Eine Fallstudie: Einbindung von Legacy-Datenbanken über JDBC

Dr. Rainer Kerth*

Object Technology Practice, IBM Unternehmensberatung GmbH

Zusammenfassung Wir stellen in diesem Artikel eine Architektur zur Einbindung von hierarchischen Legacy-Datenbanken in eine objektorientierte Anwendungsentwicklung in Java vor. Grundlage dieser Einbindung ist ein maßgeschneiderter JDBC-Treiber, mit dessen Hilfe ein Zugriff auf die komplexen Datenstrukturen auf der Legacy-Seite transparent vorgenommen werden kann. Der Treiber unterstützt einen großen Teil der Funktionalitäten von JDBC 1.22.
Die Vorteile dieses Ansatzes liegen im wesentlichen in der Bereitstellung der hierarchischen Daten unter einer standardisierten, relationalen Schnittstelle. Dadurch besteht insbesondere die Möglichkeit, die Daten durch kommerziell verfügbare objektrelationale Abbildungen in Objekte umzuwandeln. Die Schwierigkeiten des Ansatzes liegen z.Zt. noch im Bereich Transaktionskoordination.
Die Architektur wurde in einem Pilotprojekt in einem Unternehmen der Versicherungsbranche vollständig implementiert.

1 Projektkontext und Ziele

F In einem bundesdeutschen Versicherungsunternehmen wurde seit Mitte der siebziger Jahre eine traditionelle Anwendungsentwicklung in COBOL betrieben. Die im Laufe der Zeit angewachsenen Programmbibliotheken laufen problemlos auf einem Hostsystem, sind aber relativ schwer zu pflegen. Softwaretechnisch bestehen insofern Probleme, als daß die Programme sehr stark durch proprietäre Standards verschiedener Hersteller geprägt ist. Dies führt zu einer starken Abhängigkeit von diesen Herstellern und zu einem Mangel an Flexibilität bei Make or BuyEntscheidungen. Desweiteren ist in den gewachsenen Bibliotheken häufig eine Vermischung von Geschäfts-, Datenzugriffs- und Oberflächenlogik festzustellen. Dadurch wird in ganz erheblichem Maße die Wiederverwendung von bestehendem Code erschwert.

Um die beiden angesprochenen Problemkreise anzugehen, hat sich das genannte Unternehmen entschlossen, in einem Pilotprojekt die Möglichkeiten und Risiken einer alternativen Anwendungsentwicklung zu evaluieren. Vorgabe war zunächst eine möglichst weitgehende Ausrichtung an offenen Standards, um auf

* Dieser Artikel präsentiert Ergebnisse, die der Autor gemeinsam mit Richard Stubbs (UIST Consultants Ltd.) erarbeitet hat.

diese Weise die bestehenden Abhängigkeiten von proprietären Standards zu reduzieren. Auch bestand von Anfang an die klare Vorstellung, sich der objektorientierten Anwendungsentwicklung zu öffnen, die mehr und mehr eine führende Rolle als modernes Entwicklungsparadigma einnimmt.

Diese Öffnung sollte aber unter Berücksichtigung der bestehenden Strukturen im Legacy-Bereich vorgenommen werden. Insbesondere war ein entscheidendes Erfolgskriterium für das Pilotprojekt eine weitgehende Einbindung dieser Strukturen in die neue Anwendungsentwicklung. Damit war nicht nur die Integration des vorhandenen Hostsystems als Datenserver, sondern soweit möglich auch eine Integration existierender Host-Programme gemeint.

Die Anwendungsentwicklung im Rahmen des Pilotprojektes erfolgte auf PCs unter Windows NT 4.0. Als LAN Server diente ein AIX-Rechner der Firma Bull. Bei dem zu integrierenden Hostsystem handelte es sich ebenfalls um einen Rechner aus dem Hause Bull mit GCOS 8. Als relationale Datenbank wurde Informix auf AIX verwendet. Auf GCOS 8 standen die Stammdaten des Unternehmens in der hierarchische Datenbank IDS/2 zur Verfügung. Als CASE-Tool wurde Innovator 6.1 eingesetzt, als Entwicklungsumgebung diente Visual Age for Java 2.0 Enterprise. Der verwendete CORBA-ORB war VisiBroker 3.2.

2 Die Auswahl der Programmiersprache

Als Programmiersprache für die neu aufzusetzende Anwendungsentwicklung wurde Java gewählt. Die Gründe für diese Entscheidung waren die folgenden:

- die leichte Erlernbarkeit der Sprache: diese war für das Unternehmen, das zu Beginn des Projektes nur über geringe Erfahrungen in der Objekttechnologie verfügte, ein wichtiger Aspekt und hat aus heutiger Sicht viel zum Erfolg des Projektes beigetragen. Es bestanden in der Anfangsphase durchaus Einarbeitungsschwierigkeiten, die durch die neue Programmiersprache, aber auch durch ein neues Entwicklungsparadigma (Objekttechnologie) und durch eine neue Entwicklungsumgebung verursacht wurden. Diesen typischen Anfangsschwierigkeiten wurde aber frühzeitig durch gezielte Schulungsmaßnahmen und durch ein begleitendes Training-on-the-Job begegnet. Sie haben sich innerhalb von 3-6 Monaten weitgehend abbauen lassen.
- die Plattformunabhängigkeit: sie ermöglichte einen Entwicklungsbetrieb, der weitgehend von administrativen Maßnahmen freigehalten werden konnte. Die Anwendung wurde auf NT entwickelt und getestet, unter Verwendung von Werkzeugen, die auf anderen Plattformen nicht zur Verfügung stehen. Nach Abschluß der Entwicklung wurde ein Teil der Anwendung (ohne Neukompilation) auf AIX installiert. Damit lief dieser Teil auf der gleichen Maschine wie der Datenbankserver, was i.a. große Performancevorteile mit sich bringt. Die Netzwerktransparenz, die bei dieser Plattform-Migration implizit ausgenutzt wurde, wurde einerseits durch die Anwendungsarchitektur und andererseits durch einen CORBA-ORB [5] gewährleistet. Durch diese

Plattformunabhängigkeit von Java wird aber auch ein transparenter Austausch der zugrunde liegenden Hardware möglich. Dies gewährt eine zusätzliche Flexibilität bei Investitionsentscheidungen in diesem Bereich. Es sei deswegen nochmals hervorgehoben, daß im Rahmen des Projektes keinerlei Plattformabhängigkeiten der entwickelten Anwendung festgestellt wurden.
- die weitgehende Unterstützung von Java durch verschiedene Hersteller: sie spielte in diesem speziellen Projekt eine große Rolle, weil auch die Firma Bull für ihre Hostsysteme eine virtuelle Maschine für Java angekündigt hat. Damit wird es möglich, in Java entwickelte Programme auch direkt auf dem existierenden Hostsystem laufen zu lassen. Insofern eröffnet Java auch eine interessante Perspektive für die traditionelle Anwendungsentwicklung auf dem Host.

3 Übersicht über die Gesamtarchitektur des entwickelten Systems

Auch wenn die Gesamtarchitektur des entwickelten Systems nicht im Zentrum dieses Artikels steht, trägt eine kurze Beschreibung dieses Kontextes sicherlich zum Verständnis bei. Als Basis für eine stabile und offene Netzinfrastruktur wurde eine CORBA-Architektur [5] gewählt. Auf dieser Grundlage wurde ein verteiltes Model-View-Controller Pattern implementiert, das eine klare Trennung von Geschäfts-, Datenzugriffs- und Oberflächenlogik ermöglicht. Die Modellschicht wurde zu diesem Zweck nochmals in eine Geschäfts- und in eine Datenzugriffsschicht unterteilt.

- Die Geschäftslogik wurde sehr detailliert im o.g. CASE-Tool modelliert und über einen selbst entwickelten Generator in Form von Templates in die Implementierung eingebracht. Die Vervollständigung dieser Templates lieferte dann die Implementierung der Geschäftsschicht.
- In der Datenzugriffsschicht wurde eine objektrelationale Abbildung über ein existierendes Framework realisiert. Die dafür benötigten Klassen wurden ebenfalls durch den Generator aus dem Objektmodell im Innovator erzeugt.
- Die Oberflächen wurden unter Verwendung von standardisierten GUI-Bibliotheken (Swing 1.0, vergl. [6] entwickelt und lediglich an die speziellen Bedürfnisse des Projektes angepaßt. In diesem Bereich wurde schon während der Analyse auf die Bildung von graphischen Komponenten geachtet, die in verschiedenen Kontexten die Darstellung von Geschäftsobjekten übernehmen können. Diese Komponenten wurden dann mit o.g. GUI-Builder zu komplexen, situationsangepaßten Views zusammengesetzt. Die Oberflächenlogik wurde, soweit sie über reine Formatprüfungen hinausging, in Controller-Klassen implementiert.

Die Geschäftslogik wurde in Anlehnung an den Enterprise JavaBean (EJB) Standard [2] modelliert und implementiert. Wiewohl zum Zeitpunkt dieser Entscheidung der EJB Standard noch nicht sehr verbreitet war, ließ sich doch schon

ein starker Trend in dieser Richtung beobachten. Leider enthält der Standard in seiner Version 1.0 noch nicht alle Festlegungen, die für eine solide Infrastruktur benötigt werden. Im Projekt machten sich insbesondere die Abwesenheit einer Spezifikation für die Themen "Navigation zwischen Entitäten" und "ereignisgesteuerte Kommunikation zwischen Server und Client" bemerkbar. Aus diesem Grunde wurde im Projekt auf einen Einsatz eines EJB Server verzichtet. Die Migration auf einen solchen Server wurde jedoch vorbereitet und sollte bei Bedarf, unter weitgehender Beibehaltung der Implementierungen, möglich sein.

Natürlich ließe sich noch sehr viel mehr über die Gesamtarchitektur sagen. Dieser Teil wird hier jedoch zugunsten einer Darstellung der Legacy-Anbindung zurückgestellt.

4 Das Persistenz-Framework der Datenzugriffsschicht

Unterhalb der Geschäftsobjekte kam ein Persistenz-Framework zum Einsatz, mit dessen Hilfe diese Objekte auf relationale Tabellen abgebildet wurde. Nach ersten, nicht sehr erfolgreichen Versuchen mit einem neu auf den Markt gekommenen Framework, das lediglich als Betaversion zur Verfügung stand, wurde im Projekt auf ein anderes Persistenz-Framework zurückgegriffen, das mit VisualAge ausgeliefert wurde. Hierbei handelte es sich dabei um VisualAge Persistence (VAP). Gründe für diese Auswahl waren

- eine starke Orientierung von VAP am EJB Standard: in diesem Standard sahen alle Projektbeteiligten eine strategische Perspektive für die Anwendungsentwicklung in Java.
- die Verfügbarkeit im Source-Code: dadurch war eine komfortable Entwicklung mit Unterstützung durch einen Debugger möglich. Dies war insbesondere bei der Generierung der Datenzugriffsklassen aus dem Objektmodell eine große Hilfe.
- die enge Integration mit VisualAge: in Vorwegnahme einer graphischen Programmierung von Transaktionen spielte auch dieser Punkt eine Rolle. Im weiteren Verlauf des Projektes wurde diese Möglichkeit dann jedoch nicht eingesetzt, da Transaktionen fest mit Prozeßobjekten verknüpft wurden.

Die Toolunterstützung von VAP, die eine Modellierung von Entitäten und Datenbankschemata gestattet und die eng mit VisualAge integriert ist, spielte bei der Bewertung des Frameworks eine eher untergeordnete Rolle; es war ohnehin vorgesehen, die Generierung der Datenzugriffsschicht direkt aus dem Objektmodell vorzunehmen. Eine Verwendung der Tools von VAP wäre allerdings, wenn sie gewünscht gewesen wäre, mit Schwierigkeiten verbunden gewesen, da aus den VAP-Tools heraus kein Zugriff auf das Objektmodell des CASE-Tools möglich war.

5 Die Schnittstelle zur relationalen und zur hierarchischen Datenbank

Das Persistenz-Framework VAP verwendet das standardisierte Interface Java Database Connectivity (JDBC) [3] für den Zugriff auf relationale Datenbanken. Das Interface wird typischerweise von einem kommerziell verfügbaren Treiber implementiert, der dem Persistenz-Framework zur Verfügung gestellt wird. Danach verwendet das Framework den Treiber implizit bei jedem Zugriff auf die relationale Datenbank.

In Bereich relationale Datenbanken wurde im Laufe des Projektes auf zwei Ebenen variiert. Zum einen kamen zwischenzeitlich zu Testzwecken auch andere Datenbanken als Informix zum Einsatz, zum anderen wurde dieselbe Datenbank durch verschiedene JDBC-Treiber angesprochen. Hierbei mußten einige Inkompatibilitäten zwischen verschiedenen Treibern und Datenbanken festgestellt werden. So reagierte etwa eine der verwendeten Datenbanken unterschiedlich, je nachdem, über welchen JDBC-Treiber sie angesprochen wurde. Daneben war auch die Syntax vieler SQL-Abfragen Datenbank-spezifisch.

Insgesamt erwies sich die relationale Datenbankanbindung als nicht sehr portabel. Nach unserer Erfahrung läßt es sich im Normalfall nicht vermeiden, bei Austausch einer der Komponenten dieser Anbindung auch eine Anpassung des Codes der Datenzugriffsschicht vorzunehmen. Diese kann allerdings in manchen Fällen auf ein Neuformulieren der SQL-Statements beschränkt sein.

VAP unterstützt neben JDBC aber auch andere Speichermechanismen, z.B. eine Integration von "Function Call Backends", für die auf einer etwas höheren Abstraktionsebene eine Anpassung des Frameworks vorgenommen werden kann. In diesem Fall werden Daten über direkte Funktionsaufrufe von einem Legacy-System abgefragt. Dieser Ansatz birgt aus Sicht des Projektes zwei Nachteile:

- Die dazu benötigten Schnittstellen des Persistenz-Frameworks sind, im Gegensatz zu JDBC, nicht standardisiert. Damit ist die in diesen Bereich des Frameworks investierte Arbeit verloren, falls das Persistenz-Framework ausgetauscht werden sollte. Dies wäre insbesondere bei einem Einsatz eines EJB Servers der Fall.
- Beim Zugriff auf Daten in der Legacy-Datenbank muß ein anderer Mechanismus verwendet werden als beim Zugriff auf relationalen Daten. Dies erschwert die Arbeit des Codegenerators, der auf Informationen aus dem Objektmodell aufsetzt.

Im Projekt wurde deshalb die Entscheidung getroffen, auch auf die Daten der Legacy-Datenbank über einen passenden JDBC-Treiber zuzugreifen. Damit werden die beiden o.g. Nachteile umgangen. Dieser Ansatz vereinfacht insbesondere die zu einem späteren Zeitpunkt eventuell vorzunehmende Migration nach EJB, da ein JDBC-Treiber sich prinzipiell auch unter einem EJB Server betreiben läßt. Allerdings bestehen in der aktuellen Implementierung des JDBC-Treibers gewisse Einschränkungen, die weiter unten erläutert werden. Es muß deswegen vor dem Einsatz unter einem EJB Server zunächst geprüft werden, ob eine dieser Einschränkungen einen effektiven Betrieb verhindert.

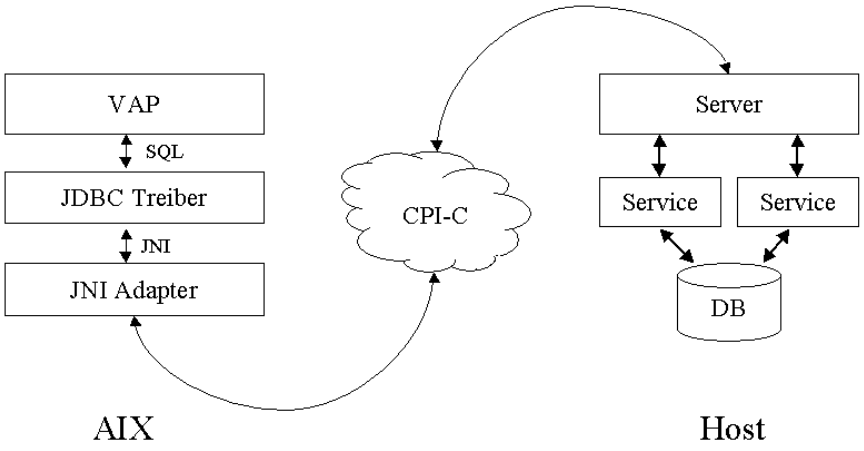

Abbildung1. Übersicht über die Host-Anbindung

6 Übersicht über die Host-Anbindung[1]

Für die Anbindung des Host-Systems wurde im Projekt ein spezieller JDBC-Treiber entwickelt, der den Zugriff auf die Legacy-Daten durch eine SQL-Schnittstelle ermöglicht. Die Details der Funktionalität des Treibers werden in den nachfolgenden Abschnitten erläutert. Hier soll zunächst nur ein kurzer Überblick über die verwendeten Komponenten gegeben werden.

Auf technischer Ebene kommt für die Kommunikation mit dem Host die Bibliothek CPI-C zum Einsatz, die die für den Zugriff erforderlichen Netzprotokolle kapselt. Da diese Bibliothek in C implementiert ist, wurde sie unter Verwendung des Java Native Interface (JNI) über eine dünne Adapterschicht in den JDBC-Treiber eingebunden. CPI-C übernimmt für den JDBC-Treiber die Session-Verwaltung und startet automatisch das erforderliche Transaktionsprogramm (TPR) auf dem Host. Die Aufrufparameter werden dabei dem TPR transparent zur Verfügung gestellt. Auf Host-Seite ist das empfangende TPR zunächst ein generisches Programm, der sog. Server, das in COBOL implementiert wurde. Es analysiert die Aufrufparameter und leitet sie dann an verschiedene, stärker spezialisierte Services weiter, von denen einige ebenfalls im Rahmen des Projektes implementiert wurden. Diese Services sind für den Zugriff auf die Legacy-Daten verantwortlich; sie führen in Kenntnis der genauen Struktur der Legacy-Datenbank die angefragten Operationen durch und geben eventuelle Ergebnisse an den Server zurück. Dieser reicht die Ergebnisse ohne weitere Verarbeitung zurück an die CPI-C-Bibliothek und damit an den JDBC-Treiber. Um die Implementierungsarbeiten auf dem Host zu reduzieren, werden even-

[1] Die Architektur auf Host-Seite wurde von Richard Stubbs entworfen.

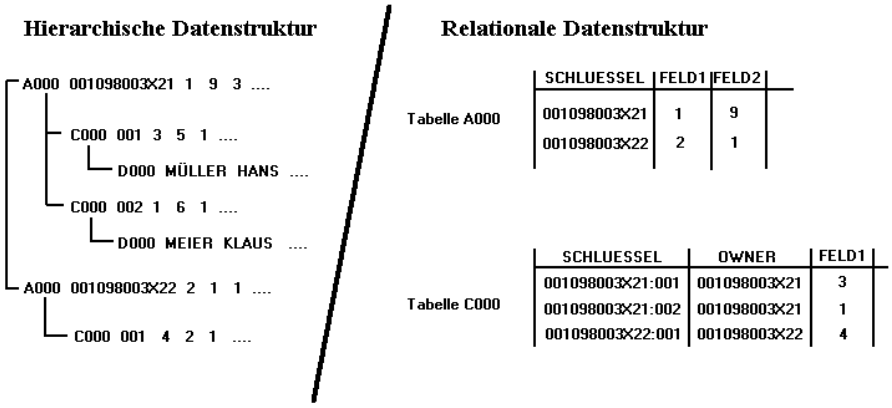

Abbildung2. Die Umsetzung von hierarchischen in relationale Daten

tuelle Ergebnisse auf Host-Seite kaum aufbereitet, sondern in ihrer bestehenden hierarchischen Struktur an den JDBC-Treiber zurückgegeben. Die Hierarchie der Datensätze wird dabei typischerweise durch eine gewisse Reihenfolge dargestellt.

Eine der wesentlichen Stärken dieser Unterteilung in Server und Services ist es, daß der Server jederzeit um neue Services erweitert werden kann. Letztere können je nach Bedarf wie traditionelle COBOL-Programme entwickelt werden und dann in den bestehenden Server integriert werden. Dadurch ist es ohne großen Aufwand möglich, weitere Zugriffsmodule für spezialisierte Auswertungen der Datenbank etc. zur Verfügung zu stellen.

Ausgehend von dieser grundsätzlichen Architektur wird in den folgenden Abschnitten die genaue Funktionsweise des JDBC-Treibers erläutert.

7 Die Umwandlung von hierarchischen in relationale Daten

Die Verwendung von JDBC zum Zugriff auf die Legacy-Datenbank impliziert, daß eine Umsetzung der Daten von einer hierarchischen in eine relationale Struktur erfolgen muß. Auf dieser relationalen Struktur müssen weiterhin SQL-Abfragen unterstützt werden, die es erlauben, die Daten unter relevanten Aspekten auszuwerten. Diese Funktionalität scheint auf den ersten Blick relativ aufwendig zu sein. Eine genauere Analyse zeigt jedoch, daß sie sich (unter gewissen Einschränkungen) durchaus realisieren läßt.

Die Umsetzung der Struktur erfolgt zur Laufzeit innerhalb des JDBC-Treibers. Dieser erhält als Ergebnis einer Abfrage vom Host Daten in einer hierarchischen Form, d.h. in einer bestimmten Reihenfolge. Die Umsetzung der Daten in eine relationale Struktur erfolgt nun unter Berücksichtigung dieser Reihenfolge,

indem alle Datensätze eines festen Typs in passenden (Hauptspeicher-)Tabellen abgelegt werden und dabei mit einem künstlich erzeugten Primärschlüssel versehen werden. Die Hierarchie der Datensätze wird während dieser Umsetzung bewahrt, denn untergeordnete Datensätze speichern als Zusatzinformation neben dem eigenen Primärschlüssel auch den Primärschlüssel ihres übergeordneten Datensatz. Damit kann im weiteren durch einfache SQL-SELECT- Abfragen herausgefunden werden, welche der Datensätze in den verschiedenen Tabellen im Sinne der ursprünglichen Hierarchie zueinander in Beziehung stehen.

Bei der Übertragung der Daten vom Host werden auch gewisse Metadaten in einem proprietären Format übermittelt, wie etwa die Feldnamen eines Datensatzes oder die Typen der einzelnen Felder. Diese Typinformationen erlauben es, die Daten korrekt zu interpretieren und mit größtmöglicher Akkuratesse in entsprechende Java-Datentypen umzuwandeln.

8 Die Erzeugung der Primärschlüssel

Die Erzeugung des Primärschlüssels erfolgt während der Umsetzung unter Verwendung (einer Gruppe) von fachlichen Attributen der Datensätze. Der JDBC-Treiber erlaubt es, für jeden Datensatz Attribute zu konfigurieren, die diesen Datensatz auf einer gegebenen Hierarchieebene eindeutig identifizieren. Man beachte, daß diese Attribute normalerweise nicht hinreichend sein werden, um den Datensatz absolut, d.h. eindeutig in der gesamten Hierarchie, zu identifizieren. Um dies zu erreichen muß in hierarchischen Datenbanken vielmehr auch der Kontext des Datensatzes eindeutig identifiziert werden, d.h. der übergeordnete Datensatz. Ein Beispiel für diese Situation ist etwa der (relative) Primärschlüssel eines Vertragsstand-Datensatzes, der lediglich aus einer laufenden Nummer bestehen kann. Diese laufende Nummer ist jeweils eindeutig in Bezug auf einen gegebenen Vertrag, der den Kontext für den Vertragsstand definiert. Unterhalb eines anderen Vertrages kann es aber durchaus einen weiteren Vertragsstand mit derselben laufenden Nummer geben. In diesem Beispiel besteht eine absolute Identifikation eines Vertragsstandes also aus der Verknüpfung eines Primärschlüssels eines Vertrages mit einem Primärschlüssel eines Vertragsstandes. Es ist möglich, daß auch der Primärschlüssel des Vertrags in ähnlicher Art und Weise zusammengesetzt werden muß. Es gibt aber in jeder hierarchischen Datenbank eine Wurzel, für die dies nicht erforderlich ist und die ohne einen Kontext absolut adressiert werden kann.

Aus diesem Grund erfolgt die Erzeugung der Primärschlüssel innerhalb des JDBC-Treibers rekursiv: zunächst werden die Primärschlüssel der Wurzel-Datensätze unter Berücksichtigung der Konfigurationsinformationen ermittelt. Danach werden alle abhängigen Datensätze in fortschreitender Hierarchietiefe bearbeitet. Dabei werden auf jeder Ebene der Hierarchie die relativen Primärschlüssel, die sich aus der Konfiguration ergeben, durch den zugehörigen Primärschlüssel der vorhergehenden Hierarchiestufe qualifiziert.

9 Die Auswertung der relationalen Daten

Typischerweise werden nicht alle Felder eines Datensatzes benötigt, um die Attribute eines Objektes zu belegen. In einigen Fällen kann es sogar erforderlich sein, Felder aus verschiedenen Datensätzen in einem einzigen Objekt zusammenzufassen. Um diese Funktionalitäten zur Verfügung zu stellen, müssen die Daten nach ihrer Umwandlung in Tabellen noch weiter gefiltert werden.

Der JDBC-Treiber implementiert zu diesem Zweck einen einfachen SQL-Interpreter. Anhand einer SQL-Abfrage werden sowohl die Felder als auch die Bedingungen identifiziert, denen die Ergebnismenge genügen soll. Die so beschriebenen Inhalte werden dann in den vom Host erhaltenen Daten gesucht und dem Benutzer in Form einer separaten Tabelle zur Verfügung gestellt.

Der JDBC-Treiber unterstützt dabei SELECT-Abfragen für den Lesezugriff und UPDATE-, INSERT- und DELETE-Abfragen für den Schreibzugriff. Die syntaktischen Mittel bei der Formulierung der Abfragen umfassen neben den normalen Feld-, Tabellen- und Aliaslisten auch eingeschränkte WHERE Klauseln, in denen atomare Formeln durch logische Konjunktion verknüpft werden können. Eine Disjunktion oder eine Negation wird von der gegenwärtigen Implementierung seitens des JDBC-Treibers nicht unterstützt. Atomare Formeln sind in diesem Zusammenhang Vergleiche (bzgl. $=$, $<$, $\leq$, $>$ und $\geq$) zwischen Feldnamen und Konstanten. Es ist zulässig, daß in einer atomaren Formel zwei Feldnamen miteinander verglichen werden. Bei diesen Vergleichen werden natürlich die Datentypen der Felder berücksichtigt. Dies ist erforderlich, weil z.B. ein String-Vergleich der Werte '001' und '0001' ein anderes Ergebnis liefert als ein Integer-Vergleich der gleichen Werte.

Eine zulässige SQL-Abfrage könnte also z.B. die folgende Form haben:

```
SELECT T1.FELD1, T1.FELD2, T2.FELD3
FROM TABELLE1 T1, TABELLE2 T2
WHERE T1.SCHLUESSEL='001098003X21' AND
T2.OWNER=T1.SCHLUESSEL
```

Hiermit werden die Felder FELD1 und FELD2 der Tabelle TABELLE1 und das Feld FELD3 der Tabelle TABELLE2 für denjenigen Datensatz in TABELLE1 selektiert, dessen Primärschlüssel als String mit dem Wert 001098003X21 übereinstimmt. Der zugehörige Satz aus TABELLE2 wird dabei über die zweite Bedingung der WHERE-Klausel definiert: sie sagt aus, daß nur solche Datensätze der TABELLE2 bei der Auswahl berücksichtigt werden, deren übergeordneter Datensatz gerade der o.g. Datensatz ist. Man beachte, daß dies genau der Navigation über die Hierarchie der zugrundeliegenden Legacy-Datenbank entspricht. Damit sind auch komplexere Anfragen möglich, die den Legacy-Kontext eines Datensatzes berücksichtigen.

Um die selektierten Daten in Objekte umwandeln zu können, müssen sie dem Persistenz-Framework VAP in Form eines java.sql.ResultSet zur Verfügung gestellt werden. Während der Aufbereitung der Daten in dieser Form werden insbesondere auch die SQL-Metadaten erzeugt, die vom Persistenz-Framework für

den Zugriff auf die Daten benötigt werden. Diese werden aus den (proprietären) Metadaten ermittelt, die vom Host übertragen wurden, soweit diese die benötigten Informationen bereitstellen.

Beim Zugriff auf die Daten des ResultSet erfolgt u.a. auch eine spezielle Verwaltung des SQL-Wertes NULL gemäß der JDBC-Spezifikation. Allerdings enthalten die ursprünglichen Legacy-Daten diesen speziellen Wert naturgemäß nicht. Des weiteren führt ein Schreibversuch über den JDBC-Treiber mit einem Attributwert NULL zu einer Fehlermeldung, da dieser Wert auf keinen bekannten Host-Typ abgebildet werden kann. Deswegen kann dieser Wert de facto nie beim Zugriff auf die Daten des ResultSet auftreten.

10 Unterstützung für Stored Procedures

Neben dem reinen Datenzugriff wurde im Rahmen des Pilotprojektes auch ein Zugriff auf Funktionen des Hosts implementiert. Für einen Prototyp wurde ein existierendes FORTRAN-Programm betrachtet. Der Zugriff auf dieses Programm folgt den gleichen technischen Prinzipien wie der Datenzugriff: zunächst wurde das FORTRAN-Programm über einen speziellen Service in die generische Serverarchitektur auf dem Host eingebunden. Dieser Service wurde dann vom JDBC-Treiber über eine entsprechende Abfrage angesprochen.

Abgesehen von einem Kommunikationsproblem zwischen dem in COBOL geschriebenen Service und dem FORTRAN-Programm ließ sich diese Einbindung auf der Host-Seite ohne nennenswerte Schwierigkeiten umsetzen. Auf Seiten des JDBC-Treibers waren allerdings einige Erweiterungen der Funktionalität erforderlich, um auch dieses Szenario abdecken zu können. Hierzu zählte vor allem eine neue Query-Syntax. Um einerseits die erforderlichen Parameter über die SQL-Schnittstelle in den JDBC-Treiber hineingeben zu können und andererseits die Ergebnisse nach der Abfrage dem Persistenz-Framework übergeben zu können, wurden "EXECUTE PROCEDURE"-Queries definiert. Diese lassen sich als sog. "Custom Queries" in VAP einbinden. Zur Laufzeit erfolgt ein Aufruf einer solchen Query in enger Anlehnung an die EJB-Architektur durch einen Aufruf einer geeigneten find-Methode auf einem Home-Objekt. Die Parameter, die für die Berechnung auf dem Host benötigt werden, werden dieser Methode in Form von geeigneten Objekten mitgegeben und dann innerhalb der Methode in eine SQL-Abfrage umgewandelt. Diese Abfrage wird dem JDBC-Treiber übergeben, der sie nach dem normalen Verfahren zum Host überträgt.

Bei der Rückübertragung der Daten müssen die Ergebnisse der Berechnung für das Persistenz-Framework verfügbar gemacht werden. Bei einem Aufruf einer StoredProcedure in einer relationalen Datenbank geschieht dies normalerweise über sog. Hostvariablen, die nach dem Aufruf der Prozedur mit den Ergebnissen belegt werden. Diese Infrastruktur steht jedoch innerhalb des Persistenz-Frameworks nicht zur Verfügung. Deswegen werden die Ergebnisse der Berechnung in Form eines java.sql.ResultSet vom JDBC-Treiber aufbereitet. VAP kann dann auf diese Darstellung zugreifen und die Ergebnisse der Berechnung mit den üblichen Mechanismen in passende Objekte überführen. Diese werden dann für

die weitere Verwendung an den Aufrufer der find-Methode zurückgegeben. Man beachte, daß damit eine reine Objekt-Schnittstelle für den Zugriff auf Funktionen des Hosts besteht: sowohl die Parameter als auch die Ergebnisse der find-Methode sind, aus Sicht des Aufrufers, normale Objekte.

11 Performance

Die Performance des JDBC-Treibers beim Zugriff auf den Host war ein entscheidender Bewertungsfaktor für die vorgestellte Architektur. Gerade die skizzierte Zusatzfunktionalität des JDBC-Treibers, wie z.B. die Umwandlung von hierarchischen in relationale Daten, stellt einen Belastungsfaktor dar. Dazu kommen noch die Kosten für das Instantiieren von Objekten aus den Rohdaten, die die Gesamtperformance beeinträchtigen. Letztere treten allerdings natürlich auch bei relationalen Datenquellen auf und sind damit nicht spezifisch für die Einbindung der Legacy-Daten. Aus diesem Grund werden diese Kosten bei der folgenden Betrachtung nicht berücksichtigt.

Grundsätzlich ist die Architektur nur bedingt dazu geeignet, große Datenmengen vom Host abzugreifen. Hierbei würde zwar die Zeit, die für die Umwandlung in relationale Daten erforderlich ist, höchstens proportional zur Datenmenge bleiben; jedoch könnte die danach erforderliche Selektion von Daten, bei gleichzeitiger Auswertung von vorgegebenen logischen Bedingungen, zu einem Performance-Engpaß werden. Ob dies bei einer konkreten Abfrage tatsächlich der Fall ist, hängt stark von den auszuwertenden Bedingungen ab.

Nun ist aber der Zugriff auf große Datenmengen für ein Persistenz-Framework eher untypisch. Benötigt werden im Normalfall eher die Daten für einige wenige Objekte, die in der Folge der Abfrage dann instantiiert werden. Für diese Art von Zugriff ist die oben beschriebene Architektur relativ gut geeignet, da die Selektion der Daten bei kleinen Datenmengen entsprechend schnell durchlaufen werden kann.

Die in den Abbildungen 3 und 4 dargestellten Werte wurden auf einem P133 mit 128MB RAM unter NT 4.0 gemessen. Es wurden vier verschiedene VMs betrachtet, nämlich Sun's und IBM's JDK1.1.7 sowie die VM des Sun JDK 1.2 in der Classic- und in der HotSpot-Version. Es wurde ohne Netzverbindung zum Host-System gemessen, um eine Bewertung der reinen Performance des JDBC-Treibers vornehmen zu können. Der Testdatenbestand wurden deswegen im Host-Format aus einer Datei in den JDBC-Treiber eingelesen. In der Abbildung 3 werden die durchschnittlichen Zeiten für den Zugriff auf einen Datenbestand von ca. 9 KB bzw. ca. 90 KB dargestellt. 9 KB Daten entsprechen einem einfachen Versicherungsvertrag mit drei Vertragsständen und einigen weiteren Datensätzen; 90 KB Daten entsprechen 40 Verträgen mit jeweils 3 Vertragsständen und weiteren Datensätzen. Man beachte, daß eine um den Faktor 10 größere Datenmenge ausreicht, um das Vierzigfache an Information zu übertragen. Der Grund für diese Komprimierung liegt in der Homogenität der Ergebnismenge. Sie erlaubt es, die Metadaten für alle 40 Datensätze nur einmal in die Ergebnismenge einzufügen und dadurch die Datenmenge zu reduzieren.

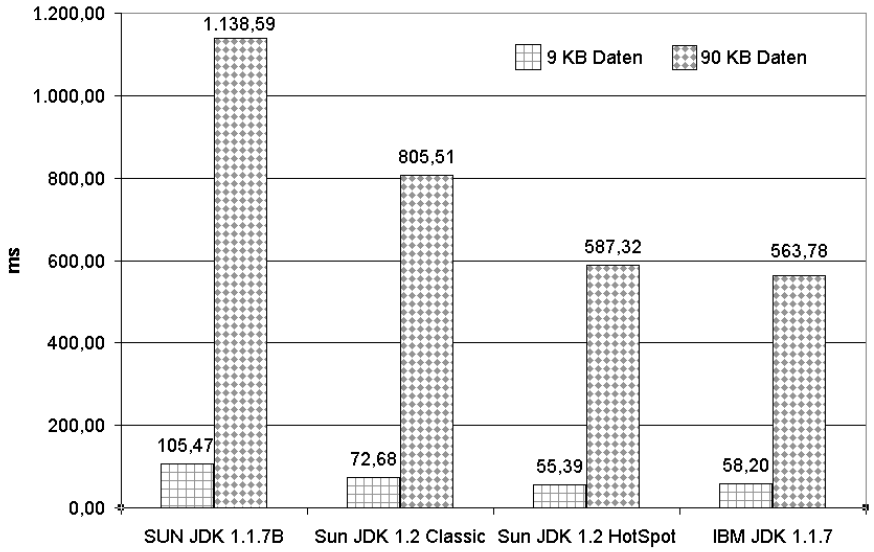

Abbildung3. Die Performance des JDBC-Treibers in verschiedenen JVMs auf NT 4.0 (Durchschnittliche Zugriffsdauer auf 9KB bzw 90 KB Daten bei 10000 Iterationen)

In jeder Iteration wurden die Daten aus der Testdatei eingelesen, in relationale Daten umgewandelt und folgende Abfrage wurde durchgeführt:

```
SELECT T1.FELD1, T1.FELD2, T1.SCHLUESSEL FROM TABELLE T1
WHERE T1.SCHLUESSEL = '001098003X21'
```

Die Testdaten waren so gestaltet, daß alle Datensätze der Tabelle TABELLE1 die angegebene Bedingung erfüllten. Das Ergebnis der Selektion wurde als ResultSet aufbereitet und dem Benutzer zurückgegeben.

Die angegebenen Werte wurden als Durchschnitt über 10000 Iterationen ermittelt. Die Schwankungen innerhalb der einzelnen Iterationen waren nach der Startphase gering; es ließ sich allerdings in der Startphase sehr deutlich das unterschiedliche Kompilationsverhalten der verschiedenen VMs beobachten, wie in Abbildung 4 dargestellt wird. So benötigten die adaptiv kompilierenden VMs[2] während der ersten hundert Iterationen zwischen 100 ms und 400 ms pro Zugriff (auf 9 KB Daten) bei relativ starken Schwankungen. Diese Schwankungen sind durch die Hintergrundtätigkeiten der VM zu erklären, die den Code

[2] IBM JDK 1.1.7 und Sun JDK 1.2 Hotspot; in Abbildung 4 werden wegen der besseren Vergleichbarkeit die Werte für Hotspot angegeben.

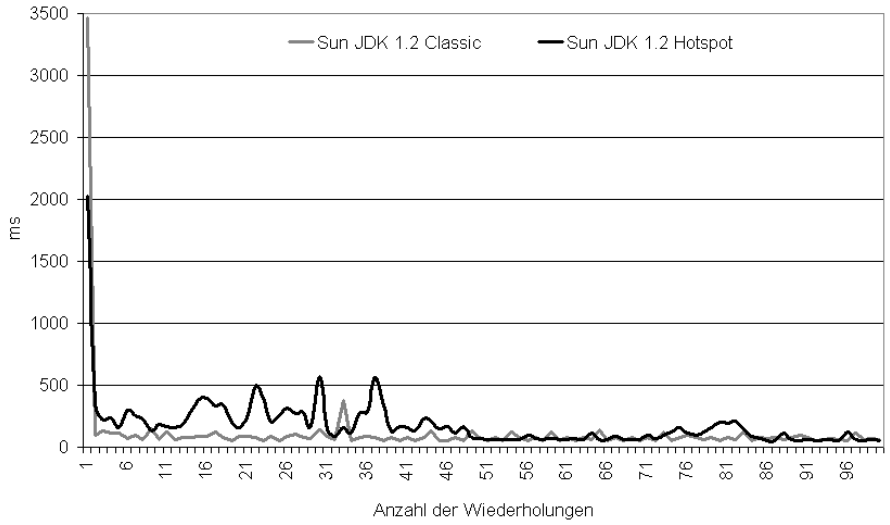

Abbildung4. Das Startup-Verhalten der Classic- und der Hotspot-Version des JDK 1.2
(beim Zugriff auf 9 KB Daten)

auf Performance-Engpässe hin analysiert. Die erste Iteration lag durch Initialisierungsoperationen in diesem Falle sogar bei ca. 2000 ms. Erst nachdem die performancekritischen Teile des Codes identifiziert worden waren, erreichten die adaptiven VMs die in Abbildung 3 angegebenen Werte. Dies war nach ca. 100 Iterationen der Fall.

Die mit einem Just-In-Time Compiler (JIT) arbeitenden VMs[3] wiesen hingegen eine signifikante Verzögerung in der ersten Iteration auf; hierfür benötigten sie bis zu 3500 ms. Ab der zweiten Iteration erreichten sie dann jedoch stabil die in der Abbildungen 3 genannten Werte. Dieses Verhalten ist dadurch zu erklren, daß alle Klassen schon während der ersten Iteration kompiliert werden.

Beim Betrieb am Netz mit einer Online-Kommunikation mit dem Host ist eine Performance von ca. 500 ms (beim Lesezugriff auf 9 KB Daten) zu messen. Bei der Laufzeitumgebung handelt es sich in diesem Fall um das IBM JDK 1.1.6 auf AIX 4.1. Dieses JDK verfügt über einen integrierten JIT. In dieser Umgebung macht die Offline-Performance, die nicht exakt mit den oben dargestellten Werten übereinstimmt, ca. 15% der Online-Performance aus, d.h. 70-90 ms. Die-

[3] Sun JDK 1.1.7 und Sun JDK 1.2 Classic; in Abbildung 4 werden wegen der besseren Vergleichbarkeit die Werte für Classic angegeben.

ser Wert gilt jedoch erst nach dem Durchlaufen einer Anfangsphase, wie es bei JIT-Architekturen zu erwarten ist.

Um die Online-Performance weiter zu verbessern, wurde auf Seiten des JDBC-Treibers ein Cache-Mechanismus implementiert. Dabei werden innerhalb einer Transaktion alle vom Host abgefragten Daten in relationaler Form innerhalb des JDBC-Treibers zwischengespeichert. Damit ist es möglich, bei erneuter Anforderung der gleichen Daten innerhalb der gleichen Transaktion den Netzwerkzugriff zu vermeiden. Außerdem wird dadurch garantiert, daß einmal in der Transaktion gelesene Daten für die Dauer der Transaktion unverändert bleiben (Read-Stability). Mit dem Ende der Transaktion wird der Cache gelöscht.

12 Transaktionskoordination

Ein bisher noch offener Punkt in der Gesamtarchitektur zur Host-Anbindung ist die Transaktionskoordination zwischen dem Hostsystem und der Java-Anwendung. Die anzustrebende Lösung in diesem Bereich ist ein 2-Phase-Commit (2pc) [1], das eine volle Integration der Transaktionen in beiden System erlaubt. Die Umsetzung dieses Ansatzes ist z.Zt. jedoch noch mit Problemen behaftet, die in diesem Abschnitt näher erläutert werden.

Sowohl auf Seiten des Hosts als auch in der Java -Anwendung besteht grundsätzlich die Möglichkeit, transaktionsgesichert auf Daten zuzugreifen. Auf Host-Seite wird diese Funktionalität von einem traditionellen Transaktionsmonitor übernommen, der im wesentlichen kurze Transaktionen zuläßt. Eine Einbindung dieses Transaktionsmonitors in ein 2pc wird ab Mitte 99 möglich sein.

Auf Java-Seite bietet das Persistenz-Framework VAP seinen eigenen Transaktionsmechanismus an. Dieser verwendet zwar intern einen Zwei-Phasen-Mechanismus, um im ersten Durchlauf die Daten auf alle beteiligten Datenquellen zu verteilen und im zweiten Durchlauf die Transaktion durch ein Commit zu beenden. Trotzdem ist kein vollständiges 2pc implementiert, weil die JDBC-Version 1.22, auf der VAP basiert, diese Funktionalität noch nicht unterstützt. Dies ist erst in der sog. Standard-Erweiterung von JDBC 2.0 vorgesehen, die im Dezember 98 publiziert wurde. Die dafür zuständigen APIs werden im Java Transaction API (JTA) [4] zusammengefaßt.

Für die Implementierung eines 2pc muß also ein Persistenz-Framework (oder ein EJB-Server) eingesetzt werden, das JTA unterstützt. Ein erster EJB-Server, der diese Anforderung erfüllt, ist seit April 99 am Markt verfügbar. Daneben müssen die JDBC-Treiber für den Zugriff auf die relationalen Datenbanken an die JDBC-Version 2.0 und an JTA angepaßt werden. Mit einer Verfügbarkeit solcher Treiber ist im Laufe dieses Jahres zu rechnen. Schließlich ist auf Host-Seite noch eine Öffnung des Transaktionsmonitors für das 2pc-Protokoll erforderlich, die jedoch auch für dieses Jahr angekündigt wurde. Sobald diese Komponenten zur Verfügung stehen, kann ein vollständiges 2pc zwischen relationalen und hierarchischen Datenbanken implementiert werden.

In Ermangelung dieser Voraussetzungen wurde zunächst ein Schreibzugriff auf den Host im Rahmen von kurzen Transaktionen implementiert. Dabei wer-

den Daten auf dem Host transaktionsgesichert in einen sog. Bewegungspool geschrieben, der zu einem späteren Zeitpunkt durch eine Batch-Prozedur in den eigentlichen Datenbestand überführt wird. Der Zugriff auf den Bewegungspool erfolgt durch einen Service auf Host-Seite; am Ende des Aufrufes dieses Service wird versucht, die Transaktion, in der dieser Zugriff stattgefunden hat, durch ein Commit zu beenden. Falls dies gelingt, so wird der Aufruf des Service fehlerfrei beendet und die Daten stehen im Bewegungspool; andernfalls wird eine Fehlermeldung an den JDBC-Treiber zurückgegeben und von diesem durch eine SQLException weitergemeldet. In letzterem Fall werden die Daten nicht im Bewegungspool gespeichert.

Problematisch ist an diesem Ansatz die Tatsache, daß u.U. die Transaktion auf Seiten von VAP mit einem Rollback beendet werden kann und trotzdem Daten in den Bewegungspool geschrieben werden. Dieser Fall tritt dann ein, wenn im Rahmen des internen Zwei-Phasen-Mechanismus von VAP zuerst Daten auf den Host geschrieben werden und anschließend von einer anderen Datenquelle eine Fehlermeldung zurückgegeben wird, die ein Rollback im Transaktionsprozeß von VAP auslöst.

Dieses Problem kann in der beschriebenen Form während der ersten Phase des internen Zwei-Phasen-Mechanismus auftreten. Es läßt sich ohne ein echtes 2pc nicht vollständig vermeiden; es kann aber auf die zweite Phase dieses Mechanismus verschoben werden. Um dies zu erreichen, werden die Schreibzugriffe, d.h. von Daten aus INSERT-, UPDATE- und DELETE-Queries, vorübergehend innerhalb des JDBC-Treiber in einem Cache gespeichert. Damit werden beim Absetzen der entsprechenden Queries die Daten zunächst nicht zum Host übertragen. Während der zweiten Phase des internen Protokolls ist es dann möglich, darüber zu entscheiden, ob der Schreibzugriff tatsächlich stattfinden soll oder nicht. Zu diesem Zeitpunkt haben alle anderen Datenquellen schon die erste Phase des internen Protokolls durchlaufen und sollten keine Fehlermeldungen mehr auslösen. In der aktuelle Implementierung des JDBC-Treibers werden bei einem Rollback in der zweiten Phase die Schreibdaten des Cache verworfen; eine Übertragung der Daten zum Host findet nicht statt. Damit können auch solche Daten noch nachträglich verworfen werden, die aus Sicht von VAP schon an den Host übertragen wurden und das oben beschriebene Problem tritt nicht mehr auf. Nur bei einem Commit in der zweiten Phase werden die Daten auch tatsächlich auf den Host geschrieben.

Dieses Vorgehen dient neben der verbesserten Transaktionskoordinierung auch der Performancesteigerung. Es hat zur Folge, daß zunächst sämtliche Schreibzugriffe auf den Host, nach Services gruppiert, in einem Cache angesammelt werden. Während der Verarbeitung des Commit können dann alle Daten für einen gegebenen Service auf einen Schlag zum Host übertragen werden. Dadurch wird die Netzlast weiter reduziert. Ein Nachteil dieses Vorgehens ist, daß die als Update-Count bezeichneten Rückgabewerte von Schreibzugriffen nicht mehr akkurat ermittelt werden können, da der Schreibzugriff effektiv erst im Laufe der Behandlung eines Commit erfolgt. Ein weiterer Nachteil ist, daß nach wie vor ein Fehler während des Commit zu inkonsistenten Datenbeständen führen kann.

Danksagung

Dieser Artikel enthält Ergebnisse, die der Author zusammen mit Richard Stubbs (UIST Consultants Ltd.) erarbeitet hat. Herr Stubbs hat die Architektur auf der Host-Seite konzipiert. Dr. Jürgen Uhl (IBM Unternehmensberatung GmbH) hat schon in einem frühen Stadium an intensiven Diskussionen über die Konzeption der Host-Anbindung teilgenommen und viele kritische Punkte frühzeitig adressiert. Die technische Umsetzung der Gesamtkonzeption war nur durch die intensive Unterstützung von Susanne Vogt und Alfons Janßen (Barmenia Versicherungen) möglich.

Literatur

1. Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 oder ISBN 1 872630 24 3
2. Enterprise JavaBeans (EJB), http://java.sun.com/products/ejb
3. Java Database Connectivity (JDBC), http://java.sun.com/products/jdbc
4. Java Transaction API (JTA), http://java.sun.com/products/jta
5. Object Management Group (OMG), http://www.omg.org
6. Swing Connection (Swing), http://java.sun.com/products/jfc