1.3	Minnesanvändning.....	74
8.1.4	EJB 2.0.....	74
8.1.5	JDO.....	74
9	REFERENSER.....	75
9.1	LITTERATUR.....	76
9.2	ÖVRIGA INFORMATIONSKÄLLOR.....	77
APPENDIX A	BEGREPP OCH FÖRKORTNINGAR.....	79
	BEGREPP.....	79
	FÖRKORTNINGAR.....	81
APPENDIX B	KODEXEMPEL.....	85

TESTFALL 11: LÄSA BOK INKLUSIVE EXTERNA ATTRIBUT	85
<i>BMPcServer</i>	85
<i>BookBMPc</i>	85
<i>BMPfServer</i>	88
<i>BookBMPf</i>	89
<i>AttributeBMPf</i>	90
<i>CMPServer</i>	92
<i>BookCMP</i>	93
<i>AttributeCMP</i>	93
<i>SLSBServer</i>	93
TESTFALL 30: LISTA ALLA FÖRFATTARE	95
<i>BMPcServer</i>	95
<i>AuthorBMPc</i>	95
<i>BMPf</i> 97	
<i>CMPServer</i>	98
<i>AuthorCMP</i>	98
<i>SLSBServer</i>	98
APPENDIX C DATABASGENERATOR	101
DATABASGENERERING.....	101
CREATEDB.SQL	101
KÄLLKOD	102
<i>DatabaseCreator</i>	102
<i>DatabaseHelper</i>	106
APPENDIX D TESTKLIENT	109
KÄLLKOD	109
APPENDIX E VÄRDEOBJEKT.....	119
GENVO	119
NUMVO	119
AUTHORVO	119
BOOKVO.....	120
ATTRIBUTEVO.....	120

Figurförteckning

FIGUR 1: IIPAX-ÖVERSIKT.....	3
FIGUR 2: IIPAX OBJEKTMODELL	4
FIGUR 3: LIVSCYKEL FÖR TILLSTÄNDSLÖS SESSIONSBÖNA.....	15
FIGUR 4: LIVSCYKEL FÖR SESSIONSBÖNA MED TILLSTÅND.....	15
FIGUR 5: LIVSCYKEL FÖR ENTITETSBÖNA.....	18
FIGUR 6: JDBC-DRIVRUTIN TYP 1	25
FIGUR 7: JDBC-DRIVRUTIN TYP 2	25
FIGUR 8: JDBC-DRIVRUTIN TYP 3	26
FIGUR 9: JDBC-DRIVRUTIN TYP 4	27
FIGUR 10: KOMPILERING AV SQLJ.....	29
FIGUR 11: TILLGÅNGLIGA TEKNIKER	31
FIGUR 12: DESIGN	43

Tabellförteckning

TABELL 1: DATABASTABELLEN PERSONER	7
TABELL 2: DATABASTABELLEN PERSONER (MED ID)	8
TABELL 3: DATABASTABELLEN FÖRENINGAR	8
TABELL 4: DATABASTABELLEN MEDLEMSKAP	9
TABELL 5: DATABASTABELLEN AUTHORS	45
TABELL 6: DATABASTABELLEN BOOKS.....	45
TABELL 7: DATABASTABELLEN ATTRIBUTES	46

TABELL 8: TESTFALL.....	55
TABELL 9: TESTADE IMPLEMENTATIONER.....	55
TABELL 10: SAMMANSTÄLLNING AV RESULTAT FÖR TESTFALL 10–21 OCH 50–61	68
TABELL 11: SAMMANSTÄLLNING AV RESULTAT FÖR TESTFALL 30–43.....	69

Diagramförteckning

DIAGRAM 1: 100 EXEKVERINGAR AV TESTFALL 10	56
DIAGRAM 2: 10 EXEKVERINGAR AV TESTFALL 30	57
DIAGRAM 3: TESTFALL 10–21.....	58
DIAGRAM 4: TESTFALL 50–61.....	59
DIAGRAM 5: TESTFALL 30 LISTA ALLA FÖRFATTARE	60
DIAGRAM 6: TESTFALL 31 LISTA ALLA BÖCKER.....	61
DIAGRAM 7: TESTFALL 40 SÖK FÖRFATTARE	62
DIAGRAM 8: TESTFALL 41 SÖK BÖCKER MED MATCHANDE TITLAR.....	63
DIAGRAM 9: TESTFALL 43 SÖK BÖCKER MED MATCHANDE PRIS	64
DIAGRAM 10: SAMMANSTÄLLNING AV TESTFALL 10–21 SAMT 50–61	66
DIAGRAM 11: SAMMANSTÄLLNING AV TESTFALL 10–21 SAMT 50–61 I STÖRRE SKALA.....	66
DIAGRAM 12: SAMMANSTÄLLNING AV TESTFALL 30–43.....	67

1 Inledning

I detta kapitel ges översiktlig information om arbetets och rapportens uppläggning tillsammans med läsanvisningar.

1.1 Om Ida Infront

Examensarbetet är utfört på företaget Ida Infront AB (fortsättningsvis kallat Ida) i Linköping. Ida har ca 100 anställda, och kontor finns även i Stockholm och Sundsvall. Ida bygger systemlösningar för komplexa och informationsintensiva verksamheter, och bland kunderna finner man till exempel Riksskatteverket och Rikspolisstyrelsen.

1.2 Läsanvisningar

Andra kapitlet ger bakgrund till arbetet och beskriver problem och arbetsmetod.

Kapitel tre innehåller tämligen grundläggande information om relationsdatabaser, SQL, EJB och JDBC. Kapitlet kan hoppas över om läsaren redan är bekant med dessa.

I kapitel fyra görs en genomgång av några för arbetet intressanta tekniker tillsammans med designmönster och optimeringsmetoder.

Femte kapitlet beskriver design och implementation av testmodellen samt innehåller en testspecifikation.

I kapitel sex redovisas resultat från mätningar på provimplementationerna, såväl i diagram- som tabellform.

I kapitel sju ger författaren sina personliga kommentarer till testresultaten och anknyter till relaterade rapporter.

Kapitel åtta innehåller slutsatser och förslag till vidare arbete inom området.

1.3 Om rapporten

Rapporten är en del av examinationen i det obligatoriska examensarbetet om 20 poäng som ingår i civilingenjörsutbildningen Datateknik vid Linköpings tekniska högskola. Den är framställd under vår, sommar höst och vinter 2001.

1.3.1 Språk

Att skriva en rapport i ett datorrelaterat ämne på svenska är ingen lätt uppgift. Svenska uttryck har använts istället för sina engelska motsvarigheter i så stor utsträckning som möjligt, men avsikten har varit att inte gå över gränsen för vad som bara låter fånigt. Svenska datatermgruppen (<http://www.nada.kth.se/dataterm/>) har i vissa fall använts som referens. Deras arbete är dock långt ifrån komplett och kan behöva genomgå en del kritisk granskning innan det kan anses normgivande.

I de fall benämningen han/hon kunde ha varit passande har av läsbarhetsskäl istället ordet han använts i princip uteslutande.

1.3.2 Teknik

Rapporten är skriven i Microsoft Word 97, svensk version. Bilder och diagram är framställda i Visio 5.0 och Microsoft Excel 97. Viss bearbetning av data och text är gjord i Microsoft Office 2000.

Brödtexten är satt i Garamond, 14 punkter före utskrift till A4. Rubrikerna är satta i Verdana. Till kodexempel och liknande har använts Courier New. Text i diagram och figurer är satt i Verdana och Arial.

Kursiv stil har använts, förutom för betonade ord, för att markera då främmande språk använts och då nya uttryck introduceras.

Halvfet stil har nästan uteslutande använts för definitioner och förklaringar.

2 Problemformulering

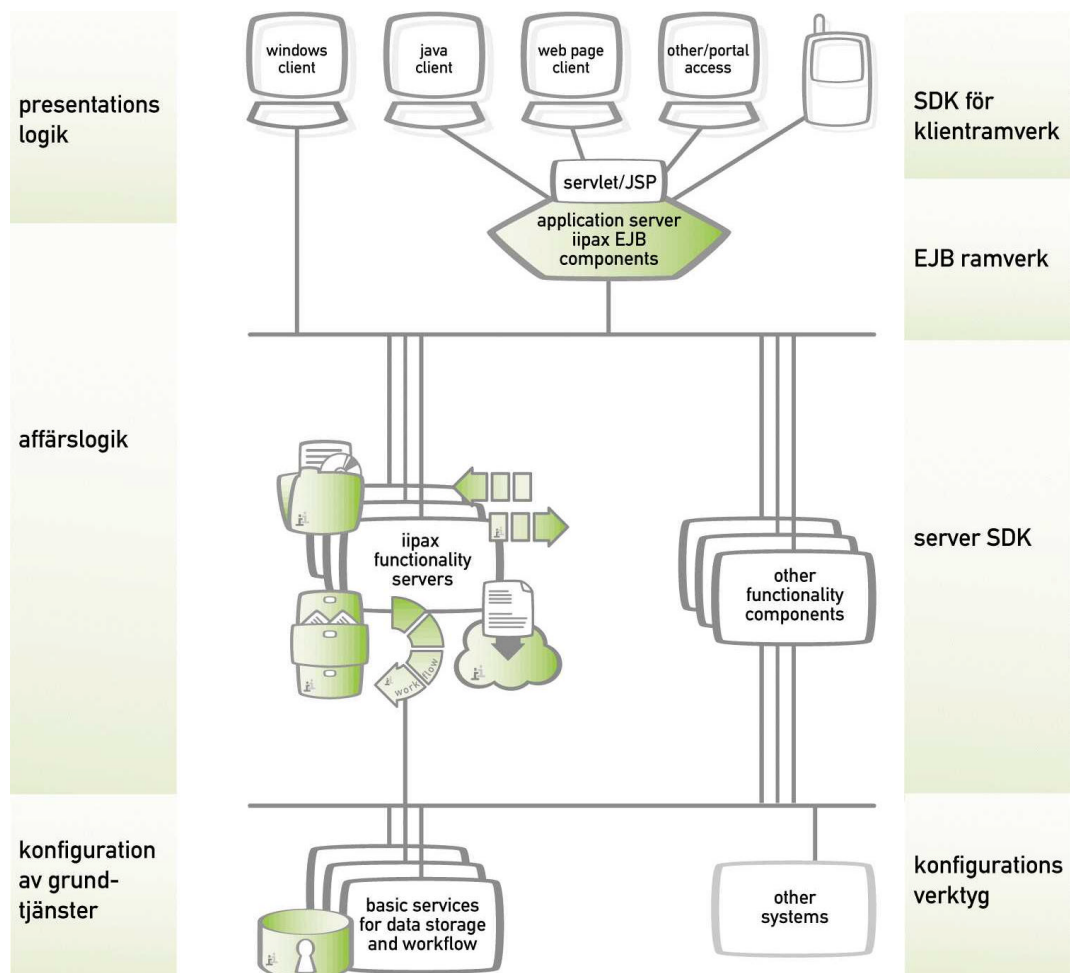
I detta kapitel ges arbetets syfte och bakgrund. Använd metodik och gjorda avgränsningar beskrivs.

2.1 Bakgrund

I detta avsnitt beskrivs kortfattat bakgrunden till examensarbetet.

2.1.1 iipax

Ida utvecklar *iipax*, ett ramverk för snabb utveckling och driftsättning av dokument- och ärendehanteringssystem.



Figur 1: iipax-översikt

(Bild: Ida Infront AB)

Primärt är iipax en samling serverprogram som är väl integrerade med varandra och kan köras både var för sig och tillsammans. Dessutom

erbjuder iipax ett ramverk för att bygga klienter för såväl WWW- som Java- och Windowsmiljö.

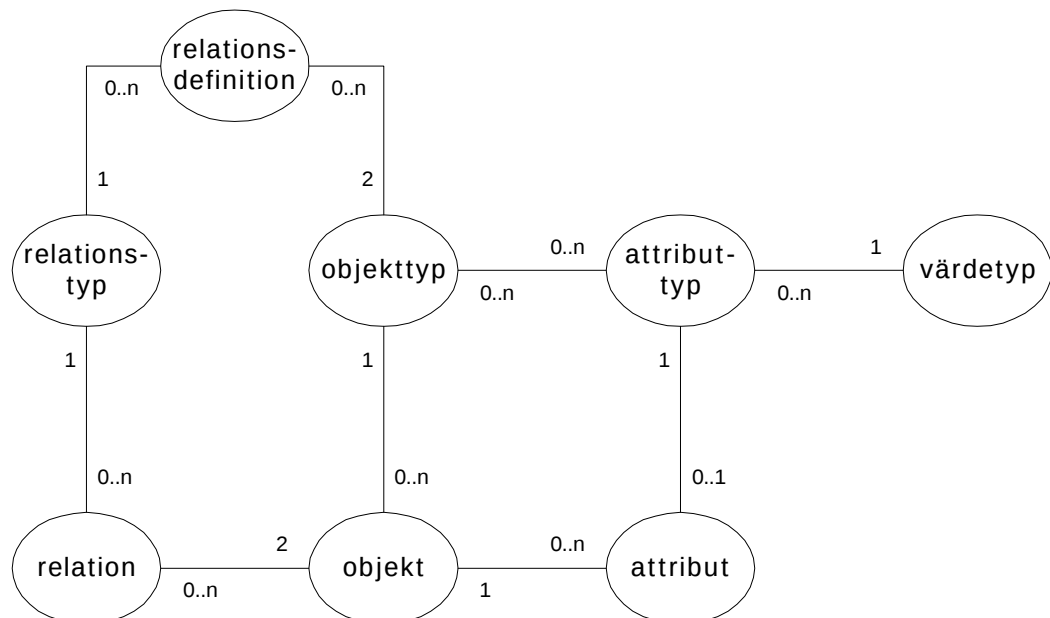
iipax är en flerlagerarkitektur med en server som är baserad på BEA:s applikationsserver *WebLogic Enterprise*, *BEA Tuxedo* och en klient för Windows som är utvecklad i C++.

iipax Java-ramverk

Ett av de senare tillskotten i iipax är ett Java-ramverk som används för att bygga klienter och affärslogik. Datalagringen sker idag i en relationsdatabas med *BEA Tuxedo* och *Jolt* som mellanlager mellan Java-delen och DMS (*Document Management System*) som är skriven i C. Idén med detta examensarbete är att utreda hur datalagring kan ske på bästa sätt direkt från Java utan att man behöver gå omvägen över kommersiella produkter och C-lager.

iipax objektmodell

iipax använder sig av en relativt enkel objektmodell (se Figur 2 nedan), enligt vilken generella objekt kan modelleras för att sedan lagras i AOS (*Advanced Object Storage*), en objektserver utvecklad av Ida. Modellen definierar begreppen objekt, attribut, värde och relation, och med hjälp av dessa kan såväl definitioner (klasser) som objekt (instanser) av olika slag hanteras på en ganska låg och därmed mycket generell nivå.



Figur 2: iipax objektmodell

2.2 Syfte

Syftet med detta arbete är att undersöka befintliga metoder för datalagring i J2EE-miljö och beskriva dem med lämpliga tillämpningsområden, för- och nackdelar och andra relevanta egenskaper. För Ida är det intressant med någon form av beslutsstöd när man skall införa datalagring direkt från Java i iipax Java-ramverk.

2.3 Problembeskrivning och arbetsmetod

I *Enterprise JavaBeans*-arkitekturen (EJB) finns en särskild sorts bönor (Java-komponenter) som används för att modellera och representera beständiga databasobjekt: entitetsbönor. För att koppla ihop dessa med databaser finns ett par olika metoder, vilka återfinns beskrivna i avsnitt 3.3 *Enterprise JavaBeans*.

För att slippa den overhead och de befarade prestandaproblem som hänger ihop med entitets-EJB kan man även ha "vanliga" Java-klasser som pratar med databasen genom t ex JDBC. I mån av tid skall även denna metod tas med i utredningen.

Skalbarhet och prestanda skall undersökas. Utvecklingstiden för Java-baserad datalagring är en annan intressant aspekt som bör behandlas.

2.3.1 Metod

Arbetet inleddes med en förstudie i vilken tillgänglig litteratur i ämnet (böcker, rapporter, artiklar och websidor) inventerades och de intressantaste delarna valdes ut för studier. En teknikinventering gjordes. Därefter vidtog en fördjupningsfas under vilken litteraturen studerades och andras resultat och slutsatser sammanfattades tillsammans med egna förutsägelser. En litteraturlista återfinns i kapitel 9 Referenser.

Under implementeringsfasen valdes några av de intressantaste teknikerna ut för testning. Testfall togs fram och en testtrigg designades. Provimplementationerna gjordes och testtriggen implementerades, varefter systemet genomgick testning och felrättning. När alla testfall befanns fungera utfördes mätningarna som ligger till grund för resultat och diskussion.

Efter utförda tester gjordes en ytlig besiktning av resultaten, vilka även diskuterades med handledarna. Därefter gjordes vissa ändringar i implementationen och några tester gjordes om.

Efter ytterligare diskussion med handledare och examinator förfinades mätmetoderna och vissa tester utfördes igen. Insamlade data åskådliggörs på olika sätt i kapitel 6 Resultat.

Under hela projektet pågick skrivandet av denna rapport.

2.4 Avgränsningar

Den underliggande databasen förutsätts uteslutande vara en relationsdatabas, och objektdatabaser behandlas inte.

Säkerhetsaspekter behandlas inte.

3 Bakgrundsteori

I detta kapitel ges grundläggande information om olika begrepp och tekniker som förekommer i rapporten. Vissa kunskaper, såsom grundläggande programmering i Java och kännedom om distribuerade system, förutsätts.

3.1 Relationsdatabaser

En databas brukar definieras som en strukturerad samling information. Den kanske mest använda databasmodellen idag är relationsmodellen, vilken introducerades i en artikel av Edgar F. Codd 1970 (Codd 1970). En relationsdatabas är baserad på denna modell, och består i grunden av tabeller i vilka varje rad motsvarar en entitet av något slag.

3.1.1 En exempeldatabas

Ett enkelt exempel på en tabell är en telefonlista med namn och nummer (se Tabell 1 nedan). I denna enkla relationsdatabas kan man med hjälp av en persons namn få reda på hans telefonnummer.

Namn	Nummer
Anna	1578
Karin	3108
Martin	5567
Per	3108

Tabell 1: Databastabellen Personer

I varje tabell i en relationsdatabas måste finnas en kolumn som inte innehåller några celler som är exakt lika. (Detta är inte helt sant, då även *sammansatta nycklar* existerar. Vi väljer av enkelhet att bortse från detta.) Denna kolumn kallas primärnyckel, och behövs för att man helt säkert skall kunna skilja raderna från varandra. I exemplet (Tabell 1) är kolumnen Namn primärnyckel. Vi kan inte använda Nummer-kolumnen som primärnyckel, eftersom det finns två rader som har samma nummer. (Man kan gissa att Karin och Per bor ihop.)

Låt oss nu säga att vi vill lägga till ytterligare en person i databasen, och hon heter Karin. Det går inte utan att vi skapar en ny primärnyckel! Vi inför en kolumn som vi kallar Id.

Id	Namn	Nummer
1	Anna	1578
2	Karin	3108
3	Martin	5567
4	Per	3108
5	Karin	1723

Tabell 2: Databastabellen Personer (med Id)

Nu kan databasen hantera personer med samma namn, och de kan dessutom ha samma telefonnummer (hur vanligt det nu är).

En relationsdatabas med bara en tabell är kanske inte så intressant, så vi lägger till en egenskap hos varje person, låt oss säga en förening han är med i. Vi skulle kunna lägga till namnet på föreningen som en extra kolumn i tabellen Personer, men det skulle innebära att namnet på en förening som flera personer är med i förekom flera gånger i tabellen. För att undvika detta skapar vi en ny tabell som vi kallar Föreningar. Vi ger den redan från början en Id-kolumn att använda som primärnyckel.

Id	Namn
1	Medicinska föreningen
2	Röda Korsets första hjälpengrupp
3	Bilmekarklubben
4	Föreningen Vi som har examen

Tabell 3: Databastabellen Föreningar

Nu kan vi enkelt införa en kolumn i tabellen Personer, där vi anger Id-numret (ett så kallat *referensattribut* eller *främmande nyckel*) på den förening personen på raden är med i. Om vi tänker efter inser vi dock att det vore dumt att utforma databasen så att varje person endast kan vara med i *en* förening. Vi kan inte stoppa in flera data i samma cell, så vi får lösa det med ytterligare en tabell, vilken beskriver relationer mellan personer och föreningar:

Person_Id	Förening_Id
1	1
1	2
2	4
3	3
3	4
4	4
5	2

Person_Id	Förening_Id
5	4

Tabell 4: Databastabellen Medlemskap

Den snabbtänkte påpekar här att det inte finns någon kolumn som kan användas som primärnyckel, vilket är helt korrekt. Primärnyckeln får helt enkelt vara sammansatt av båda kolumnerna.

Med hjälp av de tre tabellerna Personer, Föreningar och Medlemskap kan databasen innehålla information om godtyckligt många personer och deras medlemskap i godtyckligt många föreningar. Den kan också ge oss svar på frågor som till exempel "Vilka telefonnummer har de som är med i Röda Korsets första hjälpengrupp", men för detta behöver vi ett *frågespråk*, till exempel SQL.

3.2 SQL

SQL står för *Structured Query Language*, och är det mest använda frågespråket för relationsdatabaser. Eftersom i princip alla databastillverkare har sina egna utökningar och finesser finns det många dialekter inom språket, men de enklaste funktionerna ser ut på samma sätt i alla dialekter. Vi kommer här att ta upp några exempel på enkla SQL-anrop, samtliga tillämpbara på exempeldatabasen i avsnitt 3.1.1 ovan.

3.2.1 INSERT

För att skapa en ny post i en tabell används kommandot INSERT. Låt oss säga att Per skaffar en bil och går med i Bilmekarklubben. Då vill vi lägga till en rad i tabellen Medlemskap, vilket vi gör på följande sätt:

```
INSERT INTO Medlemskap (Person_Id, Förening_Id) VALUES (4, 3);
```

På motsvarande sätt kan vi lägga till en ny person i databasen:

```
INSERT INTO Personer (Id, Namn, Nummer)
VALUES (6, 'Cecilia', '2607');
```

Notera värdet vi använder för Id i exemplet; eftersom Id-kolumnen används som primärnyckel måste vi ange ett tidigare oanvänt nummer.

3.2.2 SELECT

Det förmodligen mest använda SQL-kommandot är SELECT. Det används för att få ut information ur databasen, och kan ha många parametrar och tillägg. Grundsyntaxen är dock som följer:

```
SELECT <kolumner> FROM <tabeller> WHERE <villkor>;
```

Eftersom resultatet av ett SELECT-anrop i sig är en tabell kan man genom att kombinera flera SELECT-satser ställa mycket komplicerade frågor till databasen. Vi väljer här att ge ett par enkla exempel på frågor i naturligt språk och deras motsvarigheter i SQL.

”Vad har Per för telefonnummer?”

```
SELECT Nummer FROM Personer WHERE Namn = 'Per';
```

Resultat: 3108

”Vad heter de föreningar Anna är med i?”

```
SELECT Föreningar.Namn  
FROM Personer, Föreningar, Medlemskap  
WHERE Personer.Namn = 'Anna'  
AND Medlemskap.Person_Id = Personer.Id  
AND Medlemskap.Förening_Id = Föreningar.Namn;
```

Resultat: Medicinska föreningen
Röda Korsets första hjälpenrupp

I exemplet ovan måste alla tre tabellerna kombineras för att man skall kunna få ut den efterfrågade informationen. Indatat ('Anna') återfinns ju i tabellen Personer, domänen från vilken svaren kommer finns i tabellen Föreningar, och kopplingen mellan dem finns i tabellen Medlemskap.

3.2.3 UPDATE

När en rad i en relationsdatabas skall uppdateras används kommandot UPDATE. Syntaxen är ganska rättfram, så vi går på direkt med ett exempel:

```
UPDATE Personer SET Nummer = '1287'  
WHERE Namn = 'Martin';
```

När kommandot utförs kommer alla personer vid namn Martin i databasen att få telefonnumret 1287. I vårt fall finns bara en Martin, men hade det funnits flera hade vi behövt använda något annat

urvalskriterium än `Namn = 'Martin'` för att avgöra vilka rader som skall påverkas.

3.2.4 DELETE

För att ta bort en rad ur relationsdatabasen används kommandot `DELETE`. Syntaxen är ganska lik den hos `UPDATE`. Vi exemplifierar genom att ta bort en av personerna vid namn Karin. Eftersom det finns mer än en måste vi använda oss av `Id`-numret:

```
DELETE FROM Personer WHERE Id = 5;
```

Denna operation gör att tabellen `Medlemskap` kommer att innehålla trasiga referenser till `Personer`-tabellen, nämligen alla Karins medlemskap. Vi tar bort dem också:

```
DELETE FROM Medlemskap WHERE Person_Id = 5;
```

3.3 Enterprise JavaBeans

Enterprise JavaBeans, förkortat `EJB`, har snabbt skördat stora framgångar och blivit näst intill en *de facto*-standard i IT-branschen. När Java introducerades av Sun Microsystems i mitten av 1990-talet handlade det mest om grafiska användargränssnitt, applets och framför allt plattformsoberoende. Ett halvt decennium senare har tekniken mognat ordentligt och Java används i många större system.

3.3.1 Vad är EJB?

`EJB` är en standard som definierar en rad tillägg till Java. Första versionen (1.0) av specifikationen (Sun Microsystems 1997) kom i mars 1997, och då hade många tillverkare redan börjat använda sig av den i sina applikationsservrar, databaser och webbservrar. I skrivande stund finns version 2.0 ute i vad som kallas *proposed final draft 2*, vilket innebär att den snart är färdigställd. I den här rapporten är det dock version 1.1 från december 1999 (Sun Microsystems 1999) som används som referens.

Sun Microsystems Inc. skriver så här i början av `EJB`-specifikationen, version 1.1:

The Enterprise JavaBeans architecture is a component architecture for the development and deployment of component-based distributed business applications. Applications written using the Enterprise JavaBeans architecture are scalable, transactional, and multi-user secure. These applications may be written once, and then deployed on any server platform that supports the Enterprise JavaBeans specification.

Standarden definierar en distribuerad objektorienterad komponentmodell som ska möjliggöra att komponenter (bönor) kan utvecklas och köras i en EJB-server för att sedan flyttas över till och köras i en server från en annan tillverkare.

3.3.2 Vad är en EJB-server?

En EJB-server tillhandahåller en mängd tjänster såsom namntjänst, transaktionshantering, beständighet och säkerhet. Den innehåller även en EJB-container, i vilken bönorna existerar.

Namntjänst

En av de grundläggande funktionerna i en objektmiljö är en tjänst för att hitta objekt givet deras namn eller andra attribut. I en EJB-server tillhandahålls denna *naming service* eller namntjänst av *Java Naming and Directory Interface*, JNDI. JNDI är en standardutökning av Java och erbjuder ett standardiserat API för åtkomst av namn- och katalogtjänster av olika slag som till exempel COS, DNS, LDAP och NIS. EJB-servern innehåller en JNDI-tjänst i vilken bönor, servertjänster och andra resurser är organiserade i en katalogträdsliknande struktur.

För att göra en JNDI-uppslagning måste man först ha en kontext, d v s en startpunkt i JNDI-trädets hierarkiska struktur. Denna startpunkt representeras av en instans av Java-klassen *InitialContext* och erhålls genom anrop till den JNDI-tjänst som ansvarar för den resurs man vill hitta.

Transaktionshantering

För transaktionshantering i Java finns två olika API:n, JTA (*Java Transaction API*) som är ett högnivågränssnitt och JTS (*Java Transaction Service*) som utgör en samling lågnivågränssnitt som används av EJB-servern. JTS är i grunden baserat på CORBA:s *Object Transaction Service*, och möjliggör samarbete mellan olika EJB- och transaktionsservrar i globala transaktioner.

EJB-container

EJB-containern är den miljö i vilken enterprisebönorna existerar, och är således mycket central i EJB-servern. Den erbjuder transaktionsstöd och beständighetsfunktioner. Containern ansvarar också för börnas hela livscykel.

För att kommunicera med bönorna anropar containern de speciella *callback*-metoder som EJB-standarden kräver. Detta sker till exempel då bönorna övergår mellan olika tillstånd i livscykeln.

EJB-containern gör det även möjligt att nå installerade enterprisebönor från andra klienter via nätverk. Detta sker med hjälp av fjärrmetodanrop (RMI, *Remote Method Invocation*) och är helt transparent för klienten.

EJB-containern ansvarar också för säkerhet i form av åtkomstreglering med ACL:er (*Access Control Lists*). På detta sätt kan metoder begränsas till att bara kunna anropas av behöriga användare.

Utöver de tjänster och funktioner som EJB-standarden kräver av servern erbjuder de flesta EJB-servertillverkare en hel del extrafunktionalitet som till exempel klustringsstöd, inbyggd webserver, avancerad övervakning och integration med tidigare existerande affärssystem.

Instance pooling

Instance pooling är en teknik för att bättre utnyttja resurser av olika slag såsom till exempel databaskopplingar och enterprisebönor. Idén bygger på att en klient sällan har behov av att hålla en egen instans av en viss resurs under hela exekveringstiden utan resursen kan göras tillgänglig för andra klienter mellan anropen. På detta sätt kan EJB-containern tillfredsställa ett stort antal klienter med ett förhållandevis litet antal samtidiga instanser av en EJB. De bönor som för tillfället inte används sägs ligga i en pool, och i många fall kan denna pools storlek förändras dynamiskt för att anpassas till efterfrågan.

3.3.3 Anatomi hos EJB

En EJB består av fyra delar:

Homeinterface

Här finns de metoder som behövs för att skapa nya instanser av bönan och för att hitta redan existerande instanser. För entitetsbönor finns även

metoder för att hitta bönor innehållande önskade data. Homeinterfacet är registrerat i JNDI-trädet för att det skall gå att hitta.

Remoteinterface

Det här är bönans gränssnitt mot omvärlden. Endast de metoder som deklarerats i remoteinterfacet kan anropas av klienterna.

Implementation

Det är här implementationen av bönans metoder finns. Detta inkluderar förutom affärslogiken även de metoder som deklarerats av home- och remoteinterfacen samt vissa metoder EJB-containern kräver.

Installationsbeskrivning

Det här är en XML-fil som beskriver olika aspekter av hur bönan skall installeras i containern. Här hittar vi bland annat behörighets- och åtkomstkontroll tillsammans med andra instruktioner för hur bönorna skall hanteras när de väl är installerade. För entitetsbönor definieras även vilka attribut som ska vara beständiga och hur de är kopplade till databasen.

Det finns två huvudtyper av komponenter i EJB-standard, sessionsbönor och entitetsbönor, vilka beskrivs nedan.

3.3.4 Sessionsbönor

En sessionsböna, även kallad sEJB, definieras av att den implementerar gränssnittet `javax.ejb.SessionBean`. Sessionsbönor finns i två smaker, med och utan interna tillstånd. Eftersom en tillståndslös sessionsböna inte har några interna tillstånd kan den inte hålla några data mellan metodanrop. Tack vare att *instance pooling* används kan dock andra klienter utnyttja bönan mellan anropen.

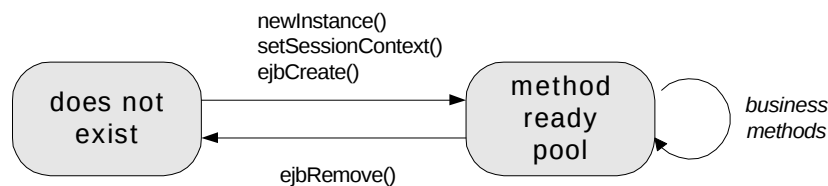
En sessionsböna med internt tillstånd kan däremot hålla data internt, och utgör på så sätt ett slags utökning av klienten på server-sidan. Den är unik, och klienten måste själv hålla reda på den. Sessionsbönans tillstånd går dock förlorat då containern avslutas.

Sessionsbönor med internt tillstånd är kopplade till den klient som anropar dem, och återgår till poolen när de inte längre behövs. De kan därför inte heller användas för att lagra eller överföra data mellan sessioner. De mest typiska användningsområdena för sessionsbönor är

att implementera affärslogik som till exempel beräkningsfunktioner och att hålla data under en session (till exempel en varukorg).

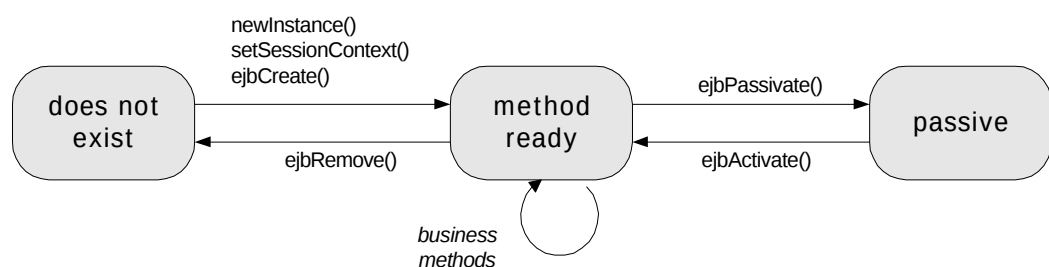
Varje klient (vilket också kan vara en annan enterpriseböna) som vill anropa en sessionsböna får en "egen" instans av den, och sålunda kan många instanser av samma sessionsböna finnas samtidigt i containern. Tack vare detta behöver sessionsbönsans metoder inte vara trådsäkra.

Sessionsbönsans livscykel



Figur 3: Livscykel för tillståndslös sessionsböna

Den tillståndslösa sessionsbönsans livscykel är med sina två tillstånd ganska okomplicerad. Då EJB-servern startas kommer den vanligtvis (beroende på inställningar) att skapa ett mindre antal instanser av bönan genom den metodsekvens som anges vid pilen från "does not exist" till "method ready pool". När en klient sedan anropar homeinterfacets `create()`-metod kommer anropet inte att propagera till bönan själv utan containern väljer ut en lämplig instans ur poolen och returnerar en referens till den. När klienten efter avslutade anrop av bönsans affärsmetoder anropar `remove()` är detta endast en signal till EJB-containern att klienten är färdig med bönan och innebär inte automatiskt att bönan förstörs.



Figur 4: Livscykel för sessionsböna med tillstånd

För sessionsbönan med tillstånd är livscykeln lite, men inte mycket, mer komplicerad. Den största skillnaden mot den tillståndslösa sessionsbönan är att *instance pooling* inte tillämpas här; då en böna skapats är den knuten till sin klient tills den förstörs. EJB-containern skapar alltså en ny instans då homeinterfacets `create()`-metod anropas, och förstör bönan då dess `remove()`-metod anropas.

Om EJB-containern vill spara systemresurser har den möjlighet att göra detta genom att passivera bönan, d v s spara undan dess tillstånd och ta bort den ur minnet. Containern måste då först förvarna bönan genom att anropa dess `ejbPassivate()`-metod. Bönan har då en chans att släppa eventuella resurser den haft öppna innan den försätts i passivt tillstånd. När klienten anropar en affärsmetod i den passiverade bönan aktiveras denna av EJB-containern. Så snart containern återställt bönan anropar den bönans `ejbActivate()`-metod, och bönan ges då möjlighet att återanskaffa de resurser den släppte vid passiveringen. Därefter utförs den anropade affärsmetoden.

3.3.5 Entitetsbönor

Det finns två olika sätt att se på entitetsbönor (även kallade eEJB): Från databashållet sett används entitetsbönor för att i objektmiljön representera entiteter i en databas. Sett ur ett annat perspektiv används entitetsbönor för att representera begrepp i den värld som modelleras, och databasen är bara ett hjälpmedel för att göra objekten beständiga. Vilket sätt man än vill se det på har varje entitetsböna något slags motsvarighet i en databas, och innehållet i entitetsbönan hålls synkroniserat med motsvarande innehåll i databasen.

Det bör dock noteras att ”databas” här inte nödvändigtvis behöver betyda databas i traditionell mening. Det kan också vara någon annan form av datalagringssystem, till exempel ett så kallat *legacy system* (i organisationen redan existerande system) eller till och med ett vanligt filsystem.

Alla entitetsbönor implementerar på ett eller annat sätt gränssnittet `javax.ejb.EntityBean`. Till skillnad mot sessionsbönor delas entitetsbönor av samtliga klienter, och normalt existerar endast en instans per unik identitet åt gången. Denna identitet definieras av den datamängd bönan representerar, till exempel en rad i en tabell över användare, och mer specifikt av sin primärnyckel. Primärnyckeln är vanligen en serialiserbar Java-lass med de attribut som behövs för att unikt identifiera bönan eller dess motsvarighet i databasen, men primärnyckelsklassen kan även utgöras av någon av Javas ”primitiva” klasser som till exempel `java.lang.Integer`.

Beständigheten hos en entitetsböna kan i princip skötas på två olika sätt, av EJB-containern eller av bönan själv. Dessa två varianter beskrivs nedan.

CMP

CMP, *Container-Managed Persistence*, innebär att EJB-containern sköter synkroniseringen mellan entitetsbönans innehåll och databasens innehåll utan att man som utvecklare behöver bry sig om det. Detta låter nästan lite magiskt, men det är det egentligen inte. I bönans installationsbeskrivning måste nämligen alla attribut som skall vara beständiga specificeras. Utifrån denna specifikation genereras sedan den kod som behövs för synkroniseringen mot databasen. Denna kodgenerering är långt ifrån enkel, och implementationerna varierar ganska mycket mellan olika tillverkare. Då CMP i version 1.1 av EJB-standarden endast kan hantera en databastabell per entitetsböna kan långt ifrån alla upptänkliga entitetsbönor realiseras med hjälp av CMP. Till exempel kan man inte göra en entitetsböna vars data representerar en *join* mellan två tabeller. I enklare fall kan dock CMP vara ett gott val då det förenklar programmeringsarbetet och dessutom inte kräver att bönan innehåller databas-specifik kod. Många applikationsservrar har dessutom mycket väl-optimerad kod för databasåtkomst och kan därför göra CMP-implementationer mycket snabba.

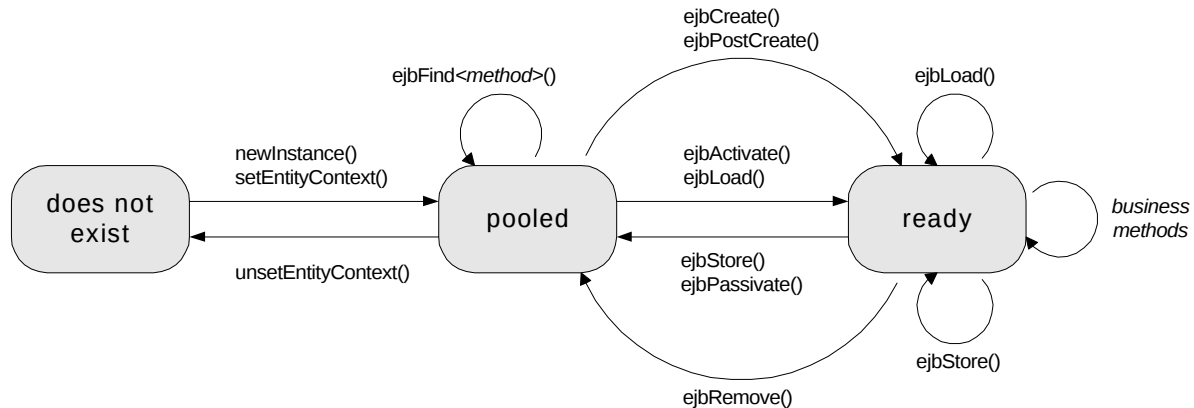
BMP

BMP står för *Bean-Managed Persistence*, vilket innebär att bönan (och därmed programmeraren) själv hanterar sin beständighet. Bönprogrammeraren måste alltså skriva den kod som gör det möjligt för bönan att hämta sina data ur databasen, uppdatera databasens innehåll, skapa nya data och genomföra sökningar. Till detta används till exempel JDBC (*Java DataBase Connectivity*, se avsnitt 3.4) eller SQLJ (se avsnitt 3.5). BMP ger programmeraren full kontroll över hur, när och vilka databasanrop som görs, vilket medger såväl komplicerade kopplingar mellan bönan och databasen som prestandaoptimeringar. Det är dock EJB-containern som initierar alla databasoperationer genom att anropa de callbackmetoder som krävs av EJB-standarden.

I och med att bönan själv hanterar beständigheten är den inte heller bunden till att använda just en (relations-)databas för sin beständiga lagring. Istället kan till exempel ett *legacy system* användas. I vissa fall kan en komplicerad design eller brister i EJB-containerns implementation göra användande av CMP omöjlig, och BMP måste användas.

Utveckling av entitetsbönor, även kallade eEJB, med BMP kräver att programmeraren har detaljkunskaper om databasens design, och koden blir knuten till densamma. Det är också, framför allt ifall JDBC används, relativt lätt att göra misstag under implementeringen och svårt att felsöka SQL-anropen.

Entitetsbönsans livscykel



Figur 5: Livscykel för entitetsböna

Från ”does not exist” till ”pooled”

När EJB-servern startar skapar den vanligtvis ett antal instanser av varje installerad entitetsböna och placerar dem i poolen. Poolens storlek kan bestämmas genom inställningar i servern. Instanser som befinner sig i poolen används för att utföra homeinterfacets findermetoder, vilket är mycket passande eftersom dessa metoder inte är beroende av interna tillstånd i bönan. Så länge entitetsbönonerna befinner sig i poolen är den inte kopplade till någon specifik entitet.

Från ”pooled” till ”ready” (”create”)

Då en klient anropar homeinterfacets `create()`-metod måste containern utföra flera operationer innan den kan returnera en referens till EJB-objektet. Först skapas ett EJB-objekt och en böninstant från poolen knyts till det. Därefter anropas bönsans `ejbCreate()`, vilken i BMP-fallet skall returnera ett färdigt primärnyckelobjekt. Om bönan använder CMP är det istället containern som efter anropet får instantiera en primärnyckel och fylla den med data. Varje böna kan ha flera create-metoder som tar olika argument. Primärnyckeln inkorporeras i EJB-objektet och ger det därmed en identitet. När detta är gjort anropas nästa callback-metod i bönan, `ejbPostCreate()`, vilken låter bönan utföra eventuella ytterligare initieringsoperationer. När allt detta är gjort returneras en referens till bönan och klienten kan börja använda den.

Väl i ready-tillståndet kan bönan utföra anrop till sina metoder i godtycklig omfattning och ordning. För att hålla bönsans data synkroniserade med databasen anropar EJB-containern BMP-bönsans `ejbLoad()` och `ejbStore()` vid strategiska tillfällen beroende på implementation

och bönans installationsbeskrivning. Även i CMP-fallet anropas dessa metoder direkt efter respektive omedelbart innan synkronisering med databasen sker, men då endast för att bönan skall kunna utföra eventuella operationer på datat innan det används i affärsmetoder eller innan det skall lagras i databasen.

Från "ready" till "pooled" (passivering och "remove")

När en entitetsböna inte används kan EJB-containern passivera den för att spara minne. Passiveringen inleds med ett anrop till bönans `ejbStore()`-metod för att säkerställa att bönan är synkroniserad med databasen (eller annan beständig lagring). Därefter anropar containern bönans `ejbPassivate()`-metod, i vilken bönan skall släppa eventuella allokerade resurser och förbereda sig på passivering. När detta gjorts kopplas bönans instans loss från EJB-objektet och återgår till pool-tillståndet. Observera att inget tillstånd i entitetsbönan sparas undan vid passivering såsom sker i fallet med sessionsbönor med tillstånd! Passivering kan naturligtvis inte ske då bönan är inblandad i en transaktion.

En annan anledning för bönan att gå från ready-tillståndet tillbaka till poolen är om klienten anropar dess `remove()`-metod för att därigenom radera de data bönan representerar. Anropet går vidare till `ejbRemove()` i vilken datat tas bort ur databasen. Bönan måste också släppa eventuella resurser precis som vid passivering, varför det kan vara praktiskt att metoden inbegriper ett anrop till bönans egen `ejbPassivate()`. När `ejbRemove()` avslutats flyttas bönan tillbaka till poolen och är redo att återanvändas.

Från "pooled" till "ready" (återaktivering och "find")

Då en passiverad entitetsböna skall aktiveras tar containern en godtycklig instans ur poolen och kopplar ihop den med EJB-objektet. Därefter anropas bönans `ejbActivate()`, i vilken bönan kan återta nödvändiga resurser och initiera ickebeständiga attribut. Därefter sker i CMP-fallet synkroniseringen med databasen, varefter `ejbLoad()` anropas för att upplysa bönan om att den är färdig att ta emot anrop av affärsmetoder. Även i BMP-fallet anropas `ejbLoad()`, men här måste bönan själv synkronisera sig med databasen. Därefter är bönan redo att utföra affärsmetoder igen.

Då en klient använder homeinterfacets findermetoder för att hitta bönor sker praktiskt taget samma sak som då passiverade bönor återaktiveras. EJB-objekt skapas för var och en av de funna bönorna och instanser ur

poolen kopplas till dem varefter rutinerna för aktivering utförs och bönorna är redo för användning.

Det bör noteras, att EJB-servern i vissa fall har inställningar för att undvika att ta instanser ur poolen ”i onödan”. Med en sådan inställning aktiverad kommer EJB-objekten att kopplas till instanser av bönan först då någon av remoteinterfacets metoder anropas.

Entitetsbönans död

När EJB-servern avslutas eller av annan anledning vill minska antalet instantierade entitetsbönor tas bönorna helt enkelt bort ur poolen. Containern anropar då bönans `unsetEntityContext()` för att upplysa den om att den kommer att förstöras, och därefter faller bönan snart offer för Javamaskinens *garbage collector*.

3.3.6 EJB och transaktioner

En av EJB-serverns viktigaste tjänster är transaktionshanteringen. Det enklaste för programmeraren är att låta servern sköta transaktionerna och bara ge den instruktioner genom de inställningar som kan göras i bönornas installationsbeskrivningar.

Transaktionsattribut

I EJB 1.1 finns sex olika transaktionsattribut som kan sättas såväl på varje metod för sig som på alla metoder genom användning av jokertecken (*) i installationsbeskrivningen. Då jokertecken används påverkas endast de metoder som inte har särskilt deklarerade transaktionsattribut, vilket gör systemet både enkelt och flexibelt. Tillsammans täcker de olika transaktionsattributen in samtliga möjliga situationer. Nedan följer en kort beskrivning av varje attribut.

Not Supported

Detta attribut innebär att metoden inte vill utföras i en transaktion. Om en transaktion är aktiv när metoden anropas kommer den att stoppas tills metoden returnerat, varefter transaktionen fortsätter.

Supports

Detta attribut anger att metoden inte kräver att utföras i en transaktion men att den ändå accepterar det. Om ett anrop till metoden ingår i en transaktion kommer den att innefatta metoden.

Required

En metod med transaktionsattributet *Required* måste utföras i en transaktion. Om klientens anrop inte ingår i någon transaktion kommer EJB-containern att skapa en ny transaktion som löper över metodens exekvering, i annat fall innefattar klientens transaktion även den anropade metoden.

Requires New

En metod med detta transaktionsattribut kommer alltid att utföras i en egen transaktion oberoende av om anropet ingår i en transaktion eller inte. Om en transaktion redan är aktiv stoppas den tills metoden returnerat och återupptas då.

Mandatory

Det här attributet kräver att anropande klient redan befinner sig i en transaktion. Om så inte är fallet startas ingen ny transaktion utan anropet resulterar i en `javax.transaction.TransactionRequiredException`.

Never

Detta är raka motsatsen till *Mandatory*. Om metoden anropas av en klient som befinner sig i en transaktion kastas ett `RemoteException`, i annat fall utförs metoden .

Sessionsbönor kan hantera sina egna transaktioner

I EJB 1.1 måste entitetsbönor använda sig av CMT, *Container-Managed Transactions*. Däremot kan sessionsbönor nyttja BMT, *Bean-Managed Transactions*, och därmed hantera sina egna transaktioner. Detta deklarerar i installationsbeskrivningen, och sker sedan med hjälp av `javax.transaction.UserTransaction`-objekt. Naturligtvis kan tillståndslösa sEJB (sessionsbönor) endast ha transaktioner som påbörjas och avslutas i samma metod.

Isoleringsnivåer och lås

När två eller flera transaktioner opererar på samma data kan situationer uppkomma där det inte är självklart hur datat ser ut för alla inblandade. För att reglera detta talar man ofta om fyra olika isoleringsnivåer:

Read Uncommitted innebär att en transaktion direkt läser data en annan aktiv transaktion förändrat. *Read Committed* innebär att data som

förändrats av en annan transaktion inte kan läsas förrän den avslutats. *Repeatable Read* innebär att transaktionen inte får ändra data som lästs av en annan transaktion. Slutligen finns också *Serializable* som ger transaktionen ensamrätt på data för både läsning och skrivning, vilket betyder att transaktionen inte kan samexistera med andra transaktioner som behandlar samma data.

Dessa isoleringsnivåer används av JDBC, och stöds av de flesta databaser. I databasen realiseras isoleringsnivåerna genom en samling olika lås, vilka förhindrar läs- och skrivoperationer på databasens data.

I normalfallet behöver programmeraren inte bry sig om isoleringsnivåerna i EJB 1.1, men ifall en böna använder sig av BMT måste isoleringsnivån anges genom inställningar i EJB-servern. EJB-servern kan också ha inställningar som styr isoleringsnivån för entitetsbönor. Exempelvis ger WebLogic 5.1 möjlighet att använda *Read Uncommitted* istället för det normala *Read Committed*.

3.4 JDBC

JDBC är ett API som tillhandahåller grundläggande funktionalitet för databasåtkomst från Java. API:t består av en uppsättning klasser och interface som kan köras i en JVM på vilken hård- och mjukvaruplattform som helst. Eftersom gränssnittet är standardiserat tillåter JDBC att anrop kan ske på samma sätt mot många olika databastyper från olika tillverkare. Det som krävs är en drivrutin som utgör gränssnitt mot databasen, se avsnitt 3.4.2 Drivrutiner nedan.

Mycket förenklat gör JDBC följande:

1. Skapar en förbindelse med en databas
2. Skickar förfrågningar och uppdateringskommandon till databasen
3. Behandlar resultaten

3.4.1 Exempel

Nedan följer ett enkelt exempel på hur JDBC kan användas för att skapa en uppkoppling mot en databas, ställa en enkel fråga och presentera resultatet.

```
// Skapa en uppkoppling till databasen
Connection conn =
    DriverManager.getConnection("jdbc:oracle:oci:logi
        nnamn/hemligt@medlemsdatabasen");
// Skapa en frågesats
```

```

Statement s = conn.createStatement();
// Utför anropet till databasen och ta hand om
// resultatet
ResultSet rs =
    s.executeQuery("SELECT namn FROM medlemmar");
// Gå igenom raderna i resultatet
while (rs.next()) {
    // Skriv ut värdet i kolumnen "namn"
    System.out.println (rs.getString("namn"));
}
// Stäng frågesatsen och kopplet
s.close();
conn.close();

```

Om allting fungerar som det skall blir resultatet en utskrift av alla namn i tabellen medlemmar.

Förberedda satser

För att effektivisera databasanropen kan man använda sig av så kallade *prepared statements*, d v s SQL-satser som i förväg förbereds syntaktiskt. Dessa kan sedan användas flera gånger med varierade parametrar utan att databasen behöver göra upp en ny frågeplan varje gång.

Exempel:

```

PreparedStatement ps = conn.prepareStatement(
    "UPDATE accounts SET balance = ? " +
    "WHERE id = ?");
// Iterera över alla konton
for (int i=0; i<accounts.length; i++) {
    // Sätt parametervärden
    ps.setFloat(1, accounts[i].getBalance());
    ps.setInt(2, accounts[i].getId());
    // Utför uppdateringen
    ps.execute();
    // Nollställ parametrarna inför nästa konto
    ps.clearParameters();
}
ps.close();

```

Här återanvänds alltså den förberedda satsen för alla konton, och databasen behöver bara tolka den och göra upp en frågeplan en gång.

Lagrade procedurer

Ett annat sätt att optimera databasanrop är att använda lagrade procedurer. Dessa är i förväg inmatade och kompillerade SQL-procedurer i databasen som kan anropas från SQL och därmed JDBC.

Förutom prestandavinster ger användandet av lagrade procedurer även möjlighet att få tillbaka resultat av till exempel ett UPDATE-anrop utan att behöva utföra en extra SELECT-sats.

Exempel:

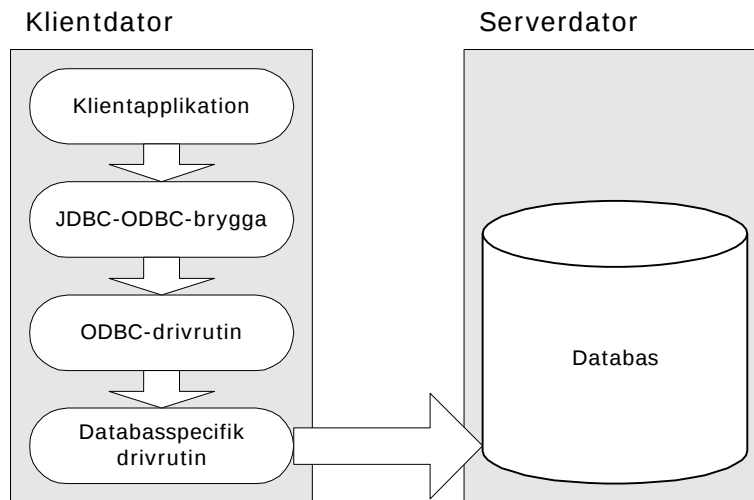
```
// Skapa en anropssats
CallableStatement cs =
    conn.prepareCall("{CALL do_stuff[(?, ?)]}");
// Tala om att andra parametern skall användas för att
// returnera flyttal
cs.registerOutParameter(2, java.sql.Types.FLOAT);
for (int i=0; i<accounts.length; i++) {
    // Sätt parametervärde
    cs.setInt(1, accounts[i].getId());
    // Utför anropet
    cs.execute();
    // ta hand om returvärdet
    accounts[i].setBalance(cs.getFloat(2));
}
ps.close();
```

Syntaxen {CALL do_stuff[(?, ?)]} är databasoberoende och översätts av JDBC till den anropade databasens egen syntax för anrop av lagrade procedurer.

3.4.2 Drivrutiner

Det finns fyra olika typer av JDBC-drivrutiner, vilka alla har sina för- och nackdelar:

Typ 1, JDBC-ODBC-brygga

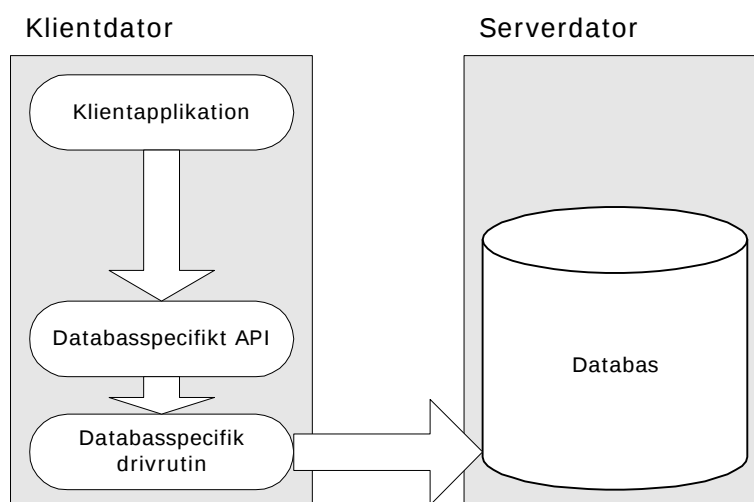


Figur 6: JDBC-drivrutin typ 1

Typ 1-drivrutinen översätter alla JDBC-anrop till ODBC-anrop och skickar dem vidare till ODBC-drivrutinen på datorn. Detta innebär alltså att en sådan måste vara installerad, och prestanda blir på grund av översättningsproceduren ofta mycket dåliga. Även resultaten genomgår översättning, men då åt motsatt håll. ODBC står för *Open DataBase Connectivity* och är ett standardiserat C-API för databaskonnektivitet.

Fördelen med den här typen är att i princip alla databaser redan har ODBC-drivrutiner, och om de finns så fungerar JDBC-drivrutinen. Detta kan vara intressant när det gäller äldre databaser. Typ 1-drivrutinen tillhandahålls av Sun.

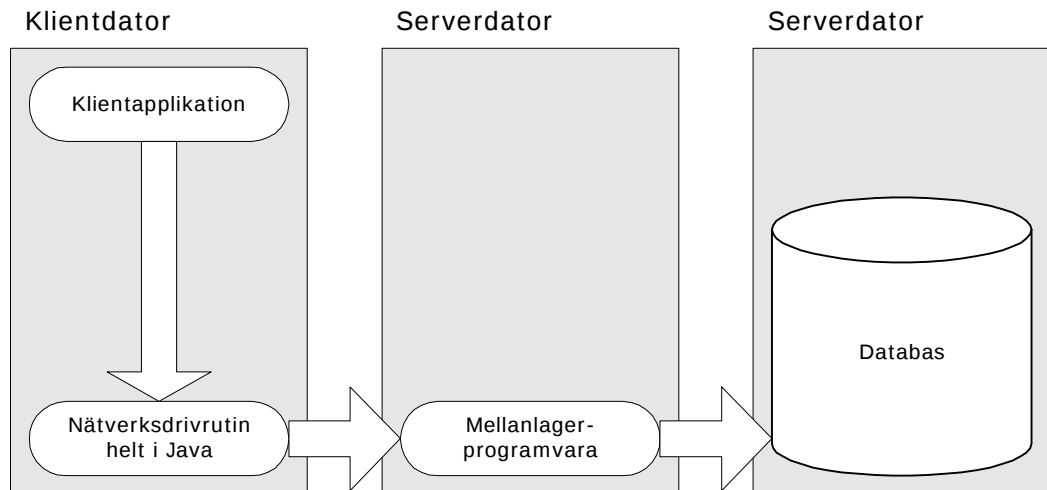
Typ 2, databasspecifikt API



Figur 7: JDBC-drivrutin typ 2

Även typ 2-drivrutinen översätter anrop, men till databasspecifika anrop som sedan skickas direkt till databasservern. För att denna kommunikation skall fungera krävs att databasspecifik icke-Java-kod exekveras på klientsidan.

Typ 3, databaskoppling med mellanlager

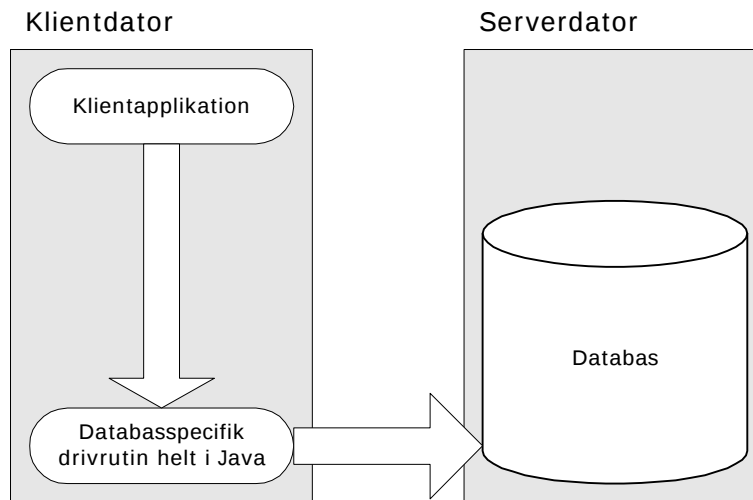


Figur 8: JDBC-drivrutin typ 3

Det här är en trelager-lösning, där mellanlagret utgörs av en server som tar emot uppkopplingar via nätverk från klienten och vidarebefordrar dem till databasservern över ett databasspecifikt protokoll. Mellanlagret kan vara mer eller mindre avancerat, och kan till exempel implementera cachning av uppkopplingar och resultat, lastbalansering och loggning.

Mellanlagret måste naturligtvis utvecklas av eller i samarbete med databastillverkaren, men den del av drivrutinen som körs på klientsidan kan alltså göras helt databasoberoende. Det finns också goda möjligheter att göra nätverksprotokollet mellan klient och mellanlager mycket effektivt.

Typ 4, direkt databaskoppling



Figur 9: JDBC-drivrutin typ 4

I en typ 4-drivrutin konverteras alla JDBC-anrop direkt till databas-serverns eget protokoll, så att klientprogram som använder sig av drivrutinen kan kommunicera direkt med databasen. Typ 4-drivrutiner är implementerade helt i Java för att undvika konverteringar mellan olika språk och uppnå plattformsoberoende.

Eftersom inga översättningar görs får den här typen av drivrutiner mycket goda prestanda. Dock är drivrutinen helt anpassad till den databas som används, och man behöver alltså flera olika drivrutiner om olika databaser skall användas. De flesta stora databastillverkare tillhandahåller dock typ 4-drivrutiner till sina databaser.

3.5 SQLJ – inbakad SQL

Eftersom det är ganska komplicerat att skriva SQL-frågor i Java har en standard utvecklats för inbakning av SQL arbetats fram av ett konsortium bestående av bland andra Oracle, IBM, JavaSoft, Microsoft och Informix. SQLJ introducerades våren 1997 och blev i slutet av 1998 accepterad som ANSI-standard (ANSI 1998).

SQLJ är egentligen tre olika standarder, men vi kommer här att närmare gå in bara på den första delen, vilken definierar en utökad syntax och semantik för att baka in statisk SQL i Java-kod. De andra två delarna behandlar lagrade procedurer i Java och en standard för att lagra Java-datatyper som objekt i en databas.

Det bör noteras att SQLJ inte ersätter JDBC, utan snarare kompletterar detsamma. SQLJ kan bara hantera statisk SQL, och JDBC måste användas då dynamiska anrop skall göras från Java.

Exempel

Betrakta följande Java-od som med hjälp av JDBC hämtar en cell ur en tabell i en relationsdatabas och stoppar den i en Java-variabel (exemplet inspirerat av (Cotner 1999)):

```
Connection con = getConnection();
String addr;
String name = "Petter";
java.sql.PreparedStatement ps = con.prepareStatement(
    "SELECT ADDRESS FROM EMP WHERE NAME=?");
ps.setString(1, name);
java.sql.ResultSet names = ps.executeQuery();
names.next();
addr = names.getString(1);
names.close();
System.out.println(name + " har adressen " + addr);
```

I SQLJ ser motsvarande kod ut så här:

```
Connection con = getConnection();
String addr;
String name = "Petter";
#sql (con) { SELECT ADDRESS INTO :addr FROM EMP WHERE
    NAME=:name };
System.out.println(name + " har adressen " + addr);
```

Det är alltså ganska stor skillnad i komplexitet och antal rader även för en relativt enkel uppgift.

Hur fungerar det?

När SQLJ-koden skall kompileras går det till så här:



Figur 10: Kompilering av SQLJ

Ett program som kallas översättare läser igenom `.sqlj`-filen och kontrollerar syntax och semantik hos SQLJ-koden. För att kunna kontrollera de Java-variabler som används anropar SQLJ-parsern även den vanliga Java-parsern. När syntax- och enklare typkontroll genomförts verifieras semantiken. I detta kan ingå att översättaren kopplar upp sig mot databasen för att kontrollera att typer överensstämmer mellan koden och databasen.

När syntax- och semantikkontrollen är färdig byter översättaren ut alla SQLJ-block mot anrop till SQLJ-metoder, och skriver alltsammans till en `.java`-fil. SQLJ-metoderna använder sig i sin tur ofta av JDBC-anrop.

Vi har nu fått fram en ”vanlig” `.java`-fil, och kan kompilera den som vanligt, till exempel med Suns SDK.

Under exekveringen av den kompilerade koden måste ett relativt kompakt SQLJ-bibliotek med implementationen av SQLJ-klasserna (i ren Java) finnas tillgängligt.

Fördelar

SQLJ förenklar SQL-programmering genom en språkutökning på högre nivå än JDBC.

SQLJ kontrollerar SQL-syntax, typning och databasschema vid kompilering, vilket medför att fel lättare upptäcks och färre programkörningsfel uppkommer.

SQLJ är heltäckande i bemärkelsen att det förutom vanliga SQL-frågor stöder transaktionshantering, DDL, DML och lagrade procedurer. Även databasspecifika utökningar som till exempel Oracles PL/SQL kan användas.

SQLJ-syntaxen är databasneutral, och medlemmarna i SQLJ-konsortiet använder sig av en gemensam referensimplementation av SQLJ-översättaren. Den kod SQLJ genererar är standardiserad, och ett SQLJ-program kan använda sig av godtycklig databasserver för vilken en SQLJ-körningsimplementation finns. Normalt producerar SQLJ-översättaren JDBC-kod, vilket innebär att så gott som alla databaser

stöds, men det finns också möjligheter att utnyttja tillverkarspecifika optimeringar och finesser.

Eftersom alla SQL-anrop som görs genom SQLJ är kompilerade är prestanda åtminstone lika bra som, men ofta bättre än, då JDBC används direkt.

Nackdelar

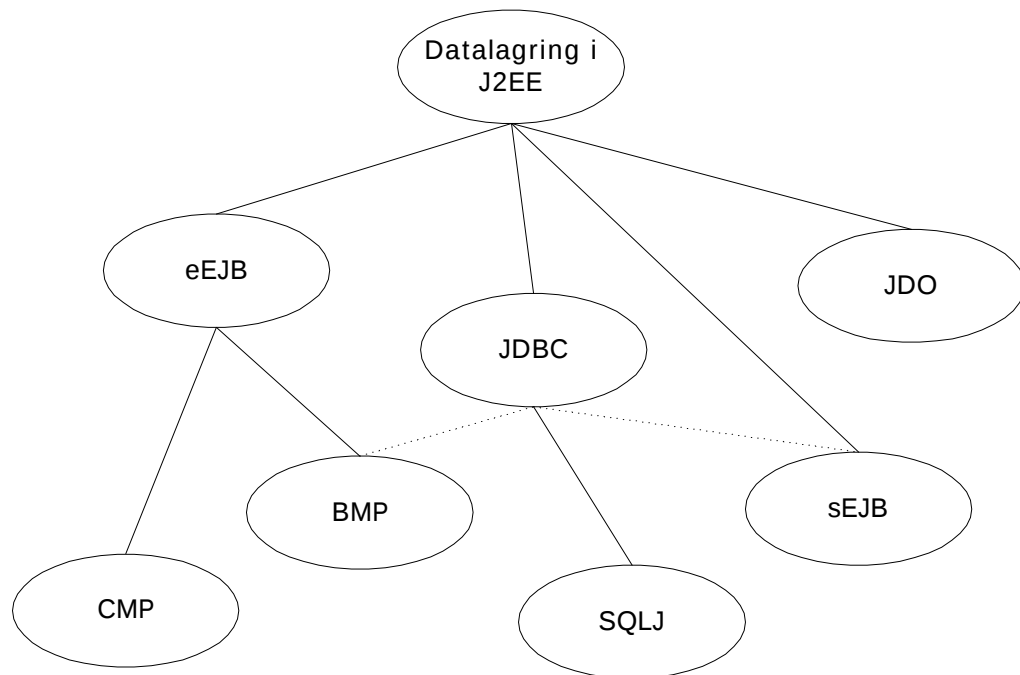
Förkompileringen/översättningen utgör naturligtvis ett extra moment i utvecklingsarbetet, men om utvecklingsmiljön har tillfredsställande stöd för förkompilering behöver detta inte vara något problem.

SQLJ är fortfarande inte helt accepterad som standard, och kan till exempel inte användas i JSP-miljö.

SQLJ kan bara användas för statisk SQL.

4 Genomgång av tillgängliga tekniker

Detta kapitel syftar till att ge överblick över de tekniker som är intressanta i sammanhanget. Dessutom ges en genomgång av några designmönster och optimeringsknep.



Figur 11: Tillgängliga tekniker

I Figur 11 ovan visas släktskap mellan de olika tekniker som funnits intressanta för detta arbete.

4.1 eEJB

Entitetsböror är en mycket intressant teknik som är väl lämpad för det användningsområde som skall testas. Den stora farhågan är att entitetsböror är alltför resurskrävande och att detta skall inverka menligt på prestanda, särskilt då sökningar i data resulterar i många träffar.

Många har också upplevt prestandaproblem i form av onödigt flitiga databasanrop (`ejbLoad()` och `ejbStore()`), men dessa problem bör gå att komma ifrån genom EJB-serverns optimeringsmöjligheter och genom korrekta installationsbeskrivningar.

Vidare kan den som inte arbetat med EJB tidigare lätt få för sig att det är svårt att sätta sig in i och komplicerat att implementera.

4.1.1 CMP eller BMP?

Valet mellan CMP och BMP är centralt då en eJB-lösning skall designas. Så länge man håller sig till EJB 1.1 är man i många fall tvingad att använda BMP, eftersom CMP helt enkelt inte kan uppfylla kraven på flexibilitet (se avsnitt 3.3.5).

Vad kan man då säga om prestandaskillnader mellan CMP och BMP? Det råder faktiskt delade meningar om det. Eriksson kommer i (Eriksson 2000) fram till att BMP är signifikant snabbare än CMP och dessutom skalar bättre då antalet samtidiga klienter ökar. Det sistnämnda hävdar han är en följd av det förra, något som man kanske kan diskutera. Eriksson anser att den enda anledningen till att välja CMP är att man vill bygga sitt system på kort tid och inte bryr sig så mycket om prestanda.

Sundberg använder i (Sundberg 2001) Erikssons resultat som en av sina utgångshypoteser, men upptäcker tidigt i sina försök att det snarare är tvärt om, det är CMP som är signifikant snabbare! Sundberg söker också stöd för sina resultat hos IBM, som tillverkat den testmiljö han använder (EJB-servern *WebSphere 3.5* och databasen *DB2 71*). Enligt honom har IBM optimerat koden för CMP-bönorna maximalt, och i kombination med IBM:s egen databas ger detta mycket goda prestanda. Inte ens med vissa optimeringar lyckades Sundberg få BMP-bönorna i klass med IBM:s CMP-bönor, och han rekommenderar därför BMP endast i de fall CMP inte kan ge önskad funktionalitet (komplexa affärsobjekt, beständighetslager som inte är databaser).

I artikeln *Seven Rules for Optimizing Entity Beans* (Sucharitakul 2001) är första regeln att använda CMP i de fall det är möjligt.

4.1.2 Finkornighet

När man designar sina entitetsböner måste man även bestämma sig för vilken nivå av *finkornighet* man skall ha. Vad det handlar om är ifall ett affärsobjekt skall modelleras som en entitetsböna eller delas upp i flera. Lösningen med en böna kallas alltså grovkornig medan lösningen med flera böner kallas finkornig. Ett exempel kan vara ett objekt som representerar en kund i ett system. Naturligtvis har då kunden vissa attribut såsom namn, inloggningsnamn och kundnummer. Men han har också en adress, eller kanske rent av flera adresser. Vi kan nu välja att låta adressen ingå som attribut i kundbönan, men vi kan också låta den bli en egen, självständig entitetsböna. Hur man bör göra beror på flera saker.

Ifall databasstrukturen redan är färdig bör man ta hänsyn till den; ligger adressfälten redan i samma tabell som kundens övriga data finns det

ingen anledning att bryta ut dem. Mer troligt är det kanske att adresserna ligger i en egen databastabell, och om vi då valt CMP för våra entitetsböner är vi helt enkelt tvungna att göra dem finkorniga eftersom CMP i EJB 1.1 inte klarar att hantera data i flera tabeller. Om vi bestämt oss för BMP får vi ställa oss en central fråga:

- Är adressen att anse som ett självständigt objekt?

Om adressen skall kunna fortleva även då kunden tas bort eller enkelt kunna flyttas över till ett annat kundobjekt är det en god idé att göra den till en egen entitetsböna. Väljer vi istället att se adressen som en del av kunden, ett *beroende objekt* som lever och dör med kunden, kan grovkornighet vara att föredra. Grovkorniga böner är också det som rekommenderas av specifikationen för EJB 1.1 (Sun Microsystems 1999).

Alla är dock inte överens om att grovkornighet är att föredra. I en artikel på webbplatsen TheServerSide.com anger Floyd Marinescu tre skäl till att inte använda grovkorniga entitetsböner (Marinescu 2000):

1. Svåra att implementera, affärslogiken klumpas ihop istället för att spridas ut på flera finkorniga böner.
2. I vissa sammanhang inte lika snabba som finkorniga böner.
3. Grovkorniga böner är ointuitiva, dålig koppling mellan databasschema och böner.

Angående punkt två menar han att i många tillämpningar utgörs de flesta transaktioner av listningar, och då också av samma data upprepade gånger. Med en finkornig lösning får då applikationsservern möjlighet att cacha data på ett effektivt sätt, vilket gör det mycket snabbt. I fallet med grovkorniga böner krävs sannolikt nya databasslagningar för varje listning. Även om man då själv implementerar en cache är det inte troligt att den blir effektivare än applikationsserverns.

4.2 Java/JDBC

Att implementera datalagringen utan EJB men med JDBC innebär att man får göra i allting själv; transaktioner, SQL-frågor, klientgränssnitt och eventuell cachning. En ren JDBC-implementation är troligen inte att föredra framför en EJB-implementation då man har ett något sånär stort system, men för prestandamätningar är det intressant att se hur lång tid JDBC-anrop och databasslagningar tar jämfört med EJB-operationer.

4.3 SQLJ

SQLJ förefaller vara mycket intressant, men har inte undersökts närmare inom ramen för detta examensarbete.

4.4 JDO

Java Data Objects är en specifikation för ett högnivå-API för att göra Java-objekt beständiga via transaktionella datalagringssystem. I slutet av maj 2001 existerar specifikationen i *proposed final draft*-version, och en enkel referensimplementation finns för nedladdning (Java Data Objects 2001). Anledningen till att specifikationen existerar är att expertgruppen som jobbar med den anser att de existerande beständighetshjälpmedlen (JDBC, SQLJ med flera) inte riktigt uppfyller behoven. Vad man framför allt vill uppnå är ett standardiserat sätt att göra objekt beständiga i många olika typer av datalagringssystem som inte kräver att programmeraren känner till hur lagringssystemet fungerar.

JDO har dock inte undersökts i detta examensarbete.

4.5 Tekniker, designmönster och optimering

Det finns många variationer, utökningar och designidéer kring de ovan nämnda teknikerna. I det här avsnittet återfinns några av dem med en kortare beskrivning.

4.5.1 Automatisk BMP

Automatisk BMP (*Automatic Bean-Managed Persistence*, *ABMP*) (Compoze Software 2001) är ett av Compoze Software Inc. patentsökt sätt att kringgå några av svårigheterna med EJB 1.1. Själva kärnan är en Java-klass som implementerar de nödvändiga EJB-callback-metoderna och som sedan används som superklass för entitetsböerna. Metoderna använder sig sedan av *Introspection* för att identifiera de attribut som skall vara beständiga. ABMP kräver att databasfälten som används har samma namn som entitetsböns attribut, men behöver i gengäld ingen installationsbeskrivning som styr beständighet. Förutom de nödvändiga metoderna implementerar ABMP några generella användbara finder-metoder.

ABMP kombinerar CMP-böns enkelhet med BMP-böns flexibilitet, och har också visst stöd för lagring av beroende objekt.

4.5.2 Förhindrande av onödiga databasoperationer

I vissa applikationsservrar kommer EJB-containern att anropa bönans `ejbStore()`-metod varje gång en affärsmetod utförts på en eEJB. Detta sker oavsett om något beständigt fält i bönan ändrats eller inte, och kan därför leda till onödiga skrivningsoperationer på databasen. För att förhindra att detta sker kan man använda sig av ett extra, icke-beständigt attribut i bönan som signalerar till `ejbStore()` huruvida bönans innehåll behöver skrivas till databasen eller ej.

Denna flagga kallas ofta `isModified`, och sätts med ett anrop till `setModified()` av de av bönans affärsmetoder som uppdaterat data-fält. `ejbStore()` kan sedan kontrollera flaggans värde genom att anropa `isModified()`, och då databasen uppdaterats nollställa flaggan med ett anrop till `unsetModified()`.

På motsvarande sätt anropas i vissa applikationsservrar bönans `ejbLoad()`-metod före varje anrop till dess affärsmetoder för att säkerställa att bönans data återspeglar databasens innehåll. Om man är säker på att ingen annan förändrar databasen kan man i åtminstone vissa applikationsservrar minska antalet `ejbLoad()`-anrop genom en inställning, till exempel i entitetsbönans installationsbeskrivning.

4.5.3 Värdeobjekt

När flera fält i en eEJB skall läsas eller skrivas är det ofta slöseri med resurser att hantera varje fält för sig, med ett fjärranrop per fält som resultat. Därför finns designmönstret värdeobjekt, vilka helt enkelt är serialiserbara Java-objekt som kan innehålla godtyckliga (serialiserbara!) attribut. Genom att skicka ett värdeobjekt som returvärde eller parameter i ett metodanrop minskar man både nätverkstrafiken och resursåtgången i det anropade objektet. (Value Objects 2001)

(Sundberg 2001) rekommenderar användning av värdeobjekt för alla läsningar av mer än ett fält, men avråder från att använda dem vid skrivning eftersom hans mätningar visade på sämre prestanda då värdeobjekt användes vid skrivning.

4.5.4 Fasadbönor

I många system ingår en klient av något slag som behöver använda sig av flera enterprisebönor, kanske både eEJB och sEJB. Detta kräver naturligtvis att klienten är bekant med bönornas gränssnitt, och alla anrop till dem sker dessutom i form av fjärranrop, vilket kan resultera i onödigt mycket nätverkstrafik. Ett enkelt sätt att slippa dessa problem är

att använda *fasadböner* (designmönstret *Session Facade* (Session Facade 2001)).

Principen är enkel: skapa en sessionsböna som klienten interagerar med, och låt sedan sessionsbönan sköta all kommunikation med andra berörda enterpriseböner. Beroende på de uppgifter som skall utföras kan det vara passande att använda sEJB med eller utan tillstånd.

4.5.5 Serialiserade attribut i databasen

Om det är ointressant att göra sökningar och läsa data direkt ur databastabeller kan man välja att serialisera vissa eller samtliga attribut och sedan lagra dem i ett binärfält i databastabellen. (Eriksson 2000) undersökte detta i kombination med entitetsböner och fann att denna metod ger vissa prestandaförbättringar jämfört med traditionell BMP/CMP när antalet samtidiga klienter ökar. Troligtvis beror denna egenskap på att endast ett fält per böna uppdateras i databasen istället för ett fält per beständigt attribut i bönan.

4.5.6 Lagring av relationer

I EJB 1.1-standarden finns inget stöd för aggregation och association mellan entiteter. Entitetsböner är designade för att hålla beständiga data, men inte referenser till andra böner. För att implementera dessa relationer finns det några olika sätt man kan göra på.

Man skiljer på dynamiska och statiska relationer. Statiska relationer är normalt fasta under entiteternas hela livstid, medan dynamiska relationer kan förändras under entiteternas livstid. En statisk relation behöver alltså bara skapas då entiteterna skapas, och sedan återskapas efter eventuell passivering. Dynamiska relationer kan däremot behöva uppdateras till exempel i anrop av affärsmetoder. I (Holland 2001) beskriver Grant Holland en relativt enkel metod som bygger på att man lagrar både en EJB-referens och för statiska relationer en beständig referens (primärnyckel) till databasen. Han utvecklar sedan metoden till att använda serialiserbara *Handles* och lagra dem i entitetsbönornas primärnyckelobjekt. På detta sätt reduceras och förenklas den kod som behöver skrivas.

4.5.7 Uppkopplingsstrategi

Oavsett vilken teknik man använder sig av för att göra sina data beständiga med hjälp av en databas måste man ofrånkomligen göra uppkopplingar till databasen. Dessa kan dock göras och hanteras på många olika sätt.

Connection pooling

Att göra en uppkoppling till en databas är tidskrävande, och bör därför undvikas. JDBC 2.0 erbjuder därför, liksom de flesta applikationsservrar, någon form av *connection pooling*, vilket innebär att ett objekt redan då servern startas skapar ett visst antal koppel mot databasen och sedan ”lånar ut” dessa till de metoder som behöver dem. På detta sätt behöver man varken skapa nya uppkopplingar eller stänga de redan öppna i onödan.

Även då man använder sig av *connection pooling* kan det dock vara värt att tänka över hur man hanterar sina databaskoppel. (Malani 2001) tar upp följande tre strategier för eEJB/BMP:

- Quick catch-and-release
- Live pool
- Servicing instance

Låt oss titta lite närmare på dessa strategier.

Quick catch-and-release strategy

Principen i den här strategin är att skaffa uppkopplingen så sent som möjligt, göra sina anrop och sedan släppa uppkopplingen så snart som möjligt. Detta innebär att `ejbCreate()`, `ejbRemove()`, `ejbLoad()`, `ejbStore()` och `ejbFind`-metoderna var och en måste innehålla kod för att skaffa ett koppel och släppa det. För att vara säker på att inga koppel tappas bort om ett fel skulle inträffa i metoden bör alla databasanrop ligga i ett `try-catch-finally`-block där kopplet släpps i `finally`-delen.

Live pool strategy

Den här strategin utnyttjar entitetsbönans livscykel, och ser till att databaskopplet finns under bönans hela livstid. Betrakta livscykeln i Figur 5 på sidan 4. Vi ser att databaskopplet behöver finnas tillgängligt i tillstånden *pooled* och *ready*, men inte i *does not exist*. Alltså låter vi helt enkelt metoden `setEntityContext()` anskaffa kopplet och `unsetEntityContext()` lämna tillbaka det.

Servicing instance strategy

Som vi såg i entitetsbönans livscykeldiagram behöver vi komma åt databasen från såväl *pooled*- som *ready*-tillståndet. I den här strategin ser vi dock på ganska skilda sätt på de två tillstånden. En böna kan vara *pooled* ganska länge utan att behöva kommunicera med databasen, så för att

bönan inte skall tjuvhålla på kopplet i onödan låter vi `ejbFind`-metoderna skaffa och släppa de koppel de behöver själva.

För att vi skall kunna komma till *ready*-tillståndet måste vi naturligtvis skaffa oss ett koppel även i metoderna `ejbCreate()` och `ejbActivate()`, men vi låter dem lagra kopplet i bönan så att det alltid finns tillgängligt i *ready*-tillståndet. Återlämnandet av kopplet sker istället i metoderna `ejbPassivate()` och `ejbRemove()`, när bönan återgår till att vara *pooled*.

Den här strategin kan alltså sägas vara en hybrid av de två tidigare nämnda.

När skall man då använda sig av de olika strategierna? (Malani 2001) ger följande rekommendationer:

- Om bönan beräknas ha ett litet antal instanser men många metoder som kräver databasåtkomst och som anropas ofta, välj *Live pool*-strategin.
- Om bönan beräknas ha många instanser men endast ett fåtal metoder med databasåtkomst som anropas sällan, välj *Quick catch-and-release*-strategin.
- I övriga fall, välj *Servicing instance*-strategin.

4.5.8 DAO

Ett problem med entitetsbönor är att de ganska intimt kopplar ihop affärslogik med den kod som används för att synkronisera data med den underliggande beständighetsmekanismen. Ett designmönster som tagits fram för att avhjälpa detta är DAO, *Data Access Object*. (Data Access Objects 2001) Mönstrets idé är att bryta ut koden för dataåtkomst och placera den i ett eget objekt kallat DAO. Entitetsbönan, vilken naturligtvis är av BMP-typ, behöver på så vis inte känna till hur beständigheten sköts, utan anropar bara standardiserade metoder i denna.

Nackdelen med DAO är den extra komplexitet mönstret tillför.

4.6 Sammanfattning

I det här kapitlet har vi gått igenom de tekniker som står till buds när det gäller datalagring i J2EE. Den mest intressanta tekniken är *Enterprise JavaBeans*, och det är också den som fått störst utrymme i avsnittet om optimering och designmönster.

Några av optimeringsmetoderna och designmönstren kommer att användas vid testimplementationerna i det här arbetet. Värdeobjekt och fasadböror utgör centrala delar i testdesignen, och `isModified()` används tillsammans med relevanta inställningar i EJB-servern för att minimera antalet onödiga läsningar och skrivningar i databasen.

5 Design och implementering

I detta kapitel beskrivs designen av den testmodell som används, testfall, vilka tekniker som valts ut för implementering samt designen av en testtrigg och en specifikation för testning.

5.1 Testmodell

En relativt enkel datamodell konstruerades, bestående av endast två entitetstyper vilka lagrades i tre tabeller i en relationsdatabas. Avsikten med denna modell är att skapa tre tabeller med inbördes relationer och variation i längd.

5.1.1 Entiteten Författare

En *författare* utgörs, förutom av ett id-nummer, endast av attributet *namn*. Författare lagras i sin helhet i databastabellen *Authors*, med fälten *id* och *name*.

5.1.2 Entiteten Bok

En *bok* har fyra attribut förutom id-numret: titel, utgivningsår, ISBN-nummer och pris. För att denna entitet skall bli lite intressantare (mer om det senare) lagras den uppdelad i två databastabeller. I tabellen *Books* lagras attributen *id* och *title*. I tabellen *Attributes* lagras utgivningsår, ISBN-nummer och pris som nycklade attribut med egna rader.

5.2 Testfall

En samling testfall valdes ut för testning. Testfallen är avsedda att avspegla normal användning av en databas samt representera flera olika typer av användning. Observera att testfallen inte numrerats löpande utan fått nummer efter den kategori de tillhör.

5.2.1 Läsning

10. Läs författare (internt attribut) med givet id
11. Läs bok (externa attribut) med givet id

5.2.2 Uppdatering

20. Uppdatera attribut i författare

21. Uppdatera externt attribut i bok

5.2.3 Listning

30. Lista alla författare

31. Lista alla böcker inklusive externa attribut

5.2.4 Sökning

40. Sök författare med matchande namn

41. Sök böcker med matchande titlar (sökning på internt attribut)

42. Sök böcker av viss författare (sökning på referens)

43. Sök böcker med matchande utgivningsår (sökning på externt attribut)

5.2.5 Skapande

50. Skapa författare

51. Skapa bok (inklusive attributen)

5.2.6 Borttagning

60. Ta bort författare (böckerna blir kvar)

61. Ta bort bok (attributen skall följa med)

5.3 Urval

Fyra huvudgrenar ur trädet i Figur 11 valdes ut till implementering; tre EJB-baserade och en icke-EJB-baserad. Två av EJB-varianterna använder sig av entitetsbönor, och dessa fall varierar ytterligare i fråga om finkornighet.

5.3.1 Entitetsbönor med CMP

En författare implementeras som en entitetsböna.

En bok implementeras som flera entitetsbönor, en "huvudböna" per bok och en "extraböna" per externt attribut.

5.3.2 Entitetsbönor med BMP

En författare implementeras som en entitetsböna.

En bok implementeras dels som en grovkornig entitetsböna innehållande även de externa attributen, dels som flera finkorniga entitetsbönor på samma sätt om i CMP-fallet.

Optimeringsmetoden *isModified* (se avsnitt 4.5.2) används.

5.3.3 Sessionsböna

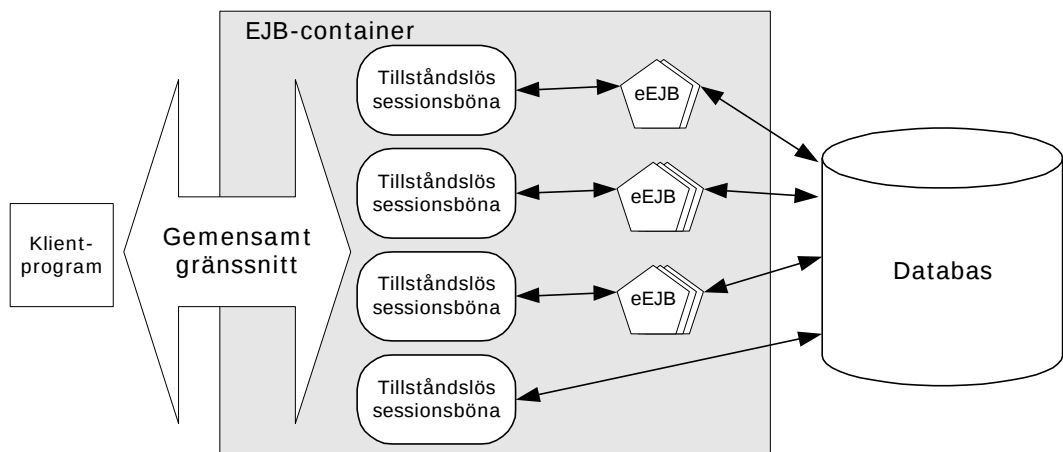
En sessionsböna används som verktyg för att hämta och uppdatera information om författare och böcker. Sessionsbönan använder JDBC för att kommunicera med databasen.

5.3.4 Java/JDBC

Som referens görs i mån av tid en icke-EJB-implementation av en Java-lass som motsvarar sessionsbönevarianten. Även här används JDBC.

5.4 Design

För att testa de valda teknikerna med testmodellen och testfallen i avsnitt 5.1 och 5.2 ovan designades en testtrigg enligt Figur 12 nedan.



Figur 12: Design

För var och en av de tre EJB-varianterna konstrueras en sEJB som tjänar som gränssnitt mot testklienten. I fallen med entitetsbönor är denna serverböna den som hanterar entitetsbönorna och presenterar data för klienten i form av värdeobjekt (se avsnitt 4.5.3). För sEJB-fallet är det serverbönan själv som kommunicerar direkt med databasen. Samtliga

varianter av serverbönan implementerar ett gemensamt Java-interface som definierar de metoder som skall finnas. På detta sätt behöver endast en testklient implementeras, och testklienten slipper också kommunicera med eEJB direkt, vilket skulle medföra prestandasänkningar (se avsnitt 4.5.4).

Gemensamt gränssnitt

Det gemensamma gränssnittet för sessionsbönorna utgörs av följande metoder:

AuthorVO	getAuthorById(int id)
AuthorVO	getAuthorByBookId(int id)
AuthorVO[]	getAuthorsByName(String pattern)
NumVO	searchAuthorsByName(String pattern)
AuthorVO[]	getAuthors()
BookVO	getBookById(int id)
BookVO[]	getBooksByAuthorId(int id)
NumVO	searchBooksByAuthorId(int id)
BookVO[]	getBooksByTitle(String pattern)
NumVO	searchBooksByTitle(String pattern)
BookVO[]	getBooks()
BookVO[]	getBooksByAttribute(String key, String value)
NumVO	searchBooksByAttribute(String key, String value)
AttributeVO[]	getAttributesByValue(String key, String pattern)
GenVO	addAuthor(AuthorVO)
GenVO	updateAuthor(AuthorVO)
GenVO	removeAuthor(int id)
GenVO	addBook(BookVO)
GenVO	updateBook(BookVO)
GenVO	removeBook(int id)
GenVO	addAttribute(AttributeVO)
GenVO	updateAttribute(AttributeVO)
GenVO	removeAttribute(int id)

Värdeobjekt

Returtyperna ovan är värdeobjekt eller vektorer av värdeobjekt. Värdeobjektens deklarationer återfinns i Appendix E.

Tidmätning

Tidmätning sker i serverbönan och mätvärdet returneras till klienten i ett fält i det returnerade värdeobjektet för lagring eller bearbetning.

Uppmätta värden avser alltså tiden det tar för serverbönan att lösa uppgiften, exklusive själva metदानropet från klienten. För tidmätning används anrop till `System.currentTimeMillis()`, vilket under Windows NT ger en noggrannhet om 10 ms.

5.4.1 Databasdesign

Databasen för testerna skall innehålla tre enkla tabeller enligt nedan. Varje tabell har en kolumn som innehåller primärnyckeln i form av ett heltal. De andra kolumnerna är av varchar-typ. Index skapas för att snabba upp sökningar.

Testerna skall genomföras med olika stora datamängder för att tabellängdens inverkan på resultaten skall kunna studeras. Data genereras med hjälp av ett litet program enligt avsnitt 5.5 Databasgenerator nedan.

Authors	
id (pk)	NUMBER
name	VARCHAR(100)

Tabell 5: Databastabellen Authors

Books	
id (pk)	NUMBER
author (fk)	REFERENCE
title	VARCHAR(100)

Tabell 6: Databastabellen Books

Attributes	
id (pk)	NUMBER
book (fk)	REFERENCE
key	VARCHAR(30)
value	VARCHAR(100)

Tabell 7: Databastabellen Attributes

Datotypen number ovan har 38 siffrors noggrannhet.

Attributtabellen har även ett *constraint* som gör att attributen automatiskt tas bort då den bok de tillhör tas bort.

För att underlätta sökningar i tabellerna skapas index för dem med avseende på `authors.name`, `books.title`, `books.author`, `attributes.book`, `attributes.[book, key]` och `attributes.[key, value]`.

Den SQL-kod som används för att skapa databasen återfinns i Appendix C.

5.4.2 Installationsbeskrivningar

Många egenskaper hos EJB-servern kan ställas in genom installationsbeskrivningarna för de böner som skall installeras. I det här avsnittet ges några exempel.

Serverböner

För serverbönerna anges att de skall vara tillståndslösa, att containern skall sköta transaktioner och att samtliga metoder kräver transaktionskontext. Exempel från BMPcServer:

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <description></description>
      <ejb-name>BMPcServer</ejb-name>
      <home>bjorn.ex.BMPcServerHome</home>
      <remote>bjorn.ex.BMPcServer</remote>
      <ejb-class>bjorn.ex.BMPcServerEJB</ejb-
class>
      <session-type>Stateless</session-type>
```

```

        <transaction-type>Container</transaction-
type>
    </session>
</enterprise-beans>
<assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>BMPcServer</ejb-name>
            <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
    </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

Entitetsbönor med BMP

För entitetsbönorna av BMP-typ vill vi ange att de skall vara just BMP-bönor samt att samtliga metoder stöder transaktioner. Exempel från BookBMPf:

```

<ejb-jar>
    <enterprise-beans>
        <entity>
            <description></description>
            <ejb-name>BookBMPf</ejb-name>
            <home>bjorn.ex.BookBMPfHome</home>
            <remote>bjorn.ex.BookBMPf</remote>
            <ejb-class>bjorn.ex.BookBMPfEJB</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>bjorn.ex.BookBMPfPK</prim-
key-class>
            <reentrant>false</reentrant>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
        <container-transaction>
            <method>

```

```
        <ejb-name>BookBMPf</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

Entitetsböror med CMP

När det gäller CMP-eEJB behöver vi ange vilka fält som skall vara beständiga. Vi tar BookCMP som exempel:

ejb-jar.xml

```
<ejb-jar>
  <enterprise-beans>
    <entity>
      <description></description>
      <ejb-name>BookCMP</ejb-name>
      <home>bjorn.ex.BookCMPHome</home>
      <remote>bjorn.ex.BookCMP</remote>
      <ejb-class>bjorn.ex.BookCMPEJB</ejb-class>
      <persistence-type>Container</persistence-
type>
      <prim-key-class>bjorn.ex.BookCMPPK</prim-
key-class>
      <reentrant>false</reentrant>
      <cmp-field>
        <field-name>author</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>title</field-name>
      </cmp-field>
      <cmp-field>
        <field-name>id</field-name>
      </cmp-field>
    </entity>
  </enterprise-beans>
```

```

<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>BookCMP</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Supports</trans-attribute>
  </container-transaction>
</assembly-descriptor>
</ejb-jar>

```

Som synes är fältnamnen det enda som sägs om de beständiga fälten i standard-installationsbeskrivningen (`ejb-jar.xml`). Hur fälten är kopplade till databastabeller varierar mellan olika EJB-container-tillverkare. I detta fall används BEA WebLogic, vilken använder sig av en fil vid namn `weblogic-ejb-jar.xml`.

weblogic-ejb-jar.xml

Här finns en rad intressanta inställningar vilka markerats i halvfet stil i listningen nedan. `max-beans-in-cache` anger hur många bönor som maximalt får finnas i minnet samtidigt. `is-modified-method-name` anger namnet på den `isModified()`-metod vi tillhandahåller, och tillsammans med `db-is-shared`, vilken i läge `false` säger att ingen annan förväntas förändra databasen, reglerar den hur ofta entitetsbönan skall synkroniseras med databasen (se avsnitt 4.5.2). Inställningen `finders-call-ejbload` styr huruvida entitetsbönor skall tas ur poolen direkt då en finder-metod exekveras eller om detta skall fördröjas tills de funna böornas metoder faktiskt anropas. Här hittar vi också en referens till ytterligare en installationsbeskrivningsfil som styr CMP mer i detalj, nämligen `weblogic-cmp-rdbms-jar-BookCMP.xml`.

```

<weblogic-ejb-jar>
  <weblogic-enterprise-bean>
    <ejb-name>BookCMP</ejb-name>
    <caching-descriptor>
      <max-beans-in-cache>100000</max-beans-in-cache>
    </caching-descriptor>
    <persistence-descriptor>
      <is-modified-method-name>isModified</is-modified-method-name>
    </persistence-descriptor>
  </weblogic-enterprise-bean>
</weblogic-ejb-jar>

```

```
<finders-call-ejbload>false</finders-call-  
ejbload>  
  <persistence-type>  
    <type-  
identifier>WebLogic_CMP_RDBMS</type-identifier>  
    <type-version>5.1.0</type-version>  
    <type-storage>META-INF/weblogic-cmp-  
rdbms-jar-BookCMP.xml</type-storage>  
  </persistence-type>  
  <db-is-shared>false</db-is-shared>  
  <persistence-use>  
    <type-  
identifier>WebLogic_CMP_RDBMS</type-identifier>  
    <type-version>5.1.0</type-version>  
  </persistence-use>  
</persistence-descriptor>  
  <jndi-name>BookCMP</jndi-name>  
</weblogic-enterprise-bean>  
</weblogic-ejb-jar>
```

weblogic-cmp-rdbms-jar-BookCMP.xml

Den här filen beskriver detaljerna för hur WebLogic skall hantera BookBMP-bönans beständighet med avseende på den underliggande relationsdatabasen. Vi går nedan närmare in på de markerade ställena i listningen omedelbart nedan.

```
<weblogic-rdbms-bean>  
  <pool-name>testPool</pool-name>  
  <table-name>books</table-name>  
  <attribute-map>  
    <object-link>  
      <bean-field>author</bean-field>  
      <dbms-column>AUTHOR</dbms-column>  
    </object-link>  
    <object-link>  
      <bean-field>title</bean-field>  
      <dbms-column>TITLE</dbms-column>  
    </object-link>
```

```
<object-link>
  <bean-field>id</bean-field>
  <dbms-column>ID</dbms-column>
</object-link>
</attribute-map>
<finder-list>
  <finder>
    <method-name>findByAttribute</method-name>
    <method-params>
      <method-param>java.lang.String</method-
param>
      <method-param>java.lang.String</method-
param>
    </method-params>
    <finder-query>
      <![CDATA[]]>
    </finder-query>
  </finder>
  <finder>
    <method-name>findByTitle</method-name>
    <method-params>
      <method-param>java.lang.String</method-
param>
    </method-params>
    <finder-query>
      <![CDATA[(like title $0)]]>
    </finder-query>
  </finder>
  <finder>
    <method-name>findByAuthorId</method-name>
    <method-params>
      <method-param>int</method-param>
    </method-params>
    <finder-query>
      <![CDATA[(= author $0)]]>
    </finder-query>
```

```
        </finder>
        <finder>
            <method-name>findAll</method-name>
            <finder-query></finder-query>
        </finder>
    </finder-list>
    <options></options>
</weblogic-rdbms-bean>
```

Först anges genom *pool-name* vilken *connection pool* containern skall använda. Poolen kopplas genom inställningar i WebLogic till en databasserver. *table-name* anger givetvis vilken tabell i databasen som skall användas för den här entitetsbönan. För varje beständigt fält i bönan anges ett *object-link-block*. I det markerade blocket kopplas fältet *author* ihop med databaskolumnen *AUTHOR*.

Det är också här bönans *finder*-metoder specificeras. Vi tar *findByTitle()* i det markerade *finder*-blocket ovan som exempel. Först talar vi om att *findByTitle()* skall ta ett argument av typen *java.lang.String*. Därefter beskriver vi med hjälp av WebLogics eget frågespråk WLQL (*WebLogic Query Language*) hur metoden skall fungera. WLQL har en Lisp-liknande syntax, och (*like title \$0*) i exemplet anger naturligtvis att vi söker de rader i databastabellen där fältet *title* matchar det argument vi givit. I SQL motsvaras den kompletta frågan av

```
SELECT id FROM books WHERE title LIKE '<argument>';
```

5.5 Databasgenerator

För att enkelt kunna fylla databasen med önskat antal data skrevs ett databasgenereringsprogram som givet ett visst frövärde genererar författare och böcker med hjälp av en pseudoslumpfunktion och lagrar dem i databasen. Om programmet matas med samma frövärde två gånger genererar det alltså samma databasinnehåll båda gångerna.

Programmet har förutom frövärdet två inparametrar: antal författare att skapa och antal böcker per författare. För varje bok genereras fyra externa attribut i attributtabellen.

Exempel:

```
java bjorn.ex.DatabaseGenerator 4711 10 12
```

ger en databas innehållande 10 författare, 120 böcker och 480 attribut.

Databasgeneratorns källkod återfinns i Appendix C.

5.6 Testklient

För genomförandet av testerna används en egen klientprogramvara skriven i Java. Den anropas med argument som specificerar vilken teknik och vilket testfall som skall testas, hur många gånger testfallet skall utföras samt nödvändiga parametrar för testfallet.

Klienten kopplar upp sig mot servern och anropar den sEJB som ansvarar för vald teknik. Returdatat tas om hand och analyseras, varefter data matas ut i strukturerat format.

5.7 Testspecifikation

Testningen skall tillgå så att tidsåtgången för att utföra ett testfall mäts i server-sEJB:en på serversidan. Varje mätning bör göras minst 100 gånger för att minimera inverkan från orelaterade faktorer såsom andra aktiva processer och diskaktivitet. Utifrån mätvärdena beräknas sedan aritmetiskt medelvärde, standardavvikelse och medianvärde att användas för jämförelser.

Alla implementerade enterprisebönor tillåts vara installerade i servern under testning, men servern startas om innan testning av en ny teknik påbörjas efter testning av en annan.

Onödiga processer på testmaskinen avslutas före testets början, och eventuell skärmläckare sätts ur funktion.

5.8 Teknik

Nedan följer en beskrivning av den tekniska utrustning som använts vid utveckling och testning.

5.8.1 Hårdvara

PC med 400 MHz Pentium II-processor och 256 MB RAM.

5.8.2 Mjukvara

Windows NT 4.0 sp6

Utveckling

Visual Café Enterprise Edition 4.1

GNU Emacs 20.7

Cygwin

Test

BEA WebLogic 5.1sp9 startad med 128/256 MB heap-storlek för JVM.
WebLogic stöder EJB 1.1.

Oracle 8i 8.1.7.

JDK 1.3.0 (HotSpot) från Sun.

6 Resultat

I detta kapitel presenteras resultaten från provimplementationerna.

6.1 Testfall och testade implementationer

I tabellerna nedan räknas testfallen respektive de testade implementationerna upp. För mer information, se kapitel 5.

Testfall		Beskrivning
Läsning	10	Läsa författare (endast interna attribut)
	11	Läsa bok (interna och externa attribut)
Uppdatering	20	Uppdatera (internt) attribut i författare
	21	Uppdatera externt attribut i bok
Listning	30	Lista alla författare
	31	Lista alla böcker inklusive externa attribut
Sökning	40	Sök författare med matchande namn
	41	Sök böcker med matchande titlar (internt attribut)
	42	Sök böcker av viss författare (internt attribut)
	43	Sök böcker med matchande pris (externt attribut)
Skapande	50	Skapa ny författare
	51	Skapa ny bok (inklusive externa attribut)
Borttagning	60	Ta bort författare (böckerna kvarstår)
	61	Ta bort bok (externa attribut raderas)

Tabell 8: Testfall

Beteckning	Beskrivning
BMPc	Entitets-EJB med BMP, grovkorniga böcker
BMPf	Entitets-EJB med BMP, finkorniga böcker
CMP	Entitets-EJB med CMP (finkorniga böcker)
SLSB	Tillståndslös sessionsböna

Tabell 9: Testade implementationer

6.2 Testresultat

För att öka överskådligheten har testresultaten delats in i tre olika grupper rörande enstaka data, listningar respektive sökningar.

De testfall som inbegriper enstaka data (avsnitt 6.2.1 nedan) exekverades mycket snabbt, ofta under 10 ms. Eftersom den mätmetod som användes (`System.currentTimeMillis()` i Java) inte kan mäta kortare tidsintervall än så förändrades implementationen av de berörda metoderna så att varje testfall kördes tusen gånger. Därefter dividerades den uppmätta tiden med 1000. Övriga testfall (listningar och sökningar) gav fullgoda mätvärden.

I testfallen 30–31 och 40–43 har linjära ”trendlinjer” ritats ut i diagrammen för att tydliggöra de mer eller mindre linjära samband som kan observeras mellan antal berörda poster i databasen och den tid som åtgått för att behandla dem. Dessa linjer är beräknade automatiskt av Microsoft Excel med minsta kvadratmetoden. Linjens ekvation är i vissa fall angiven i diagrammet, även om det troligtvis endast är riktningskoefficienten som är intressant (och pålitlig).

Varmt respektive kallt system

Det visade sig att de första exekveringarna i många fall tog avsevärt längre tid än de följande. Fenomenet kan beskådas i Diagram 1 nedan.

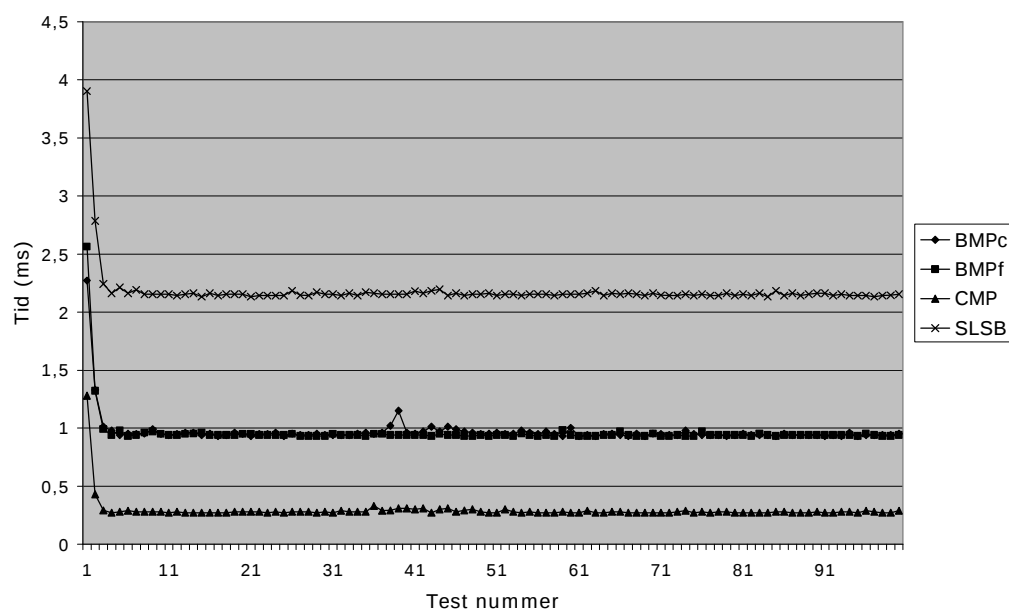


Diagram 1: 100 exekveringar av testfall 10

Vi ser att alla fyra teknikerna (BMPc och BMPf överlappar varandra fullständigt) inleder med två lite högre mätvärden och därefter ligger mycket stabilt på ett lägre medelvärde. Det bör noteras att varje mätvärde i Diagram 1 egentligen är medelvärdet av 1000 exekveringar.

Skillnaden mellan de första exekveringarna och de efterföljande är så stor att vi väljer att dela upp mätresultaten och redovisa dem var för sig. Eftersom varje serie omfattar 100 exekveringar väljer vi att dra gränsen mellan de första fem och de nästkommande 95, och benämner dessa *kallt system* respektive *varmt system*.

I de testfall som inbegriper listningar och sökningar har varje testfall exekverats 10 gånger per databasstorlek (se exempel i Diagram 2 nedan). De medelvärden som redovisas nedan är beräknade på tidsåtgången vid exekvering 3 till och med 10 för varmt system.

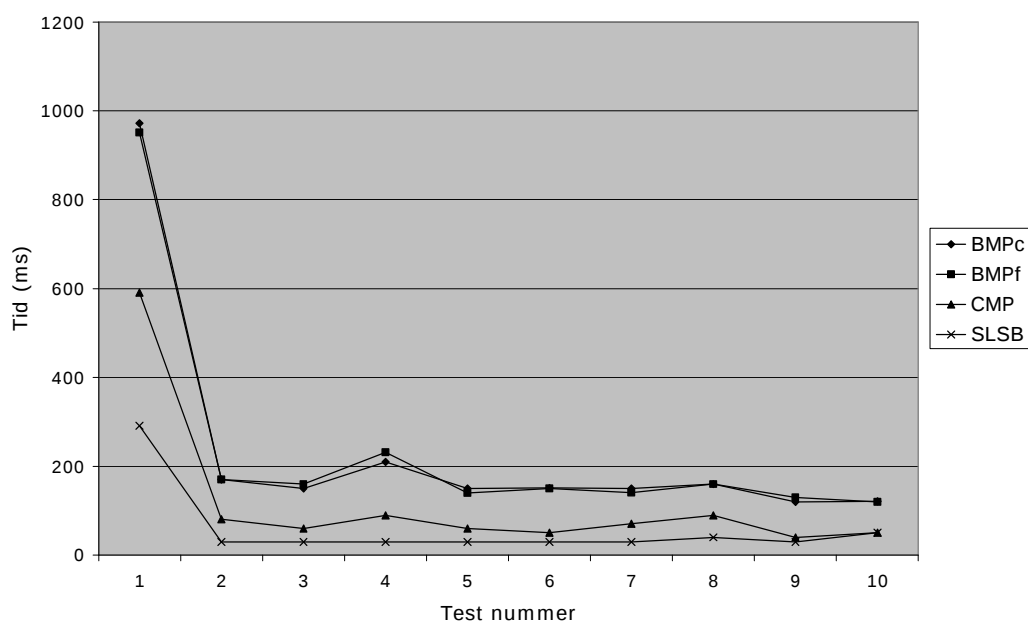
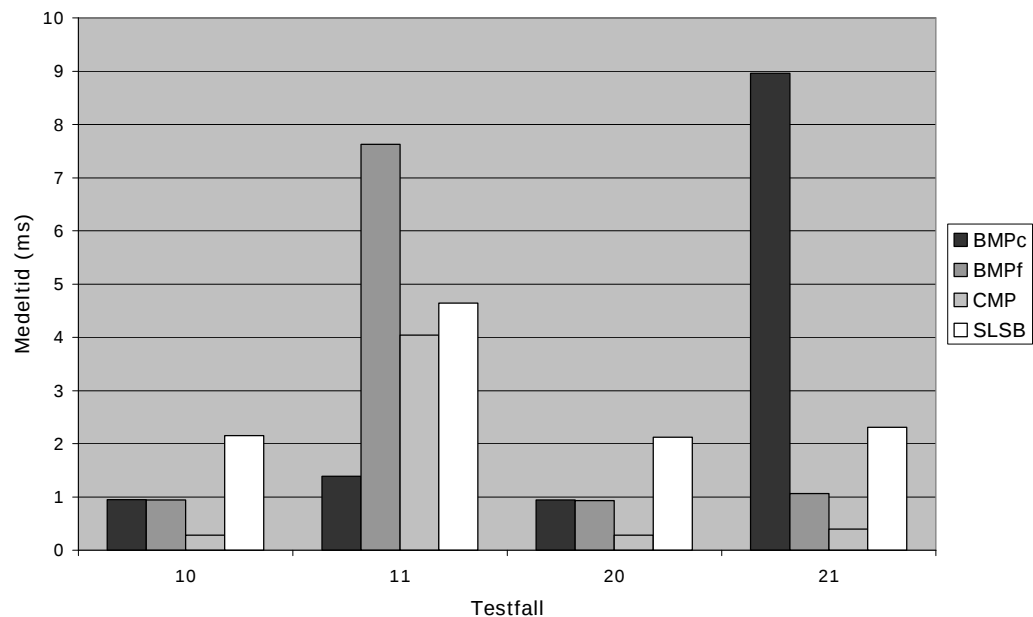


Diagram 2: 10 exekveringar av testfall 30

Resultaten för varmt system redovisas i diagramform nedan, och samtliga resultat återfinns även i tabellform i slutet av detta kapitel.

6.2.1 Testfall inbegripande enstaka data

De testfall som går ut på att en enskild post läses eller uppdateras visade sig vara så gott som helt okänsliga för variationer i databasens storlek. Detta beror på att endast en enkel uppslagning görs i databasen, vilket i de databasstorlekar det rör sig om (10–200000 poster) görs i så gott som konstant tid. Med anledning av detta presenteras endast ett värde per teknik och testfall.

Testfall 10–21: läsning och uppdatering*Diagram 3: Testfall 10–21*

Testfall 10 och 11 avser läsning av en bestämd författar- respektive bokpost. Vi noterar att de båda BMP-varianterna beter sig som förväntat: ingen skillnad när det gäller författare, men BMPf tar fem gånger så lång tid som BMPc när det gäller böcker. CMP förefaller ta hälften så lång tid som BMPf, vilket är rimligt med tanke på att även CMP är finkornig.

SLSB tar ungefär dubbelt så lång tid som BMP när det gäller författare, och är i princip jämbördig med CMP när det gäller böcker.

Ser vi då på testfall 20 och 21, uppdatering av ett (internt) attribut i en författarpost respektive ett externt attribut i en bokpost, noterar vi att CMP går så fort att det knappt är mätbart. I såväl testfall 20 som testfall 21 är värdet gott och väl under 1 ms. Tiderna för SLSB är kort och så gott som lika i de båda fallen. BMPf ligger med omkring 1 ms i samma liga som de övriga, men BMPc tar synnerligen god tid på sig i testfall 21, hela 9 ms.

Kodexempel för testfall 11 finns från samtliga implementationer i Appendix B.

Testfall 50–61: skapande och borttagande

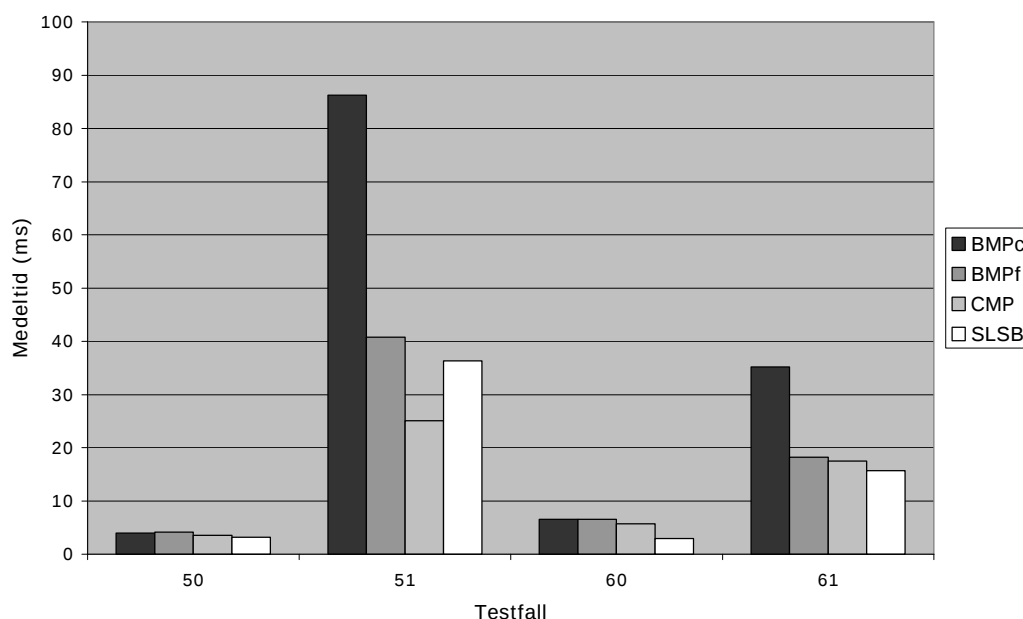


Diagram 4: Testfall 50–61

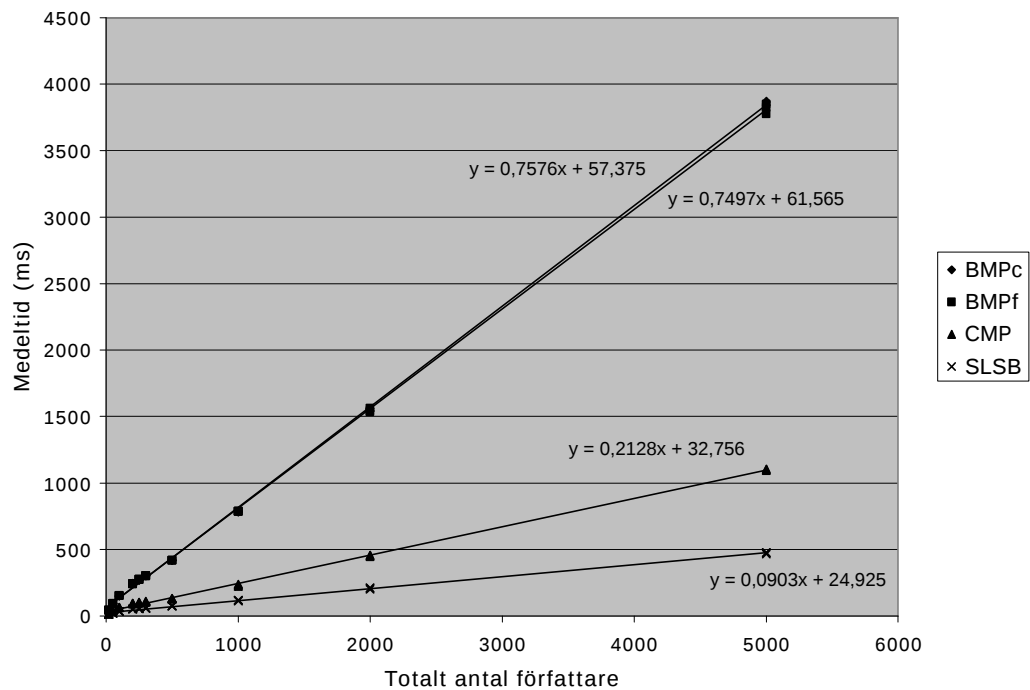
Testfall 51 innebär skapande av en ny bok i systemet, något som för samtliga tekniker visade sig vara så tidskrävande att det avsevärt påverkade skalan i diagrammet. Det är främst BMPc som utmärker sig med sina 86 ms. BMPf tar bara drygt hälften så lång tid, CMP ca en tredjedel och SLSB någonstans däremellan.

Testfall 50 och 60 (skapande respektive borttagande av författare) visar endast små skillnader mellan teknikerna. SLSB visar sig vara snabbast i båda fallen med CMP som näst snabbast.

I testfall 61 (ta bort bok) är skillnaderna större. SLSB är snabbast med sina 15 ms, och därefter följer CMP och BMPf (17 respektive 18 ms). BMPc tar hela 35 ms på sig. Även i detta testfall visar sig alltså BMPf vara snabbare än BMPc.

6.2.2 Testfall inbegripande listningar

Testfall 30 och 31 gick ut på att lista alla författare respektive böcker i testdatabasen. Detta innebär att samtliga data lästes i sin helhet för samtliga poster, vilket innebär att även böckernas externa attribut lästes.

Testfall 30: lista alla författare*Diagram 5: Testfall 30 lista alla författare*

I diagrammet ser vi tydligt att de två BMP-varianterna följs tätt åt. Detta är helt naturligt, eftersom deras kod i det här testfallet är identisk. Vi ser också att BMP är betydligt långsammare än CMP och SLSB, där testfallet tar ungefär dubbelt så lång tid att genomföra med CMP som SLSB.

Kodexempel för testfall 30 finns i Appendix B.

Testfall 31: lista alla böcker

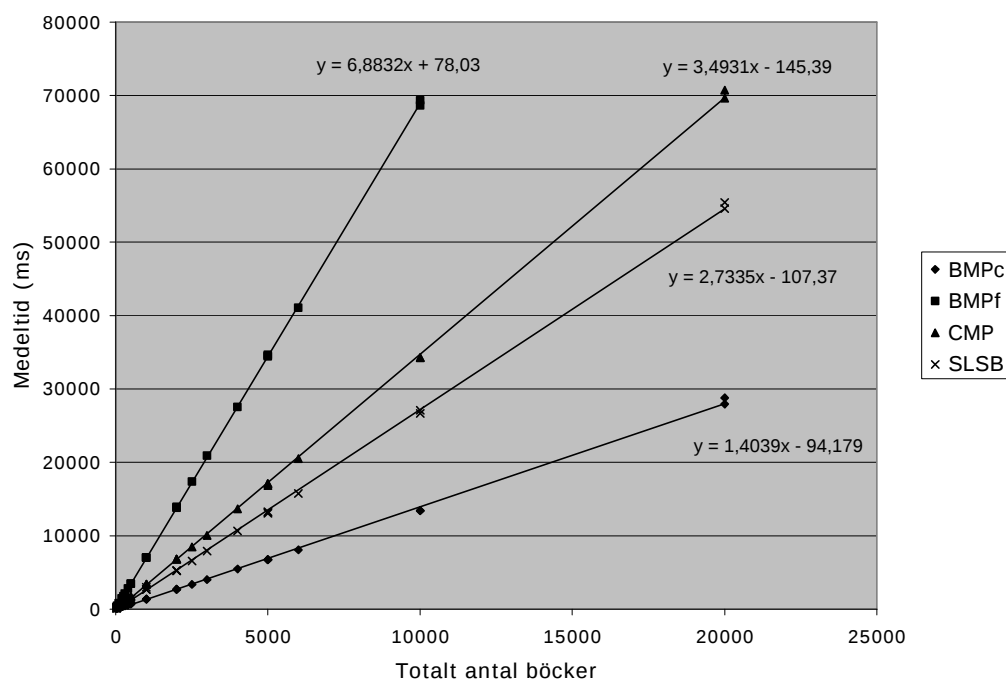


Diagram 6: Testfall 31 lista alla böcker

I Diagram 6 utmärker sig BMPf ganska ordentligt. Det tar nästan fem gånger så lång tid att genomföra testfall 31 med finkorniga som med grovkorniga BMP-entitetsböner. Det är också här de grovkorniga entitetsbönerna i BMPc kommer till sin rätt, och BMPc besegrar i detta testfall både CMP och SLSB.

Att BMPf inte fått något värde vid 20000 böcker beror på att detta skulle krävt att 80000 entitetsböner representerande var sitt attribut skapades, förutom de 20000 böner för själva böckerna. Tillsammans gjorde de här 100000 entitetsböner att WebLogic fick ont om minne och avbröt operationen. CMP kräver lika många böner, men är kanske något mindre minneskrävande. SLSB har naturligtvis inte några sådana problem, men tar ändå längre tid än BMPc.

6.2.3 Testfall inbegripande sökningar

Testfall 40–43 går ut på att göra sökningar bland data i testsystemet. Resultatet av en sökning är en samling primärnycklar som pekar ut de funna posterna. Ingen inläsning av själva datat utförs, mer än vad som krävs då själva sökningen görs. Detta medför att i eEJB-fallen kommer inga entitetsböner att kopplas till de funna EJB-objekten (se avsnitt 3.3.5).

Observera att X-axeln här visar hur många poster sökningen resulterat i och inte databasens storlek, samt att skalorna varierar ganska mycket mellan de olika diagrammen!

Testfall 40: sök författare

Först ut är testfall 40, i vilket författare vars namn innehåller den gemena bokstaven **e** eftersöks.

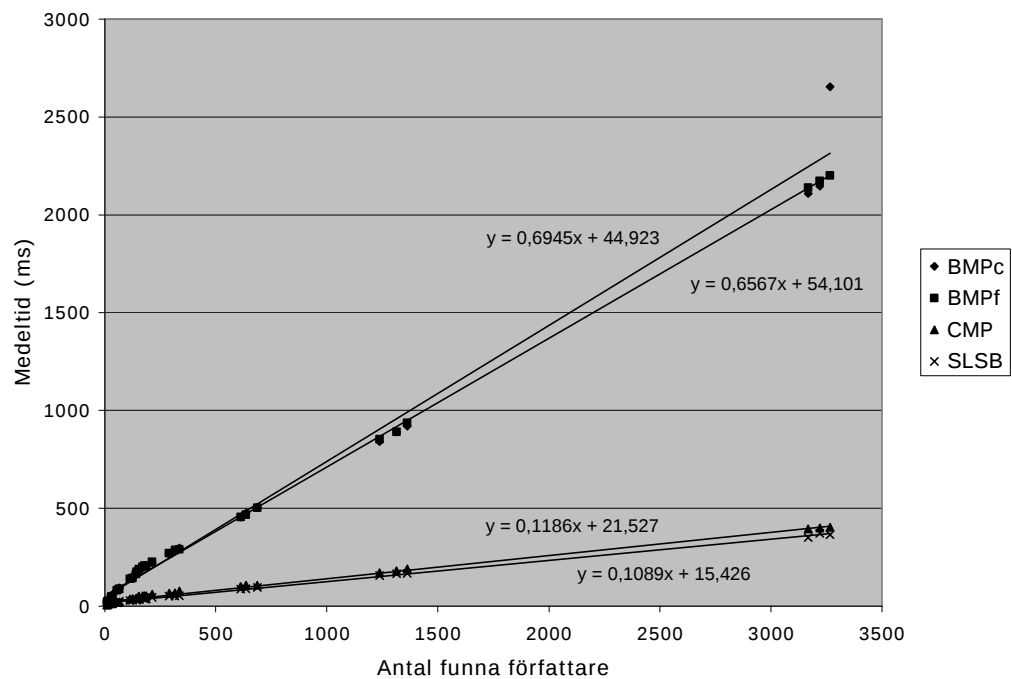


Diagram 7: Testfall 40 sök författare

Liksom i testfall 30 följs de två BMP-varianterna åt, vilket är helt naturligt då deras kod är identisk. Vi noterar att CMP är något snabbare än i testfall 30. SLSB beter sig precis som i testfall 30.

Testfall 41: sök böcker efter titel

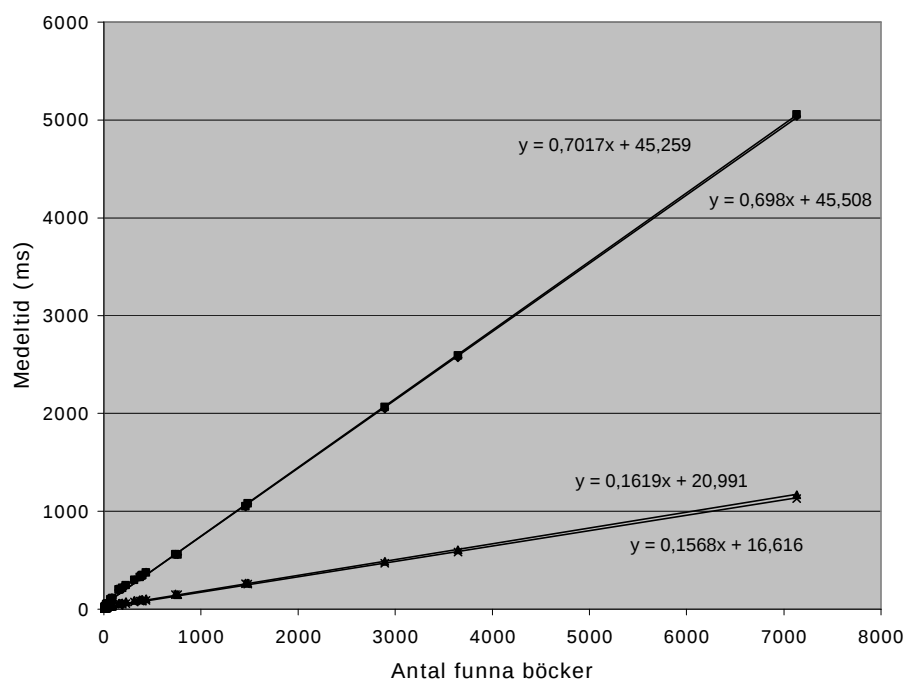
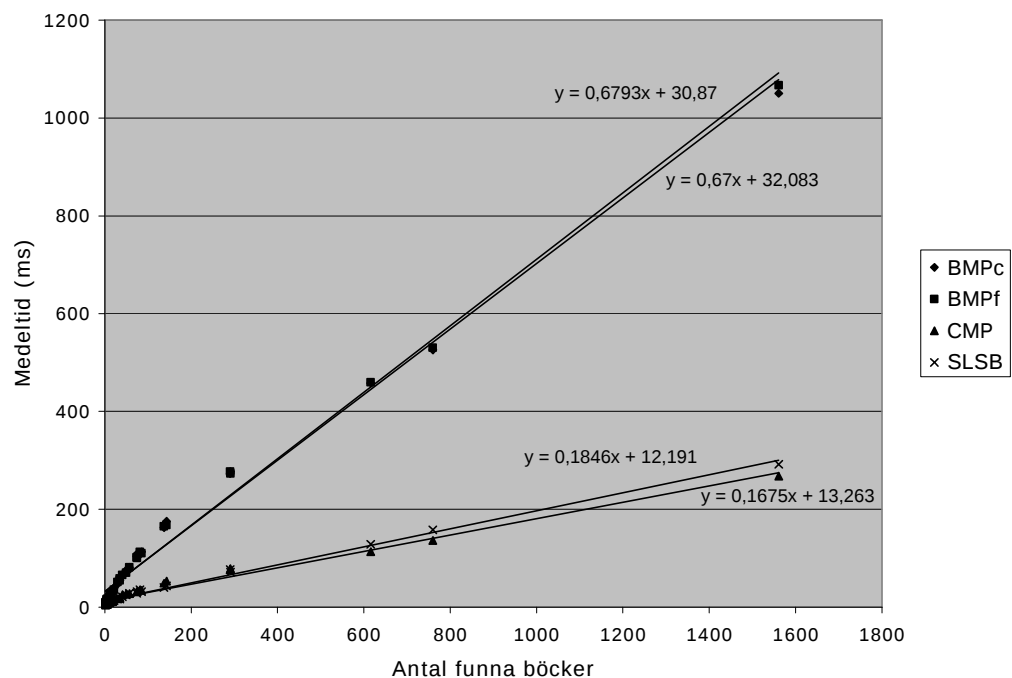


Diagram 8: Testfall 41 sök böcker med matchande titlar

Testfall 41 söker böcker där titeln innehåller strängen ”**Dog**”. I Diagram 8 ser vi att CMP och SLSB är nästan exakt lika snabba, om än aningen långsammare än i testfall 40. De båda BMP-varianterna följs även här mycket tätt åt, vilket beror på att inga böcker väcks ur poolen, varför de externa attributen inte påverkar resultatet alls.

Testfall 42: sök böcker efter författare

Testfall 42 (sökning av böcker skrivna av viss författare) visade sig vara ganska ointressant, då det befanns vara mycket likt testfall 41 och dessutom endast resulterade i en bråkdel av den mängd sökträffar testfall 41 gav. Mot bakgrund av detta redovisas inga resultat från detta testfall.

Testfall 43: sök böcker efter pris*Diagram 9: Testfall 43 sök böcker med matchande pris*

I testfall 43 genomförs en sökning på böckernas externa attribut. Närmare bestämt söks böcker vars pris börjar med siffran **9**. Resultatet liknar i mångt och mycket resultatet i testfall 41, vilket inte är så konstigt med tanke på att sökningarna är mycket lika med avseende på databasen. Skillnaden är att sökningen i detta fall sker i attribut-tabellen istället för i bok-tabellen.

Vi noterar att CMP här förefaller vara något snabbare än SLSB, vilket i sin tur är betydligt snabbare än de båda BMP-varianterna. BMPc och BMPf följer varandra mycket tätt även här, vilket är naturligt då de utför identiska JDBC-anrop.

6.3 Teknikgenomgång

I detta avsnitt görs en genomgång av testresultaten som är ortogonal mot den i avsnitt 6.2 ovan, nämligen med fokus på en teknik i taget.

6.3.1 BMPc

De grovkorniga BMP-entitetsbönorna gör inte speciellt bra ifrån sig i testerna. Det är egentligen bara i två testfall de är bättre än de andra teknikerna, nämligen de två där hela böcker läses ut ur databasen: 11 och

31 (se Diagram 3 och Diagram 6 ovan). Där är de å andra sidan mycket konkurrenskraftiga i fråga om prestanda.

Det finns också två fall där BMPc-lösningen är riktigt usel: testfall 21 och 51 (se Diagram 3 och Diagram 4 ovan).

6.3.2 BMPf

Den finkorniga BMP-varianten har inte heller fått några toppbetyg direkt. Den utmärker sig ensam i testfall 11 och 31, men där genom att ha de absolut sämsta tiderna. Vi noterar att detta även är de fall där BMPc klarat sig bäst! I testfall 21 är förhållandet det omvända, men det beror inte så mycket på att BMPf är snabbt som att BMPc är extremt långsamt i det testfallet. I övriga testfall är de båda BMP-varianterna mycket lika i fråga om prestanda.

6.3.3 CMP

CMP är den av de provade teknikerna som är snabbast i flest testfall. Det finns inget testfall där den klarar sig dåligt, även om den är ganska mycket sämre än BMPc i testfall 11 och 31, något som givetvis har med finkornigheten att göra.

6.3.4 SLSB

Icke-entitetsbönelösningen har generellt fått bra mätvärden. I sökningar är den nästan precis lika snabb som CMP, och i listningar slår den CMP med någon marginal. I testfallen som berör enstaka poster är SLSB i allmänhet jämbördig med den bästa av BMP-varianterna, med undantag för testfall 11, där dess resultat ligger nära CMP:s.

6.4 Sammanställningar

Nedan följer några diagram som är avsedda att ytterligare åskådliggöra förhållanden mellan de olika teknikerna och testfallen.

6.4.1 Enstaka poster

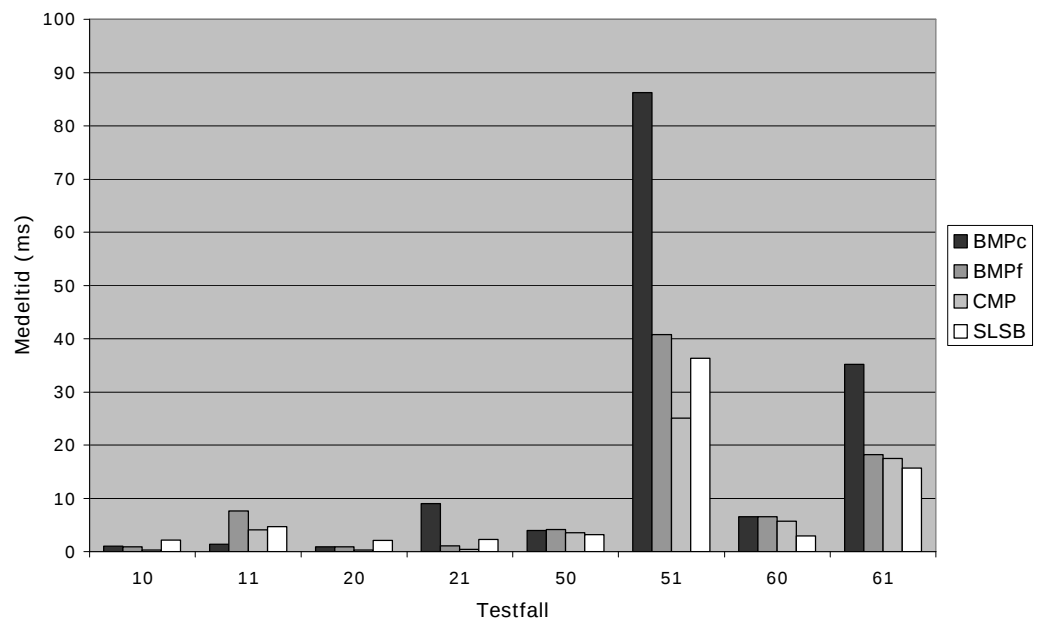


Diagram 10: Sammanställning av testfall 10–21 samt 50–61

I Diagram 10 presenteras samma data som i Diagram 3 och Diagram 4, men här i samma skala. Tiderna från testfall 51 och 61 gör dock att skalan inte riktigt ger rättvisa åt de övriga testfallen, varför en omskalad version av samma diagram återfinns i Diagram 11 här nedan.

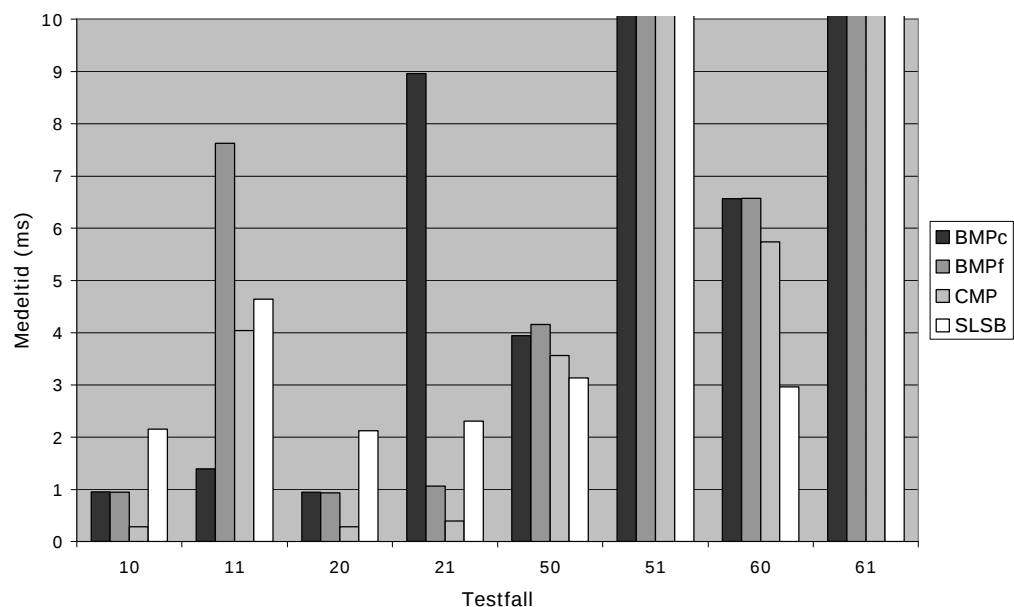


Diagram 11: Sammanställning av testfall 10–21 samt 50–61 i större skala

6.4.2 Listningar och sökningar

Även för listningar och sökningar kan en tid per post beräknas, och då enklast i form av riktningskoefficienten för den minstakvadratanpassade linjen i diagrammen ovan. Detta värde ger en uppfattning om hur lång tid listningar och sökningar tar per behandlad post.

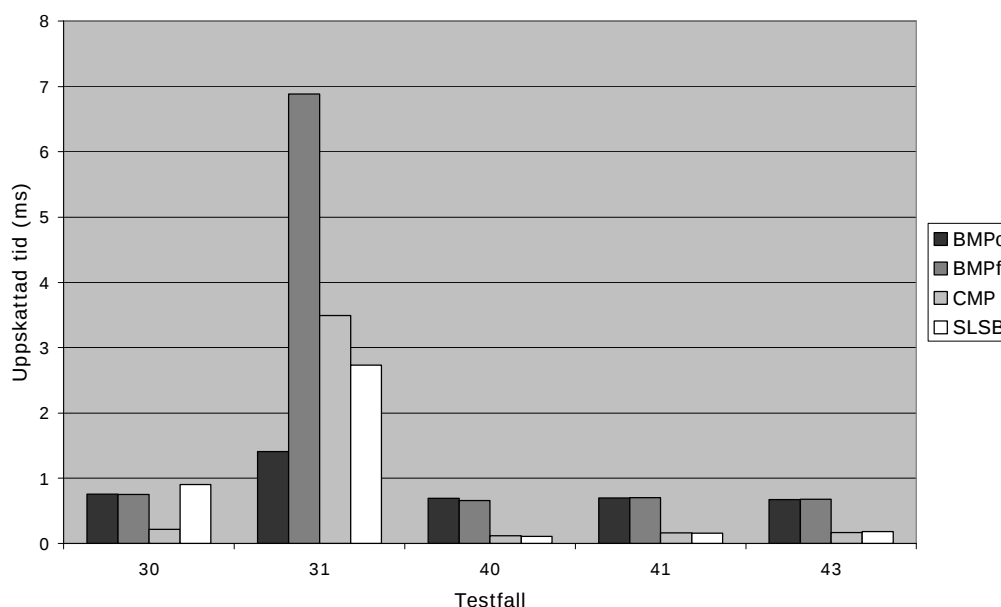


Diagram 12: Sammanställning av testfall 30–43

6.5 Tabeller

I Tabell 10 ges lägsta och högsta värde, aritmetiskt medelvärde samt standardavvikelse för såväl kallt som varmt system. Samtliga tider är angivna i millisekunder.

		kallt system				varmt system			
typ/testfall		Tmin	Tmax	Tavg	SD	Tmin	Tmax	Tavg	SD
BMPc	10	0,94	2,27	1,31	0,56	0,93	1,15	0,95	0,03
	11	1,39	2,72	1,74	0,57	1,36	1,43	1,39	0,01
	20	0,94	1,33	1,04	0,17	0,93	0,98	0,95	0,01
	21	9,08	12,45	9,98	1,41	8,27	10,20	8,96	0,52
	50	3,57	5,73	4,33	0,85	3,40	4,46	3,94	0,27
	51	84,06	90,75	87,69	2,69	83,85	93,66	86,20	1,65
	60	6,29	9,26	7,22	1,21	6,09	7,10	6,57	0,27
	61	34,37	37,67	35,36	1,36	33,39	38,63	35,14	0,90
BMPf	10	0,94	2,56	1,36	0,69	0,93	0,98	0,94	0,01
	11	7,67	10,34	8,32	1,14	7,57	7,82	7,63	0,04
	20	0,93	1,34	1,02	0,18	0,92	0,96	0,93	0,01
	21	1,05	1,92	1,33	0,37	1,05	1,09	1,06	0,01
	50	4,27	7,69	5,24	1,42	3,54	4,87	4,16	0,29
	51	40,33	53,35	43,20	5,68	39,49	47,61	40,80	1,13
	60	6,29	8,67	7,26	0,88	6,20	7,06	6,57	0,24
	61	17,88	23,03	19,48	2,07	16,93	21,42	18,20	0,73
CMP	10	0,27	1,28	0,51	0,44	0,27	0,33	0,28	0,01
	11	4,06	5,98	4,52	0,83	4,02	4,23	4,04	0,02
	20	0,28	0,46	0,32	0,08	0,27	0,32	0,28	0,01
	21	0,38	0,81	0,50	0,18	0,37	0,78	0,39	0,04
	50	3,52	5,92	4,24	1,01	3,01	4,40	3,56	0,29
	51	23,21	26,73	25,02	1,50	23,52	35,33	25,03	1,21
	60	5,36	7,13	6,27	0,65	5,28	6,27	5,74	0,26
	61	17,16	21,54	18,55	1,92	16,57	21,25	17,53	0,81
SLSB	10	2,16	3,91	2,66	0,74	2,13	2,19	2,15	0,01
	11	4,64	6,92	5,13	1,00	4,61	4,96	4,64	0,04
	20	1,96	2,62	2,25	0,29	1,93	2,83	2,12	0,24
	21	2,03	3,28	2,60	0,59	1,95	16,77	2,31	1,52
	50	2,62	6,47	4,12	1,44	2,47	3,79	3,13	0,33
	51	36,01	38,81	37,16	1,22	34,83	47,19	36,30	1,45
	60	2,52	3,69	3,22	0,45	2,32	3,56	2,97	0,31
	61	14,61	17,52	15,89	1,07	14,61	18,59	15,69	0,61

Tabell 10: Sammanställning av resultat för testfall 10–21 och 50–61

Tabell 11 visar den framräknade medelexecveringstiden för testfall 30–43 i millisekunder.

Testfall	BMPc	BMPf	CMP	SLSB
30	0,76	0,75	0,21	0,90
31	1,40	6,88	3,49	2,73
40	0,69	0,66	0,12	0,11
41	0,70	0,70	0,16	0,16
43	0,67	0,68	0,17	0,18

Tabell 11: Sammanställning av resultat för testfall 30–43

7 Diskussion

I det här kapitlet ges författarens subjektiva kommentarer och spekulationer, och resultaten jämförs med dem från tidigare undersökningar.

7.1 Observationer och kommentarer

I testfall 21 visar BMPc mycket dåliga resultat. Troligtvis beror detta på BMPc-bok-entitetsbönans interna representation av de externa attributen.

I testfall 40 visar sig CMP vara något snabbare än i testfall 30, vilket troligtvis beror på att bönorna inte behöver väckas ur poolen då endast homeinterfacets metoder används (se avsnitt 3.3.5).

Testfall 51 var mycket tidskrävande för samtliga testade tekniker. Troligtvis beror den stora tidsåtgången delvis på att två databastabeller är inblandade, men även på mer eller mindre ineffektiva implementationer, framför allt i BMP-fallen.

Generellt kan sägas att de operationer som innebär förändringar i databasen är de som tar längst tid att genomföra. Testfall 51 och 61, vilka innebar skapande respektive borttagande av en bok i systemet, var de testfall som tog överlägset längst tid.

7.2 Vad säger andra?

Mikael Eriksson kommer i sin rapport (Eriksson 2000) fram till att BMP generellt sett är snabbare än CMP, något som direkt motsäger resultaten i denna rapport. Han har dock använt en tidigare applikationsserver (BEA WebLogic Server 4.0.3), vilket med stor sannolikhet inverkat menligt på CMP-prestanda i hans mätningar.

Kent Sundbergs resultat (Sundberg 2001) pekar i samma riktning som de vi fått fram här. CMP är svårslaget när det gäller snabbhet, och BMP-böner bör användas endast då det är nödvändigt. Sundberg har använt applikationsservern WebSphere 3.5 från IBM i kombination med IBM:s databas DB2 71.

8 Slutsatser

I det här examensarbetet har jag undersökt vilka möjligheter som finns för datalagring i J2EE-miljö, valt ut några tekniker för testning och gjort provimplementationer av dessa. Jag har sedan gjort prestandamätningar på provimplementationerna och sammanställt resultaten.

Baserat på gjorda mätningar har jag dragit tre slutsatser, vilka redovisas nedan.

CMP är bättre än BMP

I de fall *Container-Managed Persistence* (CMP) kunnat användas med hänsyn till teknikens begränsningar i aktuell EJB-version är den överlägsen *Bean-Managed Persistence* i fråga om både prestanda och enkelhet vid implementation. Nyare EJB-versioner utlovar större användningsområden och kan knappast förmodas bli mindre effektiva än nuvarande.

Databasoperationer är dyra

Databasoperationer är dyra, och bör om möjligt undvikas. Naturligtvis kan inte alla databasoperationer undvikas, men det är en god idé att hålla antalet anrop till databasen så lågt som möjligt. Det är i många fall bättre att läsa ut mer data än man egentligen behöver i ett enda anrop än att i flera anrop läsa ut just de data man vill åt.

Välj kornighetsgrad beroende på användningsområde

Att använda grovkorniga entitetsbönor kan ibland vara lämpligt ur prestandasynpunkt. Mest kommer de till sin rätt då samtliga data ska läsas ut, till exempel vid listningar, vilket troligtvis hänger samman med antalet databasanrop. Dock är det mindre ofta värt den ökade komplexitet vid design och implementation som grovkorniga entitetsbönor medför.

8.1 Vidare arbete

Under arbetets gång dök då och då nya idéer till vad man skulle kunna undersöka upp, och vissa saker som från början var tänkta att undersökas föll bort på grund av tidsbrist. I det här avsnittet tas några av dessa saker upp.

8.1.1 Flera samtidiga klienter

Att testa hur de olika teknikerna klarar av den ökade belastningen då flera klienter exekverar samtidigt i samma eller olika testfall vore förmodligen ganska intressant.

8.1.2 Verklighetsnära testsekvenser

De i detta arbete gjorda testerna bygger på att samma testfall körts om och om igen i sekvens. Att bygga testsviter som innefattar sekvenser av olika testfall i en ordning och omfattning som efterliknar realistiskt användande av systemet skulle kunna vara intressant, då det skulle kunna påvisa effekter av cacheanvändning och poolutnyttjande.

8.1.3 Minnesanvändning

I detta arbete har en i många fall viktig aspekt av de olika teknikerna nästan helt utelämnats, nämligen minnesanvändning. I en tänkbar fortsättning på arbetet vore det mycket intressant att undersöka minnesåtgången för de olika teknikerna.

8.1.4 EJB 2.0

Under slutfasen av detta arbete kom version 2.0 av EJB-standarden ut (Sun Microsystems 2001). EJB 2.0 erbjuder många intressanta nyheter såsom mer avancerade CMP-böror, ett nytt frågespråk för *finder*-metoder och andra finesser. Det vore mycket intressant att se hur den nya standarden inverkar på möjligheterna att använda CMP i större utsträckning.

8.1.5 JDO

Java Data Objects är en mycket intressant teknik som förtjänar att undersökas närmare. Se avsnitt 4.4.

9 Referenser

ANSI (1998), *ANSI X3.135.10-1998*

BEA Systems (2001), *WebLogic Server EJB Reference*

http://www.weblogic.com/docs51/classdocs/API_ejb/EJB_reference.html (kontrollerad 2001-09-28)

Codd, Edgar F (1970), *A Relational Model of Data for Large Shared Data Banks*, Communications of the ACM, Vol. 13, No. 6, June 1970

<http://www.cmpe.boun.edu.tr/~rxk00500/codd/toc.html> (kontrollerad 2001-06-08)

Compoze Software, Inc. (2001), *Automatic Bean-Managed Persistence*, TheServerSide.com

<http://www.theserverside.com/resources/abmp/ABMP.html>

(kontrollerad 2001-04-27). Se även <http://www.compoze.com>.

Cotner & Rodas (1999), *Five Reasons for Using SQLJ in Your Java Programs*, DB2 magazine, spring 1999

http://www.db2mag.com/db_area/archives/1999/q1/99sp_webdb.shtml (kontrollerad 2001-05-21)

Data Access Object (2001),

http://java.sun.com/j2ee/blueprints/design_patterns/data_access_object/ (kontrollerad 2001-09-04)

Eriksson, Mikael (2000), *Database persistence with Enterprise JavaBeans*, examensarbete 2000:52 vid Institutionen för Systemteknik, Luleå tekniska universitet, ISSN 1402-1617, ISRN LTU-EX--00/52--SE

<http://epubl.luth.se/1402-1617/2000/052/> (kontrollerad 2001-06-20)

Holland, Grant (2001), *Entity bean relationships in EJB 1.1*, Java Report, april, maj 2001 (Vol. 6, No. 4-5)

Ida Infront AB: *iipax Java framework*, 2001

Java Data Objects (2001), <http://access1.sun.com/jdo/> (kontrollerad 2001-05-23)

Malani, Prakash (2001), *Connection Strategies in EntityBeans*, Java Report, april 2001

http://www.javareport.com/html/from_pages/article.asp?id=799&mon=4&yr=2001 (kontrollerad 2001-05-28)

Marinescu, Floyd (2000), *Coarse Grained Entity Beans vs. Fine Grained Entity Beans*, TheServerSide.com 21 augusti 2000

http://www.theserverside.com/discussion/thread.jsp?thread_id=706

(kontrollerad 2001-09-06)

Session Facade (2001),

http://java.sun.com/j2ee/blueprints/design_patterns/session_facade/

(kontrollerad 2001-09-04)

Stern, Shirley Ann (1999), *SQLJ: The 'open sesame' of Java database applications*, Java World, May 1999

http://www.javaworld.com/javaworld/jw-05-1999/jw-05-sqlj_p.html

(kontrollerad 2001-05-22)

Sun Microsystems, Inc. (1997), *Enterprise JavaBeans Specification, v1.0*,

<http://java.sun.com/products/ejb/docs.html> (kontrollerad 2001-09-24)

Sun Microsystems, Inc. (1999), *Enterprise JavaBeans Specification, v1.1*,

<http://java.sun.com/products/ejb/docs.html> (kontrollerad 2001-09-24)

Sun Microsystems, Inc. (2001), *Enterprise JavaBeans Specification, v2.0*,

<http://java.sun.com/products/ejb/docs.html> (kontrollerad 2001-09-24)

Sucharitakul, Akara (2001), *Seven Rules for Optimizing Entity Beans*, Java Developer Connection, maj 2001

<http://developer.java.sun.com/developer/technicalArticles/ebeans/sevenrules/> (kontrollerad 2001-09-05)

Sundberg, Kent (2001), *Optimeringar för Enterprise JavaBeans arkitekturer*, examensarbete vid Institutionen för datavetenskap, Umeå universitet

<http://www.cs.umu.se/~c97ksg/> (kontrollerad 2001-06-20)

Value Object (2001),

http://java.sun.com/j2ee/blueprints/design_patterns/value_object/

(kontrollerad 2001-09-04)

9.1 Litteratur

Monson-Haefel, Richard (2000), *Enterprise JavaBeans, 2nd edition*, O'Reilly & Associates Inc. 2000, ISBN 1-56592-869-5

Roman, Ed (1999), *Mastering Enterprise JavaBeans and the Java2 Platform, Enterprise Edition*, John Wiley & Sons, Inc. 1999, ISBN 0-471-33229-1

9.2 Övriga informationskällor

iipax	http://www.iipax.com (kontrollerad 2001-09-24)
jGuru	http://www.jguru.com (kontrollerad 2001-05-22)
The Server Side	http://www.theserverside.com (kontrollerad 2001-06-08)

Appendix A Begrepp och förkortningar

Begrepp

Affärslogik	<i>Business logic</i> , rutiner för affärsverksamhetsrelaterade aktiviteter, till exempel påförande av ränta i ett bankkonto eller beräkning av rabatt vid försäljning av varor.
Applikationsserver	En applikationsserver betjänar klienter (vanligen web- eller Java-baserade) och utgör ett lager mellan dessa och servertjänster som till exempel databaser. Detta mellanlager kan hantera sessioner och autentisering, men även innehålla stora mängder affärslogik.
Beständighet	<i>Persistence</i> , egenskap hos data i systemet.
Connection pooling	Att hålla ett antal databaskopplingar redo för användande så att den som behöver kommunicera med databasen slipper göra uppkopplingen själv.
Databaskoppel	Förbindelse med databas. Se Koppel.
Deployment descriptor	Se Installationsbeskrivning.
EJB-container	EJB-behållare, styr börnornas livscykel, hanterar säkerhet och trådning samt tillhandahåller diverse tjänster såsom nätverk, beständighet och transaktioner.
Enterpriseböna	EJB-instans.
Entitetsböna	<i>Entity bean</i> , ett beständigt objekt som representerar data i en databas.
Fasadböna	sEJB som tjänstgör som mellanhand mellan en klient och andra EJB och därmed döljer deras gränssnitt.
Frågeplan	<i>Query plan</i> , databasens (optimerade) plan för hur en viss SQL-fråga skall utföras.

Främmande nyckel	Attribut hos en databaspost som utgörs av en primärnyckel från en annan tabell. Används för att ange referenser till andra tabeller.
Garbage collector	System för att ta bort objekt som inte längre används ur minnet.
Installationsbeskrivning	<i>Deployment descriptor</i> , en datamängd som beskriver hur en EJB skall installeras i containern. Här anges till exempel behörighetsinformation och för eEJB med CMP vilka attribut som skall vara beständiga.
Koppel	<i>Connection</i> , (nätverks-)förbindelse, resultatet av en lyckad uppkoppling.
Lagrad procedur	<i>Stored procedure</i> , ett slags kompilerat makro som lagrats i databashanteraren och då det anropas utför en rad SQL-kommandon i databasen. Till exempel kan det användas för att utföra alla operationer som behövs då en användare läggs till i ett system.
Passivera	En EJB-container kan passivera en enterpriseböna, d v s spara undan dess tillstånd och ta bort den ur minnet.
Primärnyckel	Attribut hos en databaspost som identifierar den och endast den posten. Oftast ett heltal.
Serialiserbar	<i>Serializable</i> , egenskap hos objekt som säger att objektet kan omvandlas till en oktettföljd för enkel överföring eller lagring och sedan omvandlas till ett objekt igen.
Stored procedure	Se Lagrad procedur.
TP-monitor	<i>Transaction Processing Monitor</i> , ett slags operativsystem för affärssystem, som tillhandahåller tjänster för fjärranrop, transaktioner och meddelandehantering.
Value object	Se Värdeobjekt.
Värdeobjekt	<i>Value object</i> , ett serialiserbart Java-objekt som används för att "klumpa ihop" flera data till ett

objekt för att minska resursåtgången framför allt vid anrop av avlägsna objekts metoder.

Förkortningar

ACL	<i>Access Control List</i>
AOS	<i>Advanced Object Storage</i> , objektservern som utgör en av grundstenarna i iipax.
API	<i>Application Programming Interface</i> , gränssnitt som gör det möjligt för programmerare att i sin kod utnyttja tjänster som finns tillgängliga i annat program eller funktionssamling.
BMP	<i>Bean-Managed Persistence</i> , innebär att en eEJB hanterar databeständighet själv.
BMT	<i>Bean-Managed Transactions</i> , innebär att en EJB hanterar transaktioner själv.
CMP	<i>Container-Managed Persistence</i> , innebär att EJB-containern hanterar databeständigheten åt eEJB.
CMT	<i>Container-Managed Transactions</i> , innebär att EJB-containern sköter transaktioner åt EJB.
CTM	<i>Component Transaction Monitor</i> , en kombination av en ORB och en TP-monitor.
DAO	<i>Data Access Object</i>
DMS	<i>Document Management System</i>
DNS	<i>Domain Name System</i> , katalogtjänst som kopplar ihop domän- och datornamn med IP-nummer i nätverk.
DBMS	<i>DataBase Management System</i> , databashanterare. DBMS är den programvara som sköter sökningar och andra operationer på en databas.
DDL	<i>Data Definition Language</i> , språk för att konstruera och beskriva en databas.

DML	<i>Data Manipulation Language</i> , språk för att göra sökningar och uppdatera information i en databas.
eEJB	<i>Entity Enterprise JavaBean</i> , entitetsböna.
EJB	<i>Enterprise JavaBeans</i>
J2EE	<i>Java 2 Enterprise Edition</i>
JDBC	<i>Java DataBase Connectivity</i> , API för databasåtkomst i Java.
JDO	<i>Java Data Objects</i>
JNDI	<i>Java Naming and Directory Interface</i>
JTA	<i>Java Transaction API</i>
JTS	<i>Java Transaction Service</i>
JVM	<i>Java Virtual Machine</i> , virtuell Java-maskin, programkörningsmiljö i vilken Java-program kan exekveras.
ODBC	<i>Open DataBase Connectivity</i> , ett standardiserat gränssnitt för databasåtkomst
ORB	<i>Object Request Broker</i> , kommunikationscentral i distribuerat system som sköter metoder och liknande mellan objekt.
RMI	<i>Remote Method Invocation</i> , fjärranrop av metoder.
RPC	<i>Remote Procedure Call</i> , fjärranrop av procedurer (ej objektorienterat).
sEJB	<i>Session Enterprise JavaBean</i> , sessionsböna.
SLSB	<i>StateLess Session Bean</i> , tillståndslös sessionsböna.
SQL	<i>Structured Query Language</i> , frågespråk för relationsdatabaser.
SQLJ	En specifikation för användande av SQL i Java med hjälp av ett förbehandlingsspråk.

VO	Värdeobjekt (<i>value object</i>).
WWW	<i>World-Wide Web</i>
XML	<i>Extensible Markup Language</i> , ett uppmärkningsspråk med dynamisk syntax.

Appendix B Kodexempel

Testfall 11: läsa bok inklusive externa attribut

BMPcServer

getBookById()

```
public BookVO getBookById(int id)
{
    // Set up the VO
    BookVO book = new BookVO();
    book.id = id;
    long start = System.currentTimeMillis(); // start the clock
    book.start(); // start the clock
    try {
        for (int doItSeveralTimes = 0; doItSeveralTimes<severalTimes; doItSeveralTimes++) {
            // Get the Book bean
            BookBMPc bean = null;
            bean = (BookBMPc) bookHome.findByPrimaryKey(new BookBMPcPK(id));
            // Get the data
            book.title = bean.getTitle();
            // Get the attributes
            book.attributes = new Vector();
            Enumeration attributeKeys = bean.getAttributeKeys();
            while (attributeKeys.hasMoreElements()) {
                AttributeVO tempAttribute = new AttributeVO();
                tempAttribute.ok = true;
                tempAttribute.bookId = book.id;
                tempAttribute.key = (String) attributeKeys.nextElement();
                tempAttribute.value = bean.getAttribute(tempAttribute.key);
                // stop the clock
                tempAttribute.time = System.currentTimeMillis() - start;
                book.attributes.addElement(tempAttribute);
            }
        }
        book.stop(); // stop the clock
        book.ok = true;
        return book;
    }
    catch (Exception e) {
        e.printStackTrace();
        book.stop(); // stop the clock
        return book;
    }
}
```

BookBMPc

För att hålla de externa attributen har BookBMPcEJB följande attribut:

```
private Vector attributeValues;
private Vector attributeKeys;
private Vector attributeIds;
```

ejbFindByPrimaryKey()

```
/**
 * The container invokes this method on the bean as a result of a
 * client-invoked find. This method is required for an entity bean.
 * @return Returns the primary key
 * @param key Primary Key to load the bean
 * @exception javax.ejb.FinderException
 */
public BookBMPcPK ejbFindByPrimaryKey(BookBMPcPK key)
    throws FinderException {
```

```
    try {
        readData(key);
        return key;
    }
    catch (EJBException ee) {
        throw new FinderException(ee.getMessage());
    }
}
```

ejbLoad()

```
/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database.
 */
public void ejbLoad() {
    // Get the primary key
    BookBMPCPK key = (BookBMPCPK)context.getPrimaryKey();

    // Now load from the database
    try {
        readData(key);
    }
    catch(Exception ex) {
        throw new RuntimeException(ex.getMessage());
    }
}
```

readData()

```
/**
 * The method reads the data from database fields into this
 * bean.
 * @param key Primary Key used to load the bean
 * @exception javax.ejb.FinderException
 * @exception javax.ejb.EJBException
 */
private void readData(BookBMPCPK key)
    throws FinderException, EJBException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = getConnection();

        // Load book from the database
        ps = con.prepareStatement("select TITLE,AUTHOR from BJORN.BOOKS where ID = ?");
        ps.setInt(1, key.id);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            ex = new FinderException("Record not found");
        else {
            id = key.id;
            title = rs.getString(1);
            authorId = rs.getInt(2);
            unsetModified();
        }
        rs.close();
        ps.close();
        // Load attributes from the database
        ps = con.prepareStatement("select ID,KEY,VALUE from BJORN.ATTRIBUTES where BOOK = ?");
        ps.setInt(1, key.id);
        rs = ps.executeQuery();
        if (!rs.next())
            ex = new FinderException("No records found");
        else {
            attributeKeys = new Vector();
            attributeValues = new Vector();
            attributeIds = new Vector();
            attributeIds.addElement(new Integer(rs.getInt(1)));
            attributeKeys.addElement(rs.getString(2));
            attributeValues.addElement(rs.getString(3));
            while (rs.next()) {
```

```

        attributeIds.addElement(new Integer(rs.getInt(1)));
        attributeKeys.addElement(rs.getString(2));
        attributeValues.addElement(rs.getString(3));
    }
    unsetModified();
}
rs.close();
}
catch (EJBException ee) {
    ex = ee;
}
catch (SQLException se) {
    ex = new EJBException(se.getMessage());
}

// Clean-up resources
try {
    ps.close();
    con.close();
}
catch (Exception e) {}

// Throw exception if error occurred
if (ex != null) {
    if (ex instanceof EJBException)
        throw (EJBException)ex;
    if (ex instanceof FinderException)
        throw (FinderException)ex;
}
}
}

```

getAttribute()

```

/**
 * @return
 * @param keyName
 */
public String getAttribute(String keyName) {
    if (attributeKeys == null) {
        return "";
    }
    else {
        int id = attributeKeys.indexOf(keyName);
        if (id < 0) { // not found
            return "";
        }
        else { // found
            return (String) attributeValues.get(id);
        }
    }
}
}

```

getConnection()

```

/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    // to do: modify datasource below to match your server
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
}

```

BMPfServer

getBookById()

```
public BookVO getBookById(int id)
{
    // Set up the VO
    BookVO book = new BookVO();
    book.id = id;
    book.start(); // start the clock
    try {
        for (int doItSeveralTimes = 0; doItSeveralTimes<severalTimes;
            doItSeveralTimes++) {
            // Get the Book bean
            BookBMPf bean = null;
            bean = (BookBMPf) bookHome.findByPrimaryKey(new BookBMPfPK(id));
            // Get the data
            book.title = bean.getTitle();
            // Get the attributes
            AttributeVO[] attributes = getAttributesByBookId(id);
            book.attributes = new Vector();
            for (int i=0; i<attributes.length; i++) {
                book.attributes.addElement(attributes[i]);
            }
        }
        book.stop(); // stop the clock
        book.ok = true;
        return book;
    }
    catch (Exception e) {
        e.printStackTrace();
        book.stop(); // stop the clock
        return book;
    }
}
```

getAttributesByBookId()

```
public AttributeVO[] getAttributesByBookId(int bookId)
{
    long start = System.currentTimeMillis(); // start the clock
    try {
        // Get the attribute beans
        Enumeration beans = null;
        beans = (Enumeration) attributeHome.findByBookId(bookId);

        // Get the data
        Vector attributeVector = new Vector();
        while(beans.hasMoreElements()) {
            AttributeBMPf bean = (AttributeBMPf) beans.nextElement();
            AttributeVO attribute = new AttributeVO();
            attribute.id = bean.getId();
            attribute.key = bean.getKeyName();
            attribute.value = bean.getValue();
            attribute.bookId = bean.getBookId();
            attribute.time = System.currentTimeMillis() - start; // stop the clock
            attribute.ok = true;
            attributeVector.addElement(attribute);
        }
        AttributeVO[] ret = new AttributeVO[attributeVector.size()];
        attributeVector.copyInto(ret);
        return ret;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

BookBMPf

ejbFindByPrimaryKey()

```

/**
 * The container invokes this method on the bean as a result of a
 * client-invoked find. This method is required for an entity bean.
 * @return Returns the primary key
 * @param key Primary Key to load the bean
 * @exception javax.ejb.FinderException
 */
public BookBMPfPK ejbFindByPrimaryKey(BookBMPfPK key)
    throws FinderException {
    try {
        readData(key);
        return key;
    }
    catch (EJBException ee) {
        throw new FinderException(ee.getMessage());
    }
}

```

readData()

```

/**
 * The method reads the data from database fields into this
 * bean.
 * @param key Primary Key used to load the bean
 * @exception javax.ejb.FinderException
 * @exception javax.ejb.EJBException
 */
private void readData(BookBMPfPK key)
    throws FinderException, EJBException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = getConnection();
        // Load from the data base
        ps = con.prepareStatement("select TITLE,AUTHOR from BJORN.BOOKS where ID = ?");
        ps.setInt(1, key.id);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            ex = new FinderException("Record not found");
        else {
            id = key.id;
            title = rs.getString(1);
            authorId = rs.getInt(2);
            unsetModified();
        }
        rs.close();
    }
    catch (EJBException ee) {
        ex = ee;
    }
    catch (SQLException se) {
        ex = new EJBException(se.getMessage());
    }
    // Clean-up resources
    try {
        ps.close();
        con.close();
    }
    catch (Exception e) {}
    // Throw exception if error occurred
    if (ex != null) {
        if (ex instanceof EJBException)
            throw (EJBException)ex;
        if (ex instanceof FinderException)
            throw (FinderException)ex;
    }
}

```

getConnection()

```
/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    // to do: modify datasource below to match your server
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
```

AttributeBMPf

ejbFindByBookId()

```
public Enumeration ejbFindByBookId(int bookId)
    throws FinderException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    Vector ret = null;
    try {
        con = getConnection();
        ps = con.prepareStatement("select ID from BJORN.ATTRIBUTES where BOOK = ?");
        ps.setInt(1, bookId);
        ResultSet rs = ps.executeQuery();
        if (!rs.next()) {
            ex = new FinderException("No records found");
        } else {
            ret = new Vector();
            ret.addElement(new AttributeBMPfPK(rs.getInt(1)));
            while(rs.next()) {
                ret.addElement(new AttributeBMPfPK(rs.getInt(1)));
            }
        }
        rs.close();
    }
    catch (EJBException ee) {
        ex = ee;
    }
    catch (SQLException se) {
        ex = new EJBException(se.getMessage());
    }

    // Clean-up resources
    try {
        ps.close();
        con.close();
    }
    catch (Exception e) {}

    // Throw exception if error occurred
    if (ex != null) {
        if (ex instanceof EJBException)
            throw (EJBException)ex;
        if (ex instanceof FinderException)
            throw (FinderException)ex;
    }

    return (ret.size()>0) ? ret.elements() : null;
}
```

ejbLoad()

```

/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database.
 */
public void ejbLoad() {
    // Get the primary key
    AttributeBMPfPK key = (AttributeBMPfPK)context.getPrimaryKey();

    // Now load from the database
    try {
        readData(key);
    }
    catch(Exception ex) {
        throw new RuntimeException(ex.getMessage());
    }
}

```

readData()

```

/**
 * The method reads the data from database fields into this
 * bean.
 * @param key Primary Key used to load the bean
 * @exception javax.ejb.FinderException
 * @exception javax.ejb.EJBException
 */
private void readData(AttributeBMPfPK key)
    throws FinderException, EJBException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    try {
        con = getConnection();

        // Load from the data base
        ps =
            con.prepareStatement("select KEY,VALUE,BOOK from BJORN.ATTRIBUTES where ID = ?");
        ps.setInt(1, key.id);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            ex = new FinderException("Record not found");
        else {
            id = key.id;
            keyName = rs.getString(1);
            value = rs.getString(2);
            bookId = rs.getInt(3);
            unsetModified();
        }
        rs.close();
    }
    catch (EJBException ee) {
        ex = ee;
    }
    catch (SQLException se) {
        ex = new EJBException(se.getMessage());
    }

    // Clean-up resources
    try {
        ps.close();
        con.close();
    }
    catch (Exception e) {}

    // Throw exception if error occurred
    if (ex != null) {
        if (ex instanceof EJBException)
            throw (EJBException)ex;
        if (ex instanceof FinderException)
            throw (FinderException)ex;
    }
}

```

getConnection()

```
/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    // to do: modify datasource below to match your server
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
```

CMPServer

getBookById()

```
public BookVO getBookById(int id)
{
    // Set up the VO
    BookVO book = new BookVO();
    book.id = id;
    book.start(); // start the clock
    BookCMPPK pk = new BookCMPPK();
    pk.id = id;
    try {
        for (int doItSeveralTimes = 0; doItSeveralTimes<severalTimes; doItSeveralTimes++) {
            // Get the Book bean
            BookCMP bean = null;
            bean = (BookCMP) bookHome.findByPrimaryKey(pk);

            // Get the data
            book.title = bean.getTitle();
            // Get the attributes
            AttributeVO[] attributes = getAttributesByBookId(id);
            book.attributes = new Vector();
            for (int i=0; i<attributes.length; i++) {
                book.attributes.addElement(attributes[i]);
            }
            book.stop(); // stop the clock
            book.ok = true;
            return book;
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        book.stop(); // stop the clock
        return book;
    }
}
```

getAttributesByBookId()

```
public AttributeVO[] getAttributesByBookId(int bookId)
{
    long start = System.currentTimeMillis(); // start the clock
    try {
        // Get the attribute beans
        Enumeration beans = null;
        beans = (Enumeration) attributeHome.findByBookId(bookId);
        // Get the data
        Vector attributeVector = new Vector();
        while(beans.hasMoreElements()) {
            AttributeCMP bean = (AttributeCMP) beans.nextElement();
            AttributeVO attribute = new AttributeVO();
        }
    }
}
```

```

        attribute.id = bean.getId();
        attribute.key = bean.getKey();
        attribute.value = bean.getValue();
        attribute.bookId = bean.getBook();
        attribute.time = System.currentTimeMillis() - start; // stop the clock
        attribute.ok = true;
        attributeVector.addElement(attribute);
    }
    AttributeVO[] ret = new AttributeVO[attributeVector.size()];
    attributeVector.copyInto(ret);
    return ret;
}
catch (Exception e) {
    e.printStackTrace();
    return null;
}
}

```

BookCMP

ejbLoad()

```

/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database. This method is called after the container
 * has loaded the bean's state from the database.
 */
public void ejbLoad() {
    unsetModified();
}

```

AttributeCMP

ejbLoad()

```

/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database. This method is called after the container
 * has loaded the bean's state from the database.
 */
public void ejbLoad() {
    unsetModified();
}

```

SLSBServer

getBookById()

```

/**
 * @param id id of book to get
 * @return book VO
 */
public BookVO getBookById(int id)
{
    long start = System.currentTimeMillis(); // start the clock
    BookVO book = new BookVO();
    book.ok = false;
    book.id = id;
    book.start(); // start the clock
    try {
        for (int doItSeveralTimes = 0; doItSeveralTimes < severalTimes; doItSeveralTimes++) {
            Connection con = getConnection();
            PreparedStatement ps = con.prepareStatement("SELECT title, author FROM books
WHERE id = ?");
            ps.setInt(1, id);
            ResultSet rs = ps.executeQuery();

```

```

        if (rs.next()) {
            book.title = rs.getString(1);
            book.authorId = rs.getInt(2);
            // attributes
            AttributeVO[] attributeArray = getAttributesByBookId(book.id);
            if (attributeArray != null) {
                book.attributes = new Vector();
                for(int i=0; i<attributeArray.length; i++) {
                    AttributeVO tempAttribute = new AttributeVO();
                    tempAttribute.bookId = book.id;
                    tempAttribute.id = attributeArray[i].id;
                    tempAttribute.key = attributeArray[i].key;
                    tempAttribute.value = attributeArray[i].value;
                    tempAttribute.ok = true;
                    // stop the clock
                    tempAttribute.time = System.currentTimeMillis() - start;
                    book.attributes.addElement(tempAttribute);
                }
            }
            book.ok = true;
        }
        ps.close();
        con.close();
    }
}
catch (Exception e) {
    e.printStackTrace();
}
book.time = System.currentTimeMillis() - start; // stop the clock
return book;
}

```

getAttributesByBookId()

```

/**
 * @param bookId
 * @param key
 * @return AttributeVO
 */
public AttributeVO[] getAttributesByBookId(int bookId)
{
    long start = System.currentTimeMillis(); // start the clock

    try {
        Connection con = getConnection();
        PreparedStatement ps = con.prepareStatement("SELECT id, key, value FROM attributes
        WHERE book = ?");
        ps.setInt(1, bookId);

        ResultSet rs = ps.executeQuery();
        Vector attributeVector = new Vector();
        while (rs.next()) {
            AttributeVO tempAttribute = new AttributeVO();
            tempAttribute.ok = true;
            tempAttribute.bookId = bookId;
            tempAttribute.id = rs.getInt(1);
            tempAttribute.key = rs.getString(2);
            tempAttribute.value = rs.getString(3);
            tempAttribute.time = System.currentTimeMillis() - start; // stop the clock
            attributeVector.addElement(tempAttribute);
        }

        AttributeVO[] ret = new AttributeVO[attributeVector.size()];
        attributeVector.copyInto(ret);
        ps.close();
        con.close();
        return ret;
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

```

getConnection()

```

/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

```

Testfall 30: lista alla författare

BMPcServer

getAuthors()

```

/**
 * @return an AuthorVO[] containing all authors
 */
public AuthorVO[] getAuthors()
{
    long start = System.currentTimeMillis(); // start the clock
    try {
        // Get the author beans
        Enumeration beans = null;
        beans = (Enumeration) authorHome.findAll();

        // Get the data
        Vector authorVector = new Vector();
        while(beans.hasMoreElements()) {
            AuthorBMPc bean = (AuthorBMPc) beans.nextElement();
            AuthorVO author = new AuthorVO();
            author.id = bean.getId();
            author.name = bean.getName();
            author.time = System.currentTimeMillis() - start; // stop the clock
            author.ok = true;
            authorVector.addElement(author);
        }
        AuthorVO[] ret = new AuthorVO[authorVector.size()];
        authorVector.copyInto(ret);
        return ret;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}

```

AuthorBMPc

ejbFindAll()

```

public Enumeration ejbFindAll()
    throws FinderException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    Vector ret = null;
    try {

```

```
        con = getConnection();

        ps = con.prepareStatement("select ID from BJORN.AUTHORS");
        ResultSet rs = ps.executeQuery();
        if (!rs.next()) {
            ex = new FinderException("No records found");
        } else {
            ret = new Vector();
            ret.addElement(new AuthorBMPCPK(rs.getInt(1)));
            while(rs.next()) {
                ret.addElement(new AuthorBMPCPK(rs.getInt(1)));
            }
        }
        rs.close();
    }
    catch (EJBException ee) {
        ex = ee;
    }
    catch (SQLException se) {
        ex = new EJBException(se.getMessage());
    }

    // Clean-up resources
    try {
        ps.close();
        con.close();
    }
    catch (Exception e) {}

    // Throw exception if error occurred
    if (ex != null) {
        if (ex instanceof EJBException)
            throw (EJBException)ex;
        if (ex instanceof FinderException)
            throw (FinderException)ex;
    }

    return (ret.size()>0) ? ret.elements() : null;
}
```

ejbLoad()

```
/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database.
 */
public void ejbLoad() {
    // Get the primary key
    AuthorBMPCPK key = (AuthorBMPCPK)context.getPrimaryKey();

    // Now load from the database
    try {
        readData(key);
    }
    catch(Exception ex) {
        throw new RuntimeException(ex.getMessage());
    }
}
```

readData()

```
/**
 * The method reads the data from database fields into this
 * bean.
 * @param key Primary Key used to load the bean
 * @exception javax.ejb.FinderException
 * @exception javax.ejb.EJBException
 */
private void readData(AuthorBMPCPK key)
    throws FinderException, EJBException {
    Exception ex = null;
    Connection con = null;
    PreparedStatement ps = null;
    try {
```

```

        con = getConnection();

        // Load from the data base
        ps = con.prepareStatement("select NAME from BJORN.AUTHORS where ID = ?");
        ps.setInt(1, key.id);
        ResultSet rs = ps.executeQuery();
        if (!rs.next())
            ex = new FinderException("Record not found");
        else {
            id = key.id;
            name = rs.getString(1);
            unsetModified();
        }
        rs.close();
    }
    catch (EJBException ee) {
        ex = ee;
    }
    catch (SQLException se) {
        ex = new EJBException(se.getMessage());
    }

    // Clean-up resources
    try {
        ps.close();
        con.close();
    }
    catch (Exception e) {}

    // Throw exception if error occurred
    if (ex != null) {
        if (ex instanceof EJBException)
            throw (EJBException)ex;
        if (ex instanceof FinderException)
            throw (FinderException)ex;
    }
}

```

getConnection()

```

/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}

```

BMPf

Koden i BMPfServer och AuthorBMPf är detta testfall identisk med motsvarande kod i BMPcServer och AuthorBMPc.

CMPServer

getAuthors()

```
/**
 * @return an AuthorVO[] containing all authors
 */
public AuthorVO[] getAuthors()
{
    long start = System.currentTimeMillis(); // start the clock
    try {
        // Get the author beans
        Enumeration beans = null;
        beans = (Enumeration) authorHome.findAll();

        // Get the data
        Vector authorVector = new Vector();
        while(beans.hasMoreElements()) {
            AuthorCMP bean = (AuthorCMP) beans.nextElement();
            AuthorVO author = new AuthorVO();
            author.id = bean.getId();
            author.name = bean.getName();
            author.time = System.currentTimeMillis() - start; // stop the clock
            author.ok = true;
            authorVector.addElement(author);
        }
        AuthorVO[] ret = new AuthorVO[authorVector.size()];
        authorVector.copyInto(ret);
        return ret;
    }
    catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

AuthorCMP

ejbLoad()

```
/**
 * The container invokes this method on the bean whenever it
 * becomes necessary to synchronize the bean's state with the
 * state in the database. This method is called after the container
 * has loaded the bean's state from the database.
 */
public void ejbLoad() {
    unsetModified();
}
```

SLSBServer

getAuthors()

```
/**
 * @return an AuthorVO[] containing all authors
 */
public AuthorVO[] getAuthors()
{
    long start = System.currentTimeMillis(); // start the clock
    Connection con = getConnection();
    try {
        PreparedStatement ps = con.prepareStatement("SELECT id, name FROM authors");
        ResultSet rs = ps.executeQuery();

        Vector authorVector = new Vector();
        while(rs.next()) {
            AuthorVO author = new AuthorVO();

```

```
        author.id = rs.getInt(1);
        author.name = rs.getString(2);
        author.time = System.currentTimeMillis() - start; // stop the clock
        author.ok = true;
        authorVector.addElement(author);
    }
    AuthorVO[] ret = new AuthorVO[authorVector.size()];
    authorVector.copyInto(ret);
    ps.close();
    con.close();
    return ret;
}
catch (Exception e) {
    e.printStackTrace();
}
return null;
}
```

getConnection()

```
/**
 * The method returns a jdbc connection to the database.
 * @exception javax.ejb.EJBException
 * @return jdbc connection to database
 */
private Connection getConnection()
    throws EJBException {
    // to do: modify datasource below to match your server
    final String DATASOURCE = "myJtsDataSource";
    try {
        InitialContext ic = new InitialContext();
        DataSource ds = (DataSource)ic.lookup(DATASOURCE);
        return ds.getConnection();
    }
    catch (Exception ex) {
        throw new EJBException(ex.getMessage());
    }
}
```


Appendix C Databasgenerator

Databasgenerering

För att generera databasens schema samt några lagrade procedurer används `createdb.sql`, vilken exekveras på följande sätt:

```
SQL> @createdb.sql
```

För att fylla databasen med data används `bjorn.ex.DatabaseCreator`. Kör det så här:

```
> java -cp .;e:\orant\jdbc\lib\classes12.zip  
    bjorn.ex.DatabaseCreator 4711 100 10
```

createdb.sql

```
-- Database creation script  
-- call using "@createdb.sql"  
--  
-- bjorn.brenander@idainfront.se 2001-07  
--  
  
SET TERMOUT ON  
PROMPT Setting up database...  
--SET TERMOUT OFF  
  
@h:/work/erasedb.sql  
  
PROMPT creating tables...  
CREATE TABLE authors (  
    id NUMBER PRIMARY KEY NOT NULL,  
    name VARCHAR(100)  
);  
  
CREATE TABLE books (  
    id NUMBER PRIMARY KEY NOT NULL,  
    title VARCHAR(100),  
    author NUMBER,  
    FOREIGN KEY (author) REFERENCES authors (id) ON DELETE SET NULL  
);  
  
CREATE TABLE attributes (  
    id NUMBER PRIMARY KEY NOT NULL,  
    key VARCHAR(30),  
    value VARCHAR(100),  
    book NUMBER NOT NULL,  
    FOREIGN KEY (book) REFERENCES books (id) ON DELETE CASCADE  
);  
  
PROMPT Creating indices  
  
CREATE INDEX authorsByName ON authors (name);  
CREATE INDEX booksByTitle ON books (title);  
CREATE INDEX booksByAuthor ON books (author);  
CREATE INDEX attributesByBook ON attributes (book);  
CREATE INDEX attributesByBookAndKey ON attributes (book, key);  
CREATE INDEX attributesByKeyAndValue ON attributes (key, value);  
  
prompt creating sequences  
  
create sequence authorseq  
    increment by 1  
    start with 1
```

```
        nocache;

create sequence bookseq
    increment by 1
    start with 1
    nocache;

create sequence attributeseq
    increment by 1
    start with 1
    nocache;

prompt creating stored procedures
CREATE OR REPLACE PROCEDURE add_author
( in_name IN VARCHAR, out_id OUT NUMBER)
AS
BEGIN
    INSERT INTO authors (id, name)
        VALUES (authorseq.nextval, in_name)
        RETURNING id INTO out_id;
END;
/

CREATE OR REPLACE PROCEDURE add_book
( in_title IN VARCHAR, in_author IN NUMBER, out_id OUT NUMBER )
AS
BEGIN
    INSERT INTO books (id, title, author)
        VALUES (bookseq.nextval, in_title, in_author)
        RETURNING id INTO out_id;
END;
/

CREATE OR REPLACE PROCEDURE add_attribute
( in_key IN VARCHAR, in_value IN VARCHAR, in_book IN NUMBER,
  out_id OUT NUMBER )
AS
BEGIN
    INSERT INTO attributes (id, key, value, book)
        VALUES (attributeseq.nextval, in_key, in_value, in_book)
        RETURNING id INTO out_id;
END;
/

COMMIT;

set termout on
prompt Done.
```

Källkod

DatabaseCreator

```
package bjorn.ex;
/*****
 * DatabaseCreator connects to a database and creates authors,
 * books and attributes.
 *
 * Usage: java bjorn.ex.DatabaseCreator <seed> <# authors>
 *        <# books/author>
 * Classpath: .;e:\orant\jdbc\lib\classes12.zip
 *
 * bjorn.brenander@idainfront.se 2001-07
 *****/

import bjorn.ex.util.*;
import java.util.*;

public class DatabaseCreator {
    Random randomizer = null;
    long defaultSeed = 4711;
```

```

int defaultAuthors = 2;
int defaultBooks = 2;
static int nextUnusedId = 1;

static String[] firstNames = {"Abel", "Abraham", "Ada", "Adam", "Adela", "Adolf",
    // [...] (names taken from Swedish name days list)
    "Yvonne", "Åke", "Åsa", "Åslög", "Örjan", "Östen" };
static String[] lastNames = {"Alsén", "Bodén", "Bonet", "Borgh", "Brandt", "Bratell",
    "Brenander", "Brodin", "Ek", "Embretzen", "Erlands", "Fellnert", "Gibson",
    "Grimsell", "Heijbel", "Herling", "Jensen", "Meijer", "Modéer", "Niward", "Nordin",
    "Olsén", "Råsbjer", "Samson", "Staaf", "Svall", "Teite", "Thulin", "Schantz",
    "Törner", "Urdell", "Wohlgemuth"};
static String[] lastNamesPre = {"Palm", "Ek", "Björk", "Björn", "Gran", "Berg", "Bode",
    "Borg", "By", "Ceder", "Carl", "Karl", "Dahl", "Dal", "Ene", "Eng", "Ång", "Fors",
    "Grön", "Hammar", "Jone", "Kihl", "Kil", "Kung", "Kvarn", "Lind", "Linde", "Lund",
    "Lundh", "Malm", "Nord", "Pihl", "Rome", "Sand", "Stom", "Sjö", "Sjöö",
    "Söder", "Väster", "Öster", "Öst", "Väst", "Syd", "Tov", "Törn", "Thörn", "Ydre",
    "Å", "Ö", "Örte", "Fuhl"};
static String[] lastNamesPost = {"lund", "gren", "berg", "ström", "ås", "qvist", "kvist",
    "quist", "strand", "dal", "dahl", "sjö", "å", "au", "stedt", "sten", "kwist", "man",
    "falk", "heim", "katt", "vill", "borg", "blad", "båge", "ryd"};
static String[] titleAdjective = {"Big", "Small", "Very Small", "Dangerous",
    "Incredible", "Inappropriate", "Pornographic", "Mad", "Crazy", "Slimy", "Red", "Old",
    "Zombie", "Black", "Fearless" };
static String[] titleSubject = {"Restaurant", "Dog", "Killer", "Hotel New Hampshire",
    "Eye", "Clown", "Rock star", "Hamster", "Solar system", "Earth", "Space", "Blood",
    "Pen", "Computer" };
static String[] titlePredicate = {"sleeps", "follows", "grows", "drips", "lies", "sees",
    "frightens", "ate", "killed", "was killed", "explores", "got lost"};
static String[] titleEnding = {"in 1984", "beneath", "within", "at the end", "my
    hamster", "", "", "harmlessly", "furiously", "with no fear", "the Universe", "in
    Windows", "without a trace"};

public DatabaseCreator () {
}

/*****
 * main()
 *****/
static public void main(String args[]) {

    // check args
    if (args.length != 3) {
        System.err.println("Usage: DatabaseCreator <seed> "
            + "<# authors> <# books/author>");
        return;
    }
    long seed = Long.parseLong(args[0]);
    int authorsToCreate = Integer.parseInt(args[1]);
    int booksToCreate = Integer.parseInt(args[2]);

    // Setup
    DatabaseCreator dbc = new DatabaseCreator();
    dbc.randomizer = new Random(seed);
    DatabaseHelper dbh = new DatabaseHelper();
    dbh.connect();

    // Loop over authors
    for (int i = 0; i < authorsToCreate; i++) {
        AuthorVO author = dbc.genAuthor();
        GenVO result = dbh.addAuthor(author);
        if (!result.ok) {
            System.err.println("Error: could not add author");
            return;
        }
        int currentAuthorId = result.id;
        //System.out.println("Created author " + result.id + ", " + author.name);
        System.out.print("."); // Tell user we are still working
        // Loop over books
        for (int j = 0; j < booksToCreate; j++) {
            BookVO book = dbc.genBook(currentAuthorId);
            result = dbh.addBook(book);
            if (!result.ok) {
                System.err.println("Error: could not add book");
                return;
            }
        }
    }
}

```

```
        } //System.out.println("Created book " + result.id + ", " + book.title);
    }

    // Clean up
    dbh.disconnect();
}

/*****
 * Generator methods
 *****/

/*
 * Author
 */
public AuthorVO genAuthor() {
    long then = System.currentTimeMillis();
    AuthorVO author = new AuthorVO();
    author.name = genName();
    //author.id = getId();
    author.time = System.currentTimeMillis() - then;
    return author;
}

/*
 * Book
 */
public BookVO genBook(int authorId) {
    long then = System.currentTimeMillis();
    BookVO book = new BookVO();
    book.authorId = authorId;
    book.title = genTitle();
    //book.id = getId();
    book.attributes = genAttributes(book.id);
    book.time = System.currentTimeMillis() - then;
    return book;
}

/*
 * Attributes
 */
public Vector genAttributes(int bookId) {
    Vector vec = new Vector();
    AttributeVO attribute = new AttributeVO();
    attribute.bookId = bookId;
    attribute.key = "published";
    attribute.value = genPublished();
    attribute.id = getId();
    vec.add(attribute);

    attribute = new AttributeVO();
    attribute.bookId = bookId;
    attribute.key = "price";
    attribute.value = genPrice();
    attribute.id = getId();
    vec.add(attribute);

    attribute = new AttributeVO();
    attribute.bookId = bookId;
    attribute.key = "pages";
    attribute.value = genPages();
    attribute.id = getId();
    vec.add(attribute);

    attribute = new AttributeVO();
    attribute.bookId = bookId;
    attribute.key = "isbn";
    attribute.value = genISBN();
    attribute.id = getId();
    vec.add(attribute);
    return vec;
}
```

```

/*
 * Title
 */
public String genTitle() {
    switch(randomizer.nextInt(5)) {
        case 0:
            return "The"
                + " " + randomString(titleAdjective)
                + " " + randomString(titleSubject)
                + " " + randomString(titlePredicate)
                + " " + randomString(titleEnding);
        case 1:
            return randomString(titleSubject)
                + " " + randomString(titlePredicate)
                + " " + randomString(titleEnding);
        case 2:
            return randomString(titleSubject)
                + " " + randomString(titleEnding);
        case 3:
            return randomString(firstNames) + " and the"
                + " " + randomString(titleSubject)
                + " " + randomString(titlePredicate)
                + " " + randomString(titleEnding);
        case 4:
            return randomString(firstNames) + " and the"
                + " " + randomString(titleAdjective)
                + " " + randomString(titleSubject)
                + " " + randomString(titlePredicate)
                + " " + randomString(titleEnding);
        default:
            return "The Unknown Book";
    }
}

/*
 * Price
 */
public String genPrice() {
    return String.valueOf(randomizer.nextInt(700) + 1);
}

/*
 * Pages
 */
public String genPages() {
    return String.valueOf(randomizer.nextInt(2000) + 2);
}

/*
 * ISBN
 * NB: a real ISBN is always ten digits...
 * http://www.isbn.spk-berlin.de/
 */
public String genISBN() {
    return String.valueOf(randomizer.nextInt(100000) + 1)
        + "-" + String.valueOf(randomizer.nextInt(10000000) + 1)
        + "-" + String.valueOf(randomizer.nextInt(10000000) + 1)
        + "-" + String.valueOf(randomizer.nextInt(200) + 1801)
        + "-" + String.valueOf(randomizer.nextInt(10));
}

/*
 * Published
 */
public String genPublished() {
    return String.valueOf(randomizer.nextInt(200) + 1801);
}

/*
 * Names
 */
public String genFirstName() {

```

```
        return randomString(firstNames);
    }

    public String genLastName() {
        int i = randomizer.nextInt(4);
        switch(i) {
            case 0: // compound
                return randomString(lastNamesPre)
                    + randomString(lastNamesPost);
            case 1: // half
                return randomString(lastNamesPre);
            case 2: // simple
                return randomString(lastNames);
            case 3: // son
                return randomString(firstNames) + "sson";
            default:
                return "Nemo";
        }
    }

    public String genName() {
        return genFirstName() + " " + genLastName();
    }

    /**
     * Utility methods
     */
    public String randomString(String[] strings) {
        return strings[randomizer.nextInt(strings.length)];
    }

    static public int getId() {
        return nextUnusedId++;
    }
}
```

DatabaseHelper

```
package bjorn.ex;
/*****
 * DatabaseHelper supplies helpful methods for creating data
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/

import bjorn.ex.util.*;
import java.sql.*;
import java.util.*;

public class DatabaseHelper
{
    Connection con = null;
    String url = "jdbc:oracle:thin:@localhost:1521:test";
    String user = "bjorn";
    String password = "bjorn";

    public DatabaseHelper() {
    }

    /**
     * @param author an AuthorVO representing the author to be added
     * @return a GenVO containing the id of the inserted author
     */
    public GenVO addAuthor(AuthorVO author) {
        GenVO ret = new GenVO();
        ret.start();
        ret.ok = false;
        try {
            CallableStatement cs = con.prepareCall("CALL add_author (?, ?)");
            cs.setString(1, author.name);
            cs.registerOutParameter(2, java.sql.Types.INTEGER);
            cs.execute();
            con.commit();
        }
    }
}
```

```

        ret.id = cs.getInt(2);
        cs.close();
        ret.ok = true;
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    finally {
        ret.stop();
        return ret;
    }
}

/**
 * @param book a BookVO representing the book to be added
 * @return a GenVO containing the id of the inserted book
 */
public GenVO addBook(BookVO book) {
    GenVO ret = new GenVO();
    ret.start();
    ret.ok = false;
    try {
        // The book
        CallableStatement cs = con.prepareCall("CALL add_book (?, ?, ?)");
        cs.setString(1, book.title);
        cs.setInt(2, book.authorId);
        cs.registerOutParameter(3, java.sql.Types.INTEGER);
        cs.execute();
        int bookId = cs.getInt(3);
        cs.close();
        // The attributes
        if (book.attributes != null) {
            Enumeration enum = book.attributes.elements();
            CallableStatement csa = con.prepareCall("CALL add_attribute (?, ?, ?, ?)");
            while (enum.hasMoreElements()) {
                AttributeVO attribute = (AttributeVO)enum.nextElement();
                csa.clearParameters();
                csa.setString(1, attribute.key);
                csa.setString(2, attribute.value);
                csa.setInt(3, bookId);
                csa.registerOutParameter(4, java.sql.Types.INTEGER);
                csa.execute();
                // csa.getInt(4);
            }
            csa.close();
        }
        con.commit();
        ret.id = bookId;
        ret.ok = true;
    }
    catch(Exception e) {
        if (con != null) {
            con.rollback();
        }
        e.printStackTrace();
    }
    finally {
        ret.stop();
        return ret;
    }
}

/**
 * @param id the id of the desired author
 * @return an AuthorVO possibly containing the author
 */
public AuthorVO getAuthorById(int id) {
    AuthorVO author = new AuthorVO();
    author.start();
    author.id = id;
    author.ok = false;
    try {
        Statement select = con.createStatement();
        ResultSet result = select.executeQuery("SELECT name FROM authors WHERE id = "
            + String.valueOf(id));
    }
}

```

```
        if (result.next()) {
            author.name = result.getString(1);
            author.ok = true;
        }
    }
    catch(Exception e) {
        e.printStackTrace();
    }
    finally {
        author.stop();
        return author;
    }
}

public void connect() {
    try {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con = DriverManager.getConnection(url, user, password);
    }
    catch ( Exception e) {
        e.printStackTrace();
    }
}

public void disconnect() {
    if (con != null) {
        try {
            con.close();
        }
        catch ( Exception e) {
            e.printStackTrace();
        }
    }
    else {
        System.err.println("Not connected!");
    }
}
}
```

Appendix D Testklient

Källkod

```
package bjorn.ex;

import bjorn.ex.util.*;
import javax.ejb.*;
import javax.naming.*;
import java.util.*;
import java.rmi.*;
import java.text.DecimalFormat;

public class Client {

    static String url = "t3://194.218.15.134:7001"; // kiara
    static String user = null;
    static String password = null;

    static String serverType = null;
    static int testCase = 0;
    static int noOfRuns = 0;
    static String paramA = null;
    static String paramB = null;

    /**
     * Constructor
     */
    public Client () {
    }

    /**
     * main
     */
    static public void main(String args[]) {

        parseArguments(args);
        if (noOfRuns < 1) {
            System.err.println("NoOfRuns must be positive.");
            System.exit(1);
        }

        try {
            ServerInterface server = getServerBean(serverType);
            int severalTimes = 1000;

            // Default values
            int defaultAuthorId = 17;
            int defaultBookId = 17;
            int defaultStartId = 1000;
            String defaultKey = "price";
            String defaultNamePattern = "%e%";
            String defaultTitlePattern = "%Dog%";
            String defaultValuePattern = "%9%";
            String defaultName = "New author";
            String defaultTitle = "New book";

            int bookId = defaultBookId;
            int authorId = defaultAuthorId;
            int startId = defaultStartId;
            String key = defaultKey;
            String namePattern = defaultNamePattern;
            String titlePattern = defaultTitlePattern;
            String valuePattern = defaultValuePattern;
            String name = defaultName;
            String title = defaultTitle;

            int errors = 0; // errors encountered while testing
            int noOfMatches = 1; // number of matches when searching
            long[] times = new long[noOfRuns];

            switch(testCase) {
```

```

case(10): // Get Author
    if (paramA != null) {
        authorId = Integer.parseInt(paramA);
    }
    //System.out.println("Doing getAuthorById(" + authorId + ") "
    // + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        AuthorVO ret = server.getAuthorById(authorId);
        if (ret.ok) {
            times[i] = ret.time;
        }
        else {
            errors++;
        }
    }
    break;

case(11): // Get Book
    if (paramA != null) {
        bookId = Integer.parseInt(paramA);
    }
    //System.out.println("Doing getBookById(" + bookId + ") "
    // + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        BookVO ret = server.getBookById(bookId);
        //System.out.println(ret);
        if (ret.ok) {
            times[i] = ret.time;
        }
        else {
            errors++;
        }
    }
    break;

case(20): // Update attribute of Author
    if (paramA != null) {
        authorId = Integer.parseInt(paramA);
    }
    //System.out.println("Doing updateAuthor " + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        AuthorVO tempAuthor = server.getAuthorById(authorId);
        tempAuthor.name = "Changed " + System.currentTimeMillis();
        GenVO ret = server.updateAuthor(tempAuthor);
        if (ret.ok) {
            times[i] = ret.time;
        }
        else {
            errors++;
        }
    }
    break;

case(21): // Update external attribute of Book
    if (paramA != null) {
        bookId = Integer.parseInt(paramA);
    }
    if (paramB != null) {
        key = paramB;
    }
    //System.out.println("Doing updateAttribute " + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        AttributeVO tempAttribute =
            server.getAttributeByBookIdAndKey(bookId, key);
        //System.out.println("About to change attribute " + tempAttribute);
        tempAttribute.value = "Changed " + System.currentTimeMillis();
        GenVO ret = server.updateAttribute(tempAttribute);
        if (ret.ok) {
            times[i] = ret.time;
            tempAttribute = server.getAttributeByBookIdAndKey(bookId, key);
            //System.out.println("Changed attribute " + tempAttribute);
        }
        else {
            errors++;
        }
    }
    break;

case(30): // Get All authors
    //System.out.println("Doing getAuthors " + noOfRuns + " times...");

```

```

        for (int i=0; i<noOfRuns; i++) {
            AuthorV0[] ret = server.getAuthors();
            if (ret[ret.length-1].ok) {
                times[i] = ret[ret.length-1].time;
                noOfMatches = ret.length;
            }
            else {
                errors++;
            }
        }
        break;
case(31): // Get all books
    //System.out.println("Doing getBooks " + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        BookV0[] ret = server.getBooks();
        if (ret[ret.length-1].ok) {
            times[i] = ret[ret.length-1].time;
            noOfMatches = ret.length;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(40): // Get authors with matching names
    if (paramA != null) {
        namePattern = paramA;
    }
    //System.out.println("Doing getAuthorsByName " + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        AuthorV0[] ret = server.getAuthorsByName(namePattern);
        if (ret[ret.length-1].ok) {
            times[i] = ret[ret.length-1].time;
            noOfMatches = ret.length;
        }
        else {
            errors++;
        }
    }
    break;
case(400): // Get authors with matching names
    if (paramA != null) {
        namePattern = paramA;
    }
    //System.out.println("Doing getAuthorsByName " + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        NumV0 ret = server.searchAuthorsByName(namePattern);
        if (ret.ok) {
            noOfMatches = ret.number;
            times[i] = ret.time;
            //System.out.println(ret);
        }
        else {
            errors++;
        }
    }
    break;
case(41): // Get books with matching titles
    if (paramB != null) {
        titlePattern = paramB;
    }
    //System.out.println("Doing getBooksByTitle" + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        BookV0[] ret = server.getBooksByTitle(titlePattern);
        if (ret[ret.length-1].ok) {
            times[i] = ret[ret.length-1].time;
            noOfMatches = ret.length;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(410): // Get books with matching titles
    if (paramB != null) {

```

```

        titlePattern = paramB;
    }
    //System.out.println("Doing getBooksByTitle" + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        NumVO ret = server.searchBooksByTitle(titlePattern);
        if (ret.ok) {
            noOfMatches = ret.number;
            times[i] = ret.time;
            //System.out.println(ret);
        }
        else {
            errors++;
        }
    }
    break;
case(42): // Get books of certain author
    if (paramA != null) {
        authorId = Integer.parseInt(paramA);
    }
    //System.out.println("Doing getBooksByAuthorId "
    // + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        BookVO[] ret = server.getBooksByAuthorId(authorId);
        if (ret[ret.length-1].ok) {
            times[i] = ret[ret.length-1].time;
            noOfMatches = ret.length;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(420): // Get books of certain author
    if (paramA != null) {
        authorId = Integer.parseInt(paramA);
    }
    //System.out.println("Doing getBooksByAuthorId "
    // + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        NumVO ret = server.searchBooksByAuthorId(authorId);
        if (ret.ok) {
            noOfMatches = ret.number;
            times[i] = ret.time;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(43): // Get books with matching attribute
    if (paramA != null) {
        key = paramA;
    }
    if (paramB != null) {
        valuePattern = paramB;
    }
    //System.out.println("Doing getBooksByAttribute "
    // + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        BookVO[] ret = server.getBooksByAttribute(key, valuePattern);
        if (ret[ret.length-1].ok) {
            times[i] = ret[ret.length-1].time;
            noOfMatches = ret.length;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(430): // Get books with matching attribute
    if (paramA != null) {
        key = paramA;
    }
    if (paramB != null) {

```

```

        valuePattern = paramB;
    }
    //System.out.println("Doing getBooksByAttribute "
    //    + noOfRuns + " times...");
    for (int i=0; i<noOfRuns; i++) {
        NumVO ret = server.searchBooksByAttribute(key, valuePattern);
        if (ret.ok) {
            noOfMatches = ret.number;
            times[i] = ret.time;
            //System.out.println(ret[i]);
        }
        else {
            errors++;
        }
    }
    break;
case(50): // Create author (STARTID, NAME)
    if (paramA != null) {
        startId = Integer.parseInt(paramA);
    }
    if (paramB != null) {
        name = paramB;
    }
    //System.out.println("Doing createAuthor " + noOfRuns
    //    + " times starting with " + startId + "...");
    AuthorVO author = new AuthorVO();
    for (int i=0; i<noOfRuns; i++) {
        author.name = name + " #" + i;
        author.id = startId + i * severalTimes;
        GenVO ret = server.addAuthor(author);
        if (ret.ok) {
            times[i] = ret.time;
        }
        else {
            errors++;
        }
    }
    break;
case(51): // Create book (STARTID, TITLE)
    if (paramA != null) {
        startId = Integer.parseInt(paramA);
    }
    if (paramB != null) {
        title = paramB;
    }
    //System.out.println("Doing createBook " + noOfRuns
    //    + " times starting with " + startId + "...");
    BookVO book = new BookVO();
    book.authorId = 1; // probably exists
    // Create attributes
    book.attributes = new Vector();
    AttributeVO tmpAttribute = new AttributeVO();
    tmpAttribute.key = "price";
    tmpAttribute.value = "9";
    book.attributes.addElement(tmpAttribute);
    tmpAttribute = new AttributeVO();
    tmpAttribute.key = "pages";
    tmpAttribute.value = "99";
    book.attributes.addElement(tmpAttribute);
    tmpAttribute = new AttributeVO();
    tmpAttribute.key = "isbn";
    tmpAttribute.value = "5-3425-2234-1";
    book.attributes.addElement(tmpAttribute);
    tmpAttribute = new AttributeVO();
    tmpAttribute.key = "published";
    tmpAttribute.value = "2001";
    book.attributes.addElement(tmpAttribute);
    for (int i=0; i<noOfRuns; i++) {
        book.title = title + " #" + i;
        book.id = startId + i * severalTimes;
        GenVO ret = server.addBook(book);
        if (ret.ok) {
            times[i] = ret.time;
        }
        else {
            errors++;
        }
    }
}

```

```

        }
        break;
    case(60): // Remove Author (STARTID)
        if (paramA != null) {
            startId = Integer.parseInt(paramA);
        }
        //System.out.println("Doing removeAuthor " + noOfRuns
        // + " times starting with " + startId + "...");
        for (int i=0; i<noOfRuns; i++) {
            GenVO ret = server.removeAuthor(startId+i*severalTimes);
            if (ret.ok) {
                times[i] = ret.time;
            }
            else {
                errors++;
            }
        }
        break;
    case(61): // Remove Book (STARTID)
        if (paramA != null) {
            startId = Integer.parseInt(paramA);
        }
        //System.out.println("Doing removeBook " + noOfRuns
        // + " times starting with " + startId + "...");
        for (int i=0; i<noOfRuns; i++) {
            GenVO ret = server.removeBook(startId+i*severalTimes);
            if (ret.ok) {
                times[i] = ret.time;
            }
            else {
                errors++;
            }
        }
        break;
    case(0):
    default:
        System.err.println("Unknown test case " + testCase + ".");
        System.exit(1);
}

// Bail out if errors occurred
if (errors > 0) {
    System.err.println(errors + " errors!");
    System.exit(1);
}

// Do statistics
long totalTime = 0;
long totalSquaredTime = 0;
double averageTime = 0;
double medianTime = 0;
double minTime = 999999999;
double maxTime = 0;
double variance = 0;
double standardDeviation = 0;

String sep = "\t";

System.out.print(serverType + " " + testCase);
for (int i=0; i<times.length; i++) {
    System.out.print(sep + times[i]); // Output time
    totalTime += times[i]; // Add to sum
    totalSquaredTime += times[i] * times[i]; // Add to sum of squares
    if (times[i] < minTime)
        minTime = times[i];
    if (times[i] > maxTime)
        maxTime = times[i];
}
System.out.println();

averageTime = totalTime / noOfRuns; // Calculate arithmetic mean
medianTime = median(times); // Calculate median
// Calculate variance
variance = (totalSquaredTime - ((totalTime * totalTime)/times.length))
    / (times.length - 1);
// Calculate standard deviation
standardDeviation = java.lang.Math.sqrt(variance);

```

```

        DecimalFormat df = new DecimalFormat("#####.##");

        System.out.print(serverType + sep + testCase + sep);
        System.out.print(df.format(minTime) + sep);
        System.out.print(df.format(medianTime) + sep);
        System.out.print(df.format(averageTime) + sep);
        System.out.print(df.format(standardDeviation) + sep);
        System.out.print(df.format(maxTime) + sep);
        System.out.print(noOfRuns + sep + noOfMatches);
        System.out.println();

        // remove the server bean
        server.remove();

    }
    catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * Get hold of the desired server bean
 * @param type a string naming the type
 * @return the server bean as a ServerInterface object
 */
static ServerInterface getServerBean(String type) {
    ServerInterface ret = null;
    try {
        Context ctx = getInitialContext();
        ServerHandleFactoryHome home = (ServerHandleFactoryHome)
            ctx.lookup("ServerHandleFactory");
        ServerHandleFactory serverHandleFactory = home.create();

        if (type.equalsIgnoreCase("CMP")) {
            //System.out.println("Using CMP method");
            ret = serverHandleFactory.getCmpServerBean();
        }
        else if (type.equalsIgnoreCase("CMPc")) {
            System.err.println("CMPc not yet implemented.");
            //ret = serverHandleFactory.getCmpcServerBean();
        }
        else if (type.equalsIgnoreCase("BMPf")) {
            //System.out.println("Using BMPf method");
            ret = serverHandleFactory.getBMPfServerBean();
        }
        else if (type.equalsIgnoreCase("BMPc")) {
            //System.out.println("Using BMPc method");
            ret = serverHandleFactory.getBMPcServerBean();
        }
        else if (type.equalsIgnoreCase("BMPr")) {
            System.err.println("BMPr not yet implemented.");
            //ret = serverHandleFactory.getBMPrServerBean();
        }
        else if (type.equalsIgnoreCase("SLSB")) {
            //System.out.println("Using SLSB method");
            ret = serverHandleFactory.getSLSBServerBean();
        }
        else if (type.equalsIgnoreCase("JAVA")) {
            System.err.println("JAVA not yet implemented.");
            //ret = serverHandleFactory.getJAVAServerBean();
        }
        else {
            System.err.println("No server type " + type + ". Using default.");
            ret = serverHandleFactory.getDefaultServerBean();
        }
    }
    catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
    return ret;
}

/**
 * Parse arguments
 */

```

```

* arguments in order:
* - type of server (CMP,BMPf,BMPc,SLSB)
* - test case (number)
* - number of test runs (number)
* - paramA, first parameter of case
* - paramB, second parameter of case
*
* @param args arguments from main() call
*/
public static void parseArguments(String args[]) {
    if (args != null && args.length > 0) {
        for (int i = 0; i < args.length; i++) {
            switch(i) {
                case 0: // type
                    serverType = args[i];
                    break;
                case 1: // case
                    testCase = Integer.parseInt(args[i]);
                    break;
                case 2: // number of runs
                    noOfRuns = Integer.parseInt(args[i]);
                    break;
                case 3: // parameter A
                    paramA = args[i];
                    break;
                case 4: // parameter B
                    paramB = args[i];
                    break;
                default:
                    System.err.println("Too many arguments!");
                    System.exit(1);
            }
        }
    }
    else {
        System.err.println("Too few arguments!");
        System.exit(1);
    }
}

/**
 * get a context
 * @return Context
 */
public static Context getInitialContext()
throws NamingException
{
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY, "weblogic.jndi.WLInitialContextFactory");
    p.put(Context.PROVIDER_URL, url);

    if (user != null) {
        p.put(Context.SECURITY_PRINCIPAL, user);
        if (password == null)
            password = "";
        p.put(Context.SECURITY_CREDENTIALS, password);
    }
    return new InitialContext(p);
}

private static void insertionSort( long[] a, int low, int high )
{
    for( int p = low + 1; p <= high; p++ )
    {
        long tmp = a[p];
        int j;
        for( j = p; j > low && tmp < ( a[j - 1] ); j-- ) {
            a[j] = a[j-1];
        }
        a[j] = tmp;
    }
}

public static double median(long[] a)
{
    long[] tmp = a;

```

```
        insertionSort(tmp, 0, tmp.length-1);
        int middle = tmp.length / 2;
        if (tmp.length%2 == 0) {
            double foo = tmp[middle-1]+tmp[middle];
            return foo/2;
        }
        else {
            return tmp[middle];
        }
    }
}
```


Appendix E Värdeobjekt

GenVO

```
package bjorn.ex.util;
/*****
 * A value object
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/

import java.io.*;

public class GenVO implements java.io.Serializable
{
    public int id;
    public long time;
    public boolean ok;

    public String toString() {
        return getClass().getName()
            + ": id=" + String.valueOf(id) + ", "
            + "time=" + String.valueOf(time) + " ms, "
            + "ok=" + String.valueOf(ok);
    }

    public void start() {
        time = System.currentTimeMillis();
    }

    public void stop() {
        time = System.currentTimeMillis() - time;
    }
}
```

NumVO

```
package bjorn.ex.util;
/*****
 * A value object
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/

import java.io.*;

public class NumVO extends GenVO implements java.io.Serializable
{
    public int number;

    public String toString() {
        String foo = getClass().getName()
            + "number=" + String.valueOf(number) + ", "
            + "id=" + String.valueOf(id) + ", "
            + "time=" + String.valueOf(time) + " ms, "
            + "ok=" + String.valueOf(ok);
        return foo;
    }
}
```

AuthorVO

```
package bjorn.ex.util;
/*****
 * A value object
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/
```

```
import java.io.*;

public class AuthorVO extends GenVO implements java.io.Serializable
{
    public String name;

    public String toString() {
        return getClass().getName()
            + ": name='" + name + "' "
            + "id=" + String.valueOf(id) + ", "
            + "time=" + String.valueOf(time) + " ms, "
            + "ok=" + String.valueOf(ok);
    }
}
```

BookVO

```
package bjorn.ex.util;
/*****
 * A value object
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/

import java.io.*;

public class BookVO extends GenVO implements java.io.Serializable
{
    public String title;
    public int authorId;
    public java.util.Vector attributes;

    public String toString() {
        String foo = getClass().getName()
            + ": title='" + title + "' "
            + "id=" + String.valueOf(id) + ", "
            + "authorId=" + String.valueOf(authorId) + ", "
            + "time=" + String.valueOf(time) + " ms, "
            + "ok=" + String.valueOf(ok);
        if (attributes != null ) {
            java.util.Enumeration enum = attributes.elements();
            while (enum.hasMoreElements()) {
                foo = foo + "\n\t" + ((AttributeVO)enum.nextElement()).toString();
            }
        }
        else {
            foo = foo + "\n\t no attributes.";
        }
        return foo;
    }
}
```

AttributeVO

```
package bjorn.ex.util;
/*****
 * A value object
 *
 * bjorn.brenander@idainfront.se 2001-07-24
 *****/

import java.io.*;

public class AttributeVO extends GenVO implements java.io.Serializable
{
    public String key;
    public String value;
    public int bookId;

    public String toString() {
        return getClass().getName()
            + ": '" + key + "'=" + value + "' , "
    }
}
```

```
        + "bookId=" + String.valueOf(bookId) + ", "  
        + "id=" + String.valueOf(id) + ", "  
        + "time=" + String.valueOf(time) + " ms, "  
        + "ok=" + String.valueOf(ok);  
    }  
}
```