Integration Management

A Virtualization Architecture for Adapter Technologies

Von der Fakultät Informatik, Elektrotechnik und Informationstechnik
der Universität Stuttgart
zur Erlangung der Würde eines
Doktors der Naturwissenschaften (Dr. rer. nat.)
genehmigte Abhandlung

Vorgelegt von

Ralf Wagner

aus Stuttgart

Hauptberichter:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Mitberichter:	Prof. Dr. rer. nat. Frank Leymann
Tag der mündlichen Prüfung:	28.01.2015

Institut für Parallele und Verteilte Systeme (IPVS)
der Universität Stuttgart

2015

Abstract

There is an abundance of integration technologies and there still is no means to systematically deal with them. This leads to increasing complexity in IT environments and also to increasing integration efforts and IT costs. *Integration management* (IM) provides a means of systematically dealing with integration technologies. It abstracts from integration technologies so that software development is shielded from integration tasks. The achieved integration independence significantly alleviates maintenance and evolution of IT environments and reduces the overall complexity and costs of IT landscapes.

We designed and realized an IM system, the *virtualization tier* (VT), that allows to reuse adapters of different integration technologies so that costly development of new adapters from scratch can be avoided. The VT achieves integration independence by means of a global access layer that supports transparent processing independent of the access style chosen by a client system and independent of the integration technology used in the VT to access a remote system. A client system can access any remote system by means of the VT if there is a suitable adapter deployed in the VT. The integration tasks that are related with accessing remote systems can be completely encapsulated by means of the VT so that a software developer can concentrate on the development of the core application logic.

Deployment and administration tasks in the VT are handled by different IT roles, i.e. *VT object deployer* and *adapter deployer*, so that the complex use of integration technologies becomes practically manageable. The VT also enables different architecture patterns to realize systematic integration solutions based on IM technology. The VT-based architecture patterns reuse parts of integration technologies and provide for significantly less complex integration solutions than conventional integration solutions do. Even SOA-based applications can benefit from the VT by using the VT as a global ESB. IM technology can thereby become a critical driver of Web service infrastructures. Finally, our performance evaluation of the VT shows that IM technology can work efficiently and it shows that optimization effects can even increase performance of IT scenarios.

Zusammenfassung

Einleitung

Die Aufgaben und Prozesse eines Unternehmens sind immer stärker mit seinen Softwaresystemen verbunden. Aus diesem Grund werden Softwaresysteme auch immer stärker miteinander verknüpft und deshalb ist die Integration von Softwaresystemen ein wichtiger Baustein für ein modernes und wettbewerbsfähiges Unternehmen. Die Problematik bei der Integration von Softwaresystemen besteht darin, dass sie sehr verschieden sein können, insbesondere aufgrund unterschiedlicher Technologien. Dadurch entstehen Inkompatibilitäten zwischen den Softwaresystemen, die eine direkte Interoperabilität verhindern. Hier setzen Integrationstechnologien an, die die technischen Heterogenitäten und Inkompatibilitäten von Softwaresystemen überbrücken können. In diesem Zusammenhang tauchen verschiedene Begriffe auf:

- **Middlewaretechnologie:** Eine Middlewaretechnologie spezifiziert eine Architektur, die verschiedenartige Softwaresysteme – die Remotesysteme – integrieren kann und die anderen Softwaresystemen – den Clientsystemen – einen einheitlichen Zugriff auf die Remotesysteme ermöglicht.

- **Middlewaresystem:** Ein Middlewaresystem ist ein Softwaresystem, das auf Middlewaretechnologie basiert. Beispiele von Middlewaresystemen sind Java EE Anwendungsserver, föderierte Datenbanksysteme und Message Broker.

- **Adapter:** Ein Adapter – auch bekannt als Konnektor, Wrapper, Gateway o.ä. – ist ein Softwareartefakt, das von einem Softwaresystem, z.B. einem Middlewaresystem, dazu verwendet wird um auf ein Remotesystem zuzugreifen. Beispiele von Adaptern sind J2EE-Konnektoren, SQL-Wrapper oder Message-Broker-Adapter.

- **Adaptertechnologie:** Eine Adaptertechnologie spezifiziert eine Architektur oder einen Teil einer Architektur zur Überbrückung technologischer Inkompatibilitäten zwischen Softwaresystemen unter Verwendung von Adaptern. Beispiele von Adaptertechnologien sind die J2EE Connector Architecture [Sun03] oder SQL/MED [SQL08].

- **Integrationstechnologie:** Eine Integrationstechnologie spezifiziert eine Architektur, die eine Middlewaretechnologie und eine von der Middlewaretechnologie verwendeten Adaptertechnologie umfasst. Beispiele von Integrations-

technologien sind Java EE Anwendungsservertechnologie, föderierte Datenbanktechnologie und Message-Broker-Technologie.

Den Kern zur Überwindung technologischer Inkompatibilitäten stellen Adapter dar. Abbildung 1 zeigt das grundlegende Prinzip eines Adapters. Es gibt zwei Interaktionskategorien: das Ausführen einer Anfrage und das Empfangen einer Antwort. Kombinationen aus diesen beiden Kategorien resultieren in Interaktionsmustern, wie sie beispielsweise auch in WSDL [CMRW07] oder REST [Fie00] zu finden sind. Das bekannteste Interaktionsmuster ist das Anfrage-Antwort-Muster, das aus einer Anfrage und einer darauf folgenden Antwort besteht.

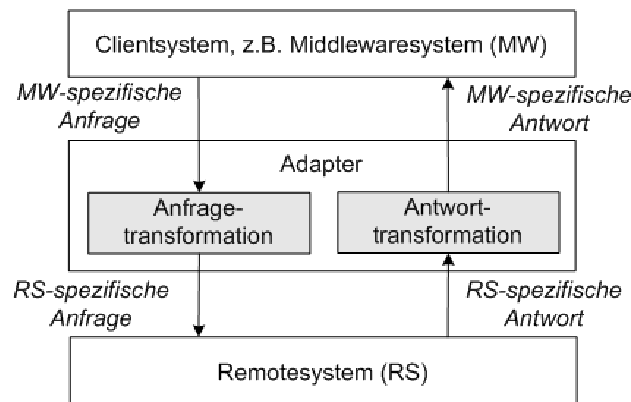


Abbildung 1: Grundlegendes Prinzip eines Adapters.

Adaptertechnologien können sich stark voneinander unterscheiden, da sie für verschiedenartige Middlewaresysteme und IT-Umgebungen konzipiert sind und in verschiedensten Anwendungsszenarien unterschiedliche Remotesysteme integrieren. Das heißt, Adaptertechnologien unterscheiden sich in den verwendeten Technologien, Programmiersprachen, Architekturen, Datenmodellen, Zugriffsmustern, Anwendungsdomänen, Verwendungszwecken usw. Diese Unterschiede sind auch der Grund, weshalb Integrationstechnologien inkompatibel zueinander sind und zum unerwünschten Sachverhalt einer "Integration der Integration" führen. Basierend auf der Vielfalt an Adaptertechnologien kann ein Unternehmen hunderte oder sogar tausende von Adaptern für verschiedene Anwendungen, Services, Middlewaresysteme und Remotesysteme besitzen.

In dieser Arbeit verwenden wir zwei typische Vertreter von Adaptertechnologien um unseren Integrationsmanagement-Ansatz zu evaluieren, nämlich die *J2EE Connector Architecture (J2EE/CA)* [Sun03], die **J2EE-Konnektoren** spezifiziert, und das *SQL Management of External Data (SQL/MED)* [SQL08], das **SQL-Wrapper** spezifiziert. Diese beiden Adaptertechnologien vertreten jeweils entgegengesetzte Verarbeitungsparadigmen, zum einen datenorientierte Verarbeitung (SQL/MED) und zum anderen operationsorientierte Verarbeitung (J2EE/CA). Datenorientierte Adaptertechnologien und operationsorientierte Adaptertechnologien unterscheiden

sich grundsätzlich in Form von Architektur, Datenmodell, Zugriffsmustern, Anwendungsdomäne und Verwendungszweck und stellen somit verschiedene Anforderungen an unseren Integrationsmanagement-Ansatz.

Integrationsmanagement

Warum sollten wir Software für eine neue oder eine sich verändernde IT-Umgebung von Grund auf neu entwickeln, wenn wir bereits existierende Software haben, die alle wesentlichen Anforderungen erfüllt, aber technologische Inkompatibilitäten bezüglich der neuen Situation aufweist? Wenn wir dazu in der Lage sind, diese Inkompatibilitäten zu überwinden, dann gibt es keinen Grund dazu. Wir verwenden die existierende Software! Ein Weg, wie vorhandene Software wiederverwendet werden kann und wie vorhandene technologische Inkompatibilitäten überwunden werden können, ist Gegenstand dieser Arbeit.

Stellen wir uns folgendes Szenario vor: zwei Unternehmen fusionieren und in diesem Zuge sollen auch die IT-Infrastrukturen beider Unternehmen zusammengeführt und vereinheitlicht werden. Betrachten wir die Human-Resources-Anwendungen (HR-Anwendungen) der beiden Unternehmen etwas genauer (s. Abbildung 2). Die HR-Anwendung von Unternehmen A ist im linken Teil der Abbildung dargestellt und basiert auf Java EE-Technologie. Die HR-Anwendung verwendet Kundendaten aus einem Customer-Relationship-Managementsystem (CRM-System) und Personaldaten aus einem LDAP-System (Lightweight Directory Access Protocol). J2EE-Konnektoren sind Bestandteil der Java EE-Architektur und erlauben die Integration von heterogenen Softwaresystemen in Java EE-Anwendungsserver. Die HR-Anwendung von Unternehmen A verwendet einen CRM-J2EE-Konnektor und einen LDAP-J2EE-Konnektor um das CRM-System und das LDAP-System in den Java EE-Anwendungsserver zu integrieren. Dadurch erhält die HR-Anwendung einen einheitlichen Zugang zu den beiden Systemen.

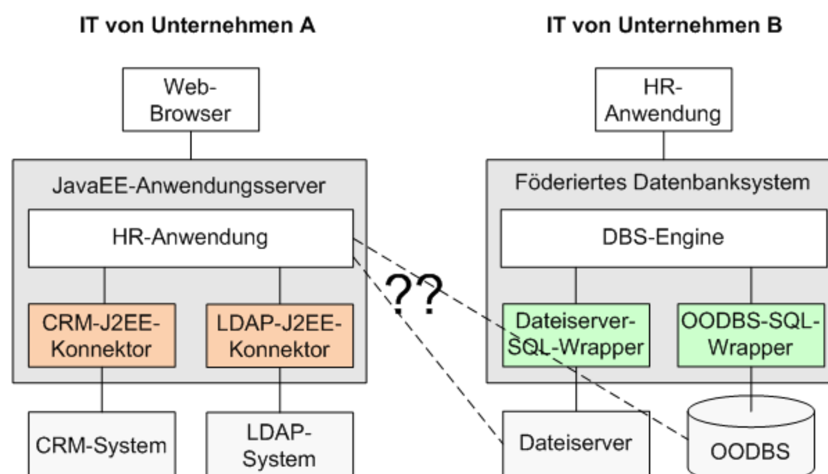


Abbildung 2: Zusammenführen der IT-Infrastrukturen zweier Unternehmen.

Unternehmen B hat ebenfalls eine HR-Anwendung, die jedoch auf einem SQL-basierten föderierten Datenbanksystem (FDBS) aufbaut (s. rechte Seite von Abbildung 2). Diese HR-Anwendung verwendet Angestelltendaten von einem Dateiserver und aus einem objektorientierten Datenbanksystem (OODBS). Der SQL-Standard definiert das Konzept von SQL-Wrappern um Softwaresysteme als externe Datenquellen in ein FDBS zu integrieren. Die HR-Anwendung von Unternehmen B greift somit über den Dateiserver-SQL-Wrapper und den OODBS-SQL-Wrapper auf den Dateiserver und auf das OODBS zu.

Das fusionierte Unternehmen möchte jetzt die beiden HR-Anwendungen zusammenführen, so dass eine daraus resultierende globale HR-Anwendung alle vier Softwaresysteme verwenden kann, also das CRM-System, das LDAP-System, den Dateiserver und das OODBS. Das Unternehmen entscheidet sich dafür die Java EE-basierte HR-Anwendung so zu erweitern, dass auch der Dateiserver und das OODBS in die globale Verarbeitung mit einbezogen werden können. Somit benötigen wir zusätzlich einen Dateiserver-J2EE-Konnektor und einen OODBS-J2EE-Konnektor wie in Abbildung 3 dargestellt. Damit kommen wir allerdings zu einem typischen Integrationsproblem: bisher gibt es weder den Dateiserver-J2EE-Konnektor noch den OODBS-J2EE-Konnektor, d.h. wir müssen die beiden J2EE-Konnektoren komplett neu erstellen. Das ist jedoch ein kostspieliges Vorgehen, da wir dazu entsprechend ausgebildete Softwareentwickler benötigen, die J2EE-Konnektoren für die Java EE-Plattform entwickeln können. Diese Entwickler benötigen außerdem Wissen über die Anwendungsumgebung und die verwendeten Softwaresysteme und sie benötigen Zeit und Ressourcen, um die Konnektoren bis zur Produktionsreife und zum Einsatz zu führen.

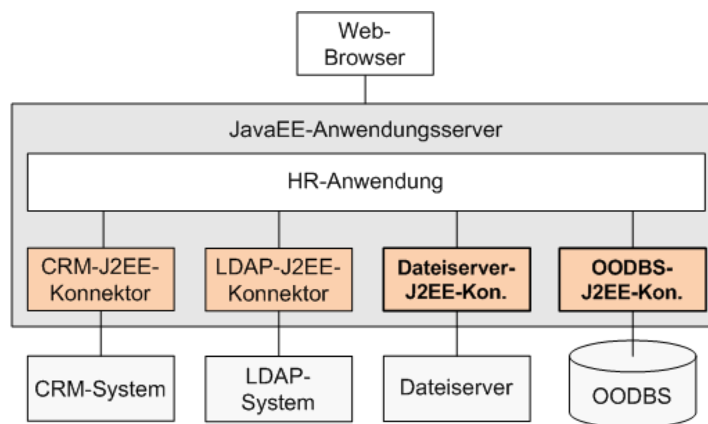


Abbildung 3: Entwicklung neuer J2EE-Konnektoren.

Auf der anderen Seite integriert die FDBS-basierte HR-Anwendung bereits den Dateiserver und das OODBS mit den entsprechenden SQL-Wrappern. Die Integrationsvorgänge im Dateiserver-SQL-Wrapper und im OODBS-SQL-Wrapper sind recht ähnlich zu den Integrationsvorgängen im erforderlichen Dateiserver-J2EE-Konnektor und OODBS-J2EE-Konnektor, z.B. das Lesen von Dateien vom Dateis-

erver oder das Anfragen von Objekten im OODBS. Aus diesem Grund wäre es sehr vorteilhaft, wenn wir die aufwändige Neuentwicklung der J2EE-Konnektoren vermeiden und stattdessen die bereits vorhandenen SQL-Wrapper wiederverwenden könnten. Allerdings ist eine derartige Wiederverwendung nicht ohne Weiteres umzusetzen, da J2EE-Konnektortechnologie und SQL-Wrappertechnologie beträchtliche technologische Inkompatibilitäten aufweisen, z.B. bezüglich des Daten- und Verarbeitungsmodells. Dies macht es unmöglich den Dateiserver-SQL-Wrapper und den OODBS-SQL-Wrapper direkt im Java EE-Anwendungsserver wiederzuverwenden. Am Ende blieb bisher nur die konventionelle Lösung der Neuentwicklung, d.h. in unserem Beispiel wären der Dateiserver-J2EE-Konnektor und der OODBS-J2EE-Konnektor von Grund auf neu entwickelt worden (s.a. Abbildung 3). Wir wären also gezwungen einen langen und aufwändigen Entwicklungsprozess zu durchlaufen, obwohl die vorhandenen SQL-Wrapper bereits sehr ähnliche Integrationsvorgänge realisieren. Die Frage ist nun, ob das wirklich notwendig ist oder ob es doch einen Weg gibt, um aus den bereits vorhandenen SQL-Wrappern einen besseren Nutzen ziehen zu können.

In dieser Arbeit führen wir den Begriff **Integrationsmanagement (IM)** ein. Wir definieren **Integrationsmanagement-Technologie (IM-Technologie)** als ein Mittel zur systematischen Verwendung von Integrationstechnologien und zum Bereitstellen von **Integrationsunabhängigkeit**, damit Anwendungsentwickler unabhängig von Integrationsvorgängen wie denen im Beispielszenario sind, d.h. IM-Technologie abstrahiert von Integrationstechnologien. Ein **Integrationsmanagement-System (IM-System)** ist ein Softwaresystem, das einen einheitlichen Zugang zu Integrationsvorgängen der IT-Infrastruktur anbietet wie in Abbildung 4 dargestellt, d.h. ein IM-System ermöglicht das Extrahieren von Integrationsvorgängen aus einer IT-Infrastruktur und seinen Anwendungen und stellt die extrahierten Integrationsvorgänge zentral zur Wiederverwendung zur Verfügung.

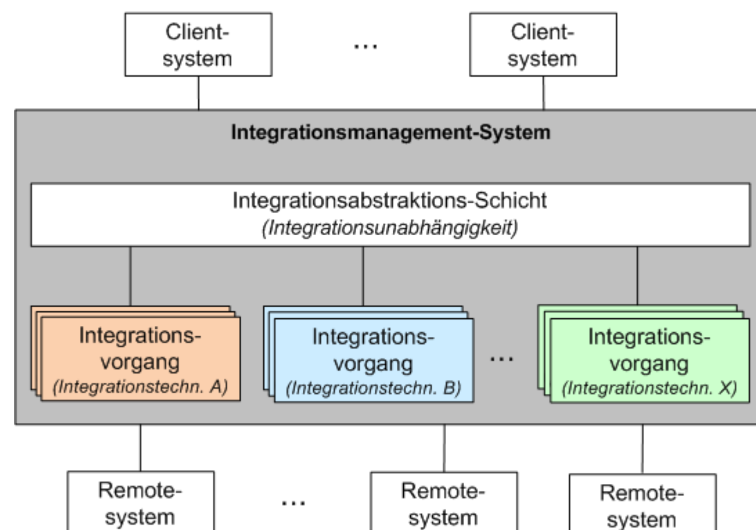


Abbildung 4: Basisarchitektur eines IM-Systems.

Wir haben ein IM-System entworfen, das **Virtualization Tier (VT)**, das die Wiederverwendung von Adaptern wie z.B. SQL-Wrappern oder J2EE-Konnektoren ermöglicht und dadurch die Komplexität und die Kosten zur Umsetzung von Integrationsvorgängen in IT-Infrastrukturen reduziert. Außerdem haben wir die Performanz unseres VT-Prototypen evaluiert um zu zeigen, dass IM-Technologie effizient eingesetzt werden kann. Abbildung 5 skizziert unseren Integrationsmanagement-Ansatz, das VT, und wie es auf das Integrationsszenario in Abbildung 2 angewendet werden kann: das VT verwendet den Dateiserver-SQL-Wrapper und den OODBS-SQL-Wrapper und erlaubt dadurch der Java EE-basierten HR-Anwendung den Zugriff auf den Dateiserver und das OODBS über die SQL-Wrapper anstelle einer Neuentwicklung von entsprechenden J2EE-Konnektoren.

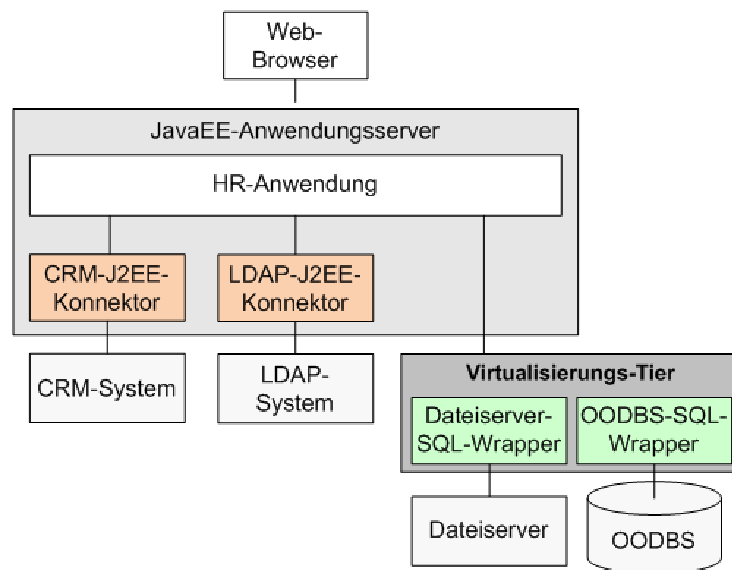


Abbildung 5: Wiederverwendung der SQL-Wrapper.

Adaptervirtualisierung

Das Virtualization Tier ist ein systematischer Integrationsmanagement-Ansatz, der die einheitliche Verwendung unterschiedlicher Adaptertechnologien ermöglicht und dem Softwareentwickler die Umsetzung der damit verbundenen Integrationsvorgänge zur Verfügung stellt. Das VT ermöglicht die Wiederverwendung von Adaptern in unterschiedlichen Integrationsszenarien und mit verschiedenen Middlewareplattformen. Es virtualisiert Adapter und verwendet diese dadurch so als ob sie in ihrer jeweiligen Middlewareplattform ausgeführt würden. Virtualisierung bedeutet hierbei die einheitliche Behandlung und Verwendung verschiedener Arten von Adaptern, so dass Anwendungsentwickler sich nicht um Integrationsvorgänge kümmern müssen, sondern unabhängig davon ihre Anwendungen entwickeln können. Das Konzept der Integrationsunabhängigkeit ist beispielsweise analog zum Konzept der Datenun-

abhängigkeit in Datenbankverwaltungssystemen, das den Anwendungsentwickler davor schützt sich im Detail mit der Verwaltung und Behandlung von Daten befassen zu müssen.

Abbildung 6 zeigt die Architektur des VT. Jede Adaptertechnologie benötigt einen Adapter-Manager, der für die korrekte Ausführung der Adapter dieser Adaptertechnologie zuständig ist. Deshalb realisiert sowohl ein Adapter-Manager als auch ein Middlewaresystem, das diese Art von Adaptern ausführen kann, dieselbe Funktionalität. Damit ist es möglich, dass die bereits vorhandene Funktionalität des Middlewaresystems zur Ausführung der Adapter als Grundlage zur Realisierung eines entsprechenden Adapter-Managers verwendet werden kann. Das ist sogar sinnvoll, da diese Funktionalität einen großen Teil eines Adapter-Managers ausmacht.

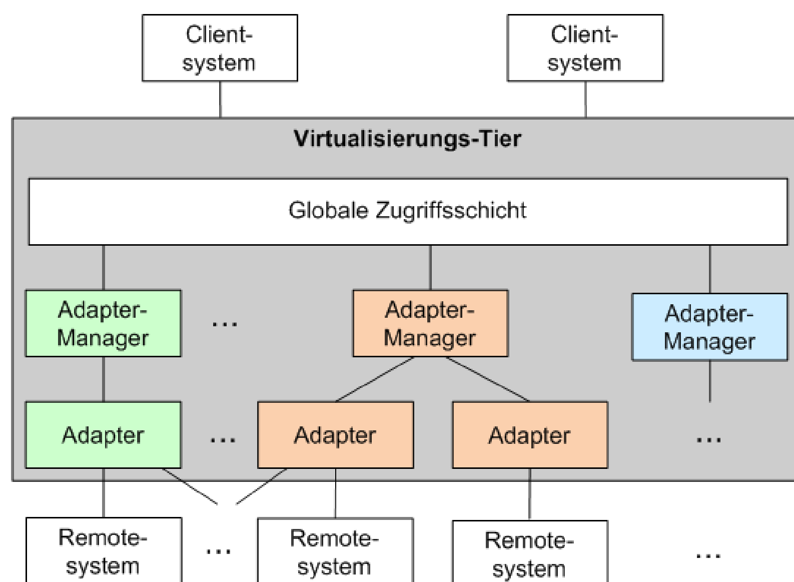


Abbildung 6: Architektur des VT.

Die globale Zugriffsschicht des VT stellt einen einheitlichen Zugriff auf die verschiedenen Adapter und Remotesysteme zur Verfügung. Abbildung 7 zeigt, wie die VT-basierte Lösung unseres Integrationsszenarios aus Abbildung 5 aussieht. Das VT verwendet einen SQL-Wrapper-Manager um SQL-Wrapper ausführen zu können, z.B. den Dateiserver-SQL-Wrapper oder den OODBS-SQL-Wrapper. Außerdem verwendet das VT noch einen Message-Broker-Adapter-Manager (MB-Adapter-Manager) um MB-Adapter ausführen zu können, z.B. den MB-Adapter der Customer-Relationship-Management-Anwendung (CRM-Anwendung). Der J2EE-Konnektor-Manager wiederum ist verantwortlich für die Ausführung von J2EE-Konnektoren und so weiter. Die HR-Anwendung kann jetzt den VT-J2EE-Konnektor verwenden um das VT dazu zu verwenden die verschiedenen Adapter auszuführen, die im VT installiert sind. Der große Vorteil dieses Lösungsansatzes ist, dass die beiden SQL-Wrapper, der MB-Adapter und die anderen im VT installierten Adapter komplett wiederverwendet werden können, wohingegen die konventionelle Lösung wie

im Beispiel in Abbildung 3 immer wieder die Neuentwicklung von Adaptern, hier J2EE-Konnektoren, erfordert.

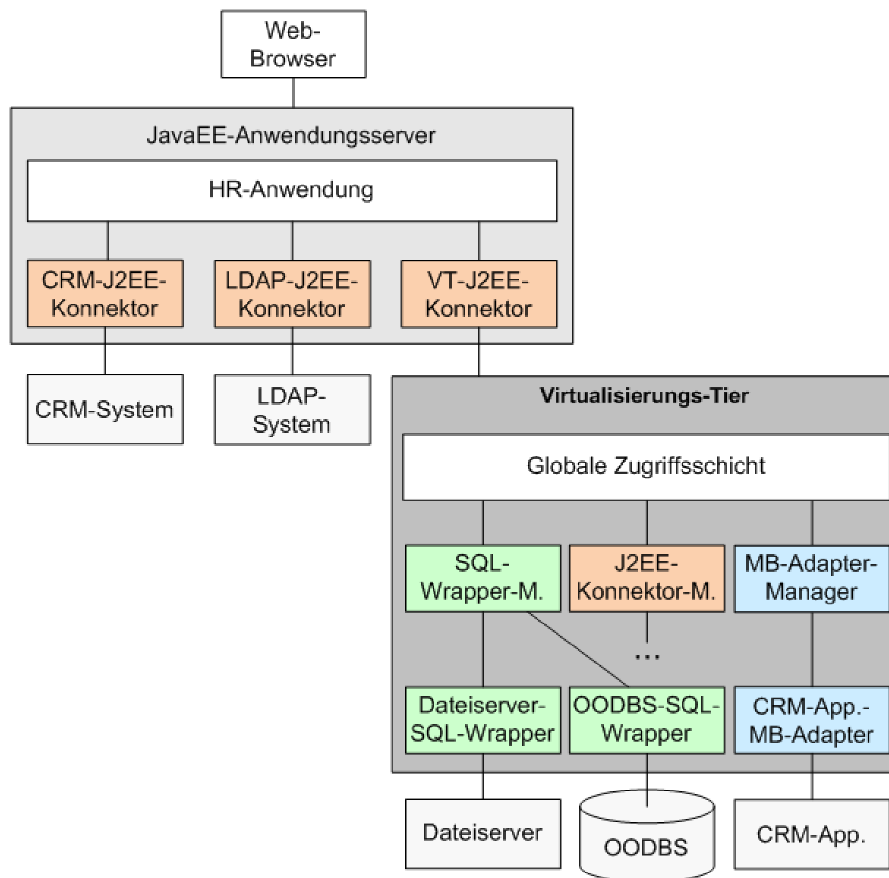


Abbildung 7: VT-basierte Integrationslösung.

Architekturevaluation

Ein wichtiger Aspekt des VT-Ansatzes ist, dass im Einsatz befindliche Middlewaresysteme nicht abgeändert werden müssen und dass im Einsatz befindliche Anwendungen und Prozesse nicht durch die Verwendung des VT beeinträchtigt werden. Das VT erweitert die Möglichkeiten von Middlewaresystemen auf nicht-invasive Art und Weise, indem es eine Alternative zum Zugriff auf Remotesysteme anbietet. Diese Alternative verwendet die native Adaptertechnologie einer Middlewareplattform um auf das VT zuzugreifen, d.h. in Form eines VT-Middleware-Adapters. Der Java EE-Anwendungsserver in unserem Integrationsszenario in Abbildung 7 verwendet zum Beispiel einen VT-J2EE-Konnektor um auf das VT zuzugreifen.

Ein weiterer Vorteil des VT-Ansatzes ist eine Erhöhung der Flexibilität der IT-Infrastruktur, da zukünftige Änderungen und Anforderungen bezüglich von Integrationsvorgängen ebenfalls mit dem VT gelöst werden können. Abstrakt betrachtet

kann das VT als ein Middleware-Multiplexer betrachtet werden, der einem Middlewaresystem den Zugriff auf jeden im VT installierten Adapter ermöglicht. Wenn m Middlewareplattformen n Remotesysteme *ohne* Verwendung des VT verwenden sollen, dann würden wir potentiell $m * n$ Adapter benötigen, wie in Abbildung 8 dargestellt. Das VT reduziert diese Komplexität auf $m + n$ Adapter, da jedes Middlewaresystem über das VT einen einheitlichen Zugriff auf alle im VT installierten Adapter erhalten kann, wie in Abbildung 9 dargestellt (das sind m Adapter um von den Middlewaresystemen auf das VT zuzugreifen und n Adapter um vom VT auf die Remotesysteme zuzugreifen). In anderen Worten: ein konventioneller Integrationsansatz ist immer mit einem spezifischen Middlewaresystem und einer spezifischen Adaptertechnologie verbunden um ein Remotesystem zu integrieren. Das VT hingegen erlaubt es mit dem gewählten Middlewaresystem jeden möglichen im VT installierten Adapter zu verwenden, der auf dieses Remotesystem zugreifen kann. Das VT ist ein systematischer und integrationstechnologieunabhängiger Ansatz, der offen ist für jede Middlewareplattform und für jede Anwendung, die die Vorteile des VT-Ansatzes nutzen möchte.

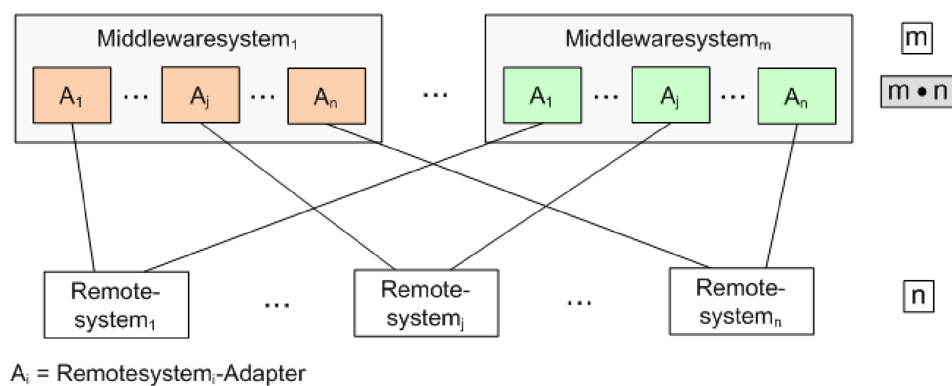


Abbildung 8: Konventionelle Integration: potentiell $m * n$ Adapter.

Bewertung

Es gibt eine Fülle von Integrationstechnologien und es gibt immer noch keine Möglichkeit sie systematisch zu behandeln. Dies führt zu einer zunehmenden Komplexität von IT-Umgebungen und zu zunehmenden Integrationsaufwänden und IT-Kosten. Integrationsmanagement stellt eine Möglichkeit dar Integrationstechnologien systematisch und einheitlich zu verwenden. Es abstrahiert von Integrationstechnologien, so dass Integrationsvorgänge unabhängig vom Rest der Softwareentwicklung umgesetzt werden können. Die dadurch erzielte Integrationsunabhängigkeit kann die Wartung und Fortentwicklung von IT-Umgebungen deutlich erleichtern und damit die Komplexität und die Kosten von IT-Landschaften signifikant reduzieren. Außerdem zeigt unsere Performanzauswertung des VT-Ansatzes, dass IM-Technologie auch effizient arbeiten kann. Damit ist IM-Technologie eine Schlüssel-

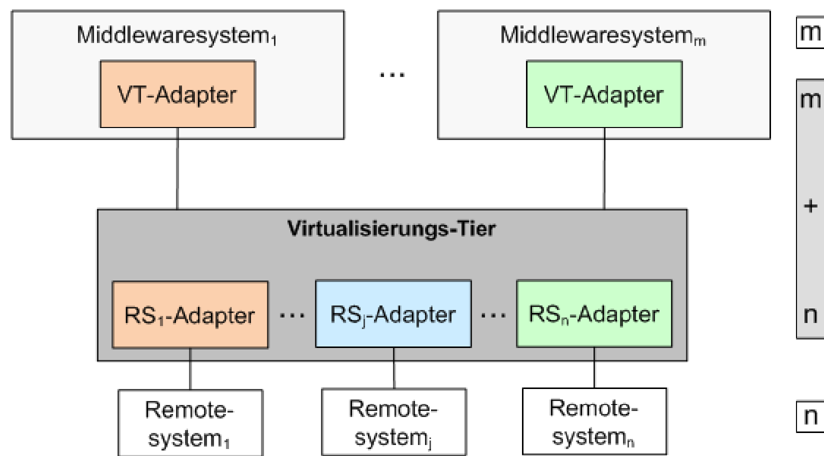


Abbildung 9: VT als Multiplexer: nur $m + n$ Adapter erforderlich.

technologie im einheitlichen Umgang mit Integrationstechnologien und zur Bewältigung von Heterogenitäten in IT-Landschaften.

Contents

1 Motivation **27**

2 Integration **31**
2.1 Integration Architectures . . . 32
2.2 Adapter Technologies . . . 34
2.3 Integration Issues . . . 38
2.4 Generation and Reuse . . . 42
2.5 Adapter Virtualization . . . 43
2.6 Integration Management . . . 49
2.7 Summary . . . 51

3 Data Model **53**
3.1 VT Objects . . . 53
3.2 Semantics . . . 54
3.3 Transactions . . . 55
3.4 Object Identity . . . 56
3.5 Non-Uniqueness . . . 58
3.6 Object References . . . 64
3.7 Summary . . . 68

4 Processing Model **69**
4.1 Processing Transparency . . . 69
4.2 VTO Access Operations . . . 75
4.2.1 Read Operation . . . 75
4.2.2 Create Operation . . . 76
4.2.3 Update Operation . . . 78
4.2.4 Delete Operation . . . 79
4.2.5 Operation-Oriented Data Usage . . . 80
4.3 VTQL Requests . . . 81
4.3.1 Read Requests . . . 81
4.3.2 Create Requests . . . 82
4.3.3 Update Requests . . . 82
4.3.4 Delete Requests . . . 83
4.3.5 Data-Oriented Data Usage . . . 83

4.4 Query Execution 84
4.5 Summary 87

5 Deployment Model 89
5.1 VT Object Configurations 89
5.1.1 Adapter Manager Information Chapter 92
5.1.2 Adapter Information Chapter 92
5.1.3 System Information Chapter 94
5.1.4 Object Information Chapter 95
5.1.5 Object Definition Chapter 97
5.2 Example Requests 101
5.3 Deployment Process 105
5.4 Mapping Issues 108
5.5 Reuse of Middleware Infrastructure 109
5.5.1 Reuse of Adapter Deployments 109
5.5.2 Reuse of Middleware Infrastructure 112
5.6 Summary 118

6 Applicability 121
6.1 Architecture Patterns 121
6.1.1 Implementation Pattern 122
6.1.2 Connection Pattern 123
6.1.3 Adapter Reuse Pattern 125
6.1.4 Deployment Reuse Pattern 128
6.1.5 Middleware Reuse Pattern 128
6.2 Web Services 131
6.2.1 Web Services & IM Technology 131
6.2.2 Web Service Infrastructures 132
6.2.3 The VT Acting as an ESB 133
6.3 Summary 136

7 Performance 139
7.1 Experiment Environment 139
7.2 Experiment Execution 142
7.2.1 Experiment Categories 142
7.2.2 Query Considerations 146
7.2.3 Experiment Architectures 147
7.2.4 Experiment Notation 147
7.3 Challenges 150
7.4 Experiment Results 151
7.4.1 Operations 152
7.4.2 Reading Data 153
7.4.3 Writing Data 156
7.4.4 Queries 158

7.5 Optimization Effects 168
7.6 Summary 175

8 Conclusion **177**

List of Figures

1.1 Merging IT Infrastructures. 27
1.2 Developing new J2EE Connectors. 28
1.3 Reuse of SQL Wrappers. 29

2.1 Integration Architecture Categories. 32
2.2 Basic Concept of an Adapter. 35
2.3 Extended Integration Scenario. 41
2.4 Reuse of the CRM App. MB Adapter. 44
2.5 Virtualization Tier Architecture. 45
2.6 VT-Based Solution. 46
2.7 VT-Based MB Solution. 47
2.8 Middleware Enhancement. 47
2.9 Conventional: Potentially $m * n$ Adapters Required. 48
2.10 VT as Multiplexer: Only $m + n$ Adapters Required. 48
2.11 VT Scalability. 49
2.12 Integration Management System Architecture. 50

3.1 VT Object Mapping – First Request. 57
3.2 VT Object Mapping – Second Request. 58
3.3 VT Object Mapping – Update. 60
3.4 VT Object Mapping – Non-Unique (Mismatch). 61
3.5 VT Object Mapping – Non-Unique (Correct). 63
3.6 VT Object Mapping – Duplicates. 65
3.7 Dereferencing VT Objects. 66
3.8 Non-Explicitly Resolvable VT Object References. 67

4.1 VT Objects and VT Access. 72

5.1 VT Object Configuration. 90
5.2 Configuration Chapter Hierarchy. 91
5.3 Example Configuration Chapter Hierarchy. 91
5.4 SQL Wrapper Manager Information Chapter (case W). 92
5.5 J2EE Connector Manager Information Chapter (case C). 92
5.6 SQL Wrapper Information Chapter (case W). 93
5.7 SQL CREATE WRAPPER Statement (case W). 93

5.8 J2EE Connector Information Chapter (case C). 93
5.9 J2EE Connector Deployment (case C). 94
5.10 CRM DBS Information Chapter (case W). 95
5.11 SQL CREATE SERVER Statement (case W). 95
5.12 CRM System Information Chapter (case C). 95
5.13 CRM_EMPLOYEE Information Chapter (case W). 96
5.14 SQL CREATE FOREIGN TABLE Statement (case W). 96
5.15 RSEmp Information Chapter (case C). 97
5.16 Java Classes of the J2EE Connector (case C). 98
5.17 VTEmployee Object Definition (case W). 99
5.18 VTEmp Object Definition (case C). 100
5.19 Request Processing (SQL Wrapper Example). 101
5.20 Request Processing (J2EE Connector Example). 104
5.21 Deployment Responsibilities. 106
5.22 Deployment of an SQL Wrapper. 107
5.23 Deployment of a J2EE Connector. 108
5.24 Extended Deployment – VT Object Mapping Issues. 109
5.25 Reusing Adapter Deployments. 111
5.26 Reusing an FDBS Middleware Infrastructure. 114
5.27 FDBS Adapter Information Chapter. 115
5.28 FDBS System Information Chapter. 115
5.29 FDBS Object Information Chapter. 116
5.30 FDBS Table Representation. 116
5.31 FDBS Object Definition Chapter. 117
5.32 Reusing a Message Broker Infrastructure. 118

6.1 Architecture Pattern Overview. 121
6.2 Developing new J2EE Connectors. 123
6.3 FDBS Integration. 124
6.4 Message Broker Integration. 125
6.5 Java EE Application Server Integration. 126
6.6 Reusing Adapters. 127
6.7 Reusing Adapter Deployments. 129
6.8 Reusing Middleware Infrastructure. 130
6.9 The VT, Web Services and more than Web Services. 132
6.10 HR BPEL Process (abstract representation). 134
6.11 HR BPEL Process Solved with ESB. 134
6.12 HR BPEL Process Solved with Middleware Systems. 135
6.13 HR BPEL Process Solved with VT. 136

7.1 VT Prototype Experiment Scenario. 140
7.2 TPC-H Tables. 141
7.3 OOApp Classes. 141
7.4 Overall VT Prototype Experiment Architecture. 143

7.5 Experiment Categories *Exec*, *Read*, *Write*, and *Query*. 144
7.6 SQL Wrapper Experiments Architectures. 148
7.7 J2EE Connector Experiments Architectures. 149
7.8 Exec-DB2. 153
7.9 Exec-WS. 154
7.10 Read-DB2. 155
7.11 Read-WS. 155
7.12 Read-DB2: Leaps. 156
7.13 Read-WS: Leaps. 156
7.14 Read-DB2: Leaps (absolute execution time). 157
7.15 Read-WS: Leaps (absolute execution time). 157
7.16 Write-DB2. 159
7.17 Write-WS. 159
7.18 Write-DB2-File. 160
7.19 Write-DB2-File (absolute execution time). 160
7.20 Write-DB2-Derby. 161
7.21 Write-DB2-Derby (absolute execution time). 161
7.22 Write-DB2-OOApp. 162
7.23 Write-DB2-OOApp (absolute execution time). 162
7.24 Write-WS-Derby. 163
7.25 Write-WS-OOApp. 163
7.26 Write-WS-Derby (absolute execution time). 164
7.27 Write-WS-OOApp (absolute execution time). 164
7.28 Request and Data Pipelining Potential. 165
7.29 Query-R-DB2. 166
7.30 Query-R-WS. 166
7.31 Query-SP-DB2/WS (query processing in DB2 and in the WS client). 166
7.32 Query-SP-DB2/WS-VT (query processing in DB2 and in the WS client). . . 167
7.33 Query-SP-DB2-Derby (query processing in DB2). 167
7.34 Query-SP-WS-Derby (query processing in the WS client). 168
7.35 Query-SP-DB2-File/OOApp (query processing in DB2). 168
7.36 Query-SP-WS-File/OOApp (query processing in the WS client). 169
7.37 Query-SP-DB2/WS-VT (query processing in the VT). 169
7.38 Query-SP-DB2 (query processing in the VT). 170
7.39 Query-SP-WS (query processing in the VT). 170
7.40 Query-J-DB2 (query push-down). 171
7.41 Query-J-WS (query push-down). 171
7.42 Query-J-DB2 (query compensation in DB2). 172
7.43 Query-J-WS (query compensation in the WS client). 172
7.44 Degeneration Effect: Querying Derby directly. 173
7.45 Query-SPJ-DB2: Derby Degeneration Effect. 173
7.46 Query-SPJ-WS: Derby Degeneration Effect. 174

Abbreviations

Common Abbreviations

API	Application Programming Interface
BPEL	Business Process Execution Language
CCI	Common Client Interface
CORBA	Common Object Request Broker Architecture
CRM	Customer Relationship Management
CRUD	Create Read (also: Retrieve) Update Delete
DBMS	Database Management System
DBS	Database System
EJB	Enterprise Java Bean
ESB	Enterprise Service Bus
FDBS	Federated Database System
HR	Human Resources
IDL	Interface Definition Language
IT	Information Technology
J2EE	Java 2 Enterprise Edition
Java EE	Java Platform, Enterprise Edition
JBI	Java Business Integration
JCP	Java Community Process
JDBC	Java Database Connectivity
JDO	Java Data Objects
JMS	Java Message Service
JSP	Java Server Page
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MB	Message Broker
ODMG	Object Database Management Group
OODBS	Object-Oriented Database System
OQL	Object Query Language
ORB	Object Request Bus
PDM	Product Data Management
REST	Representational State Transfer
RPC	Remote Procedure Call

SOAP	Simple Object Access Protocol
SPI	Service Provider Interface
SQL/MED	SQL Management of External Data
UDDI	Universal Description, Discovery and Integration
WSDL	Web Services Description Language
XML	Extensible Markup Language

Abbreviations and Terms in this Work

Adapter Deployer Person who deploys adapters into a middleware system.
Adapter Information Chapter Part of a VT object configuration. Defines information about the adapter that is used to resolve the associated VT object.
Adapter Manager Information Chapter Defines information about an adapter manager. Used by the VT to properly handle adapters that are managed by the adapter manager.
Additional VTO Access Operation Parameter A parameter that is correlated with a specific remote operation. Part of the VTO operation structure that represents the associated remote operation.
Explicitly Resolvable VT Object Reference A VT object reference that consists of a reference key and a reference operation. Used to explicitly resolve a referenced VT object.
IM – Integration Management Systematic treatment of and abstraction from integration issues. An IM system is a system that abstracts from integration issues and that provides a systematic means of dealing with them. IM technology is a technology that enables systematic treatment of and abstraction from integration issues.
Inherent VTO Access Operation Parameter A parameter that is always required for the correlated VTO access operation type.
Non-Explicitly Resolvable VT Object Reference A VT object reference that cannot be explicitly resolved, but has to be resolved when the referencing VT object is resolved.
Non-Unique VT Object A VT object without a remote identity.
Object Definition Chapter Part of a VT object configuration. Defines the VT object that is associated with the VT object configuration.
Object Information Chapter Part of a VT object configuration. Defines information about the remote data and the remote operations that are mapped to the associated VT object.
Remote Identity Identifies a remote entity that is associated with a VT object instance.
System Information Chapter Part of a VT object configuration. Defines information about the remote system that contains the data and the operations that are mapped to the associated VT object.
Unique VT Object A VT object with a remote identity.
VT – Virtualization Tier An IM system. Our prototype that we realized to ex-

emplify IM technology and that we used to perform a qualitative and quantitative analysis of IM technology.
VT Data Model The data model employed in the VT to represent remote data and remote operations.
VT Identity Uniquely identifies a VT object instance.
VT Object Part of the VT Data Model. Can represent remote data as well as remote operations.
VT Object Class A VT object class is the definition of a VT object.
VT Object Configuration Defines a VT object and how data and operations of a remote system are mapped via an adapter to the VT object and vice versa. Consists of four configuration chapters: adapter information chapter, system information chapter, object information chapter, object definition chapter.
VT Object Cursor A VT object that efficiently handles data, e.g. the result of a VTQL read request.
VT Object Deployer Person who defines and deploys VT objects in the VT.
VT Object Instance A VT object instance contains data according to the associated VT object class.
VT Object Operation Part of a VT object. Represents a remote operation.
VT Object Reference Used in a VT object to reference another VT object.
VTO Access Operation – VT Object Access Operation A standardized VT object operation. There are four types: create, read, update, delete.
VTO Operation Structure – VT Object Operation Structure A data structure that represents additional operation parameters of a VTO access operation.
VTQL – VT Query Language Based on OQL. A declarative, set-oriented query language that employs the VT data model. There are four types of VTQL requests: create, read, update, delete.

Chapter 1

Motivation

Why should we develop software for a new or changing IT scenario from scratch if we already have existing software that meets our overall requirements although it bears some technological incompatibilities to the new situation? If we are able to handle the incompatibilities, there is no reason. We reuse the existing software! A way how we can reuse existing software and how we can deal with technological incompatibilities in IT environments is subject of this work.

Imagine the following scenario: two companies merge and they also merge their IT. We take a closer look at the human resources (HR) applications of the two companies (see Figure 1.1). Company A has an HR application based on Java EE technology (left part of Figure 1.1). The HR application handles customer data from a customer relationship management (CRM) system and personnel data from an LDAP (Lightweight Directory Access Protocol) system [LDA06]. The Java EE architecture provides the concept of J2EE connectors to integrate heterogeneous software systems into a Java EE application server. The HR application of company A employs a CRM J2EE connector and an LDAP J2EE connector to integrate the

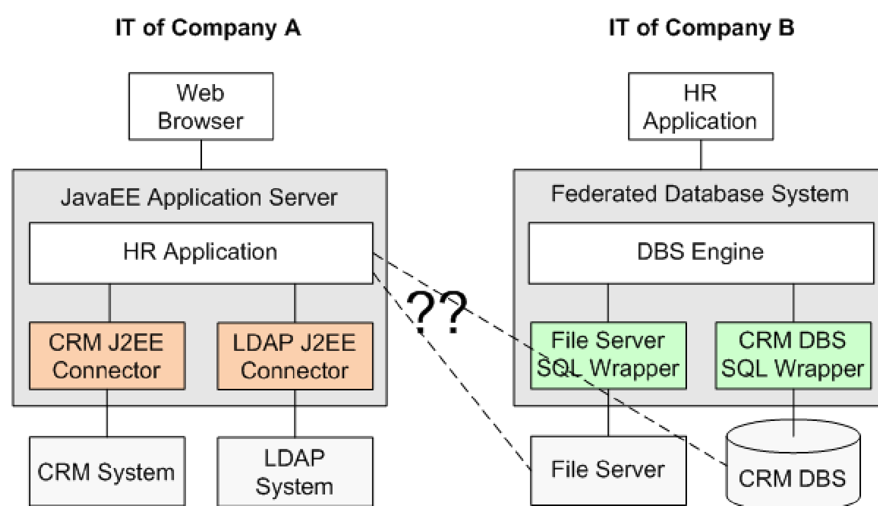


Figure 1.1: Merging IT Infrastructures.

CRM system and the LDAP system into the Java EE application server so that the HR application can uniformly access both systems.

Company B has an HR application, too. This HR application employs an SQL-based [SQL08] federated database system (FDBS) (right part of Figure 1.1). The HR application uses employee data from a file server and from an object-oriented CRM DBS. The SQL standard defines the concept of SQL wrappers to integrate software systems into an FDBS. The HR application of company B accesses the file server and the CRM DBS via the file server SQL wrapper and the CRM DBS SQL wrapper in the FDBS.

The merged company now also merges both HR applications, which means that we need a global HR application that can deal with all four software systems, i.e. the CRM system, the LDAP system, the file server and the CRM DBS. The merged company decides to extend the HR application of company A and to integrate the file server and the CRM DBS into the Java EE application server (see Figure 1.1). Hence, we need a file server J2EE connector and a CRM DBS J2EE connector. However, we face a typical integration problem: we do neither have a file server J2EE connector nor do we have a CRM DBS J2EE connector. Consequently, we have to develop the file server J2EE connector and the CRM DBS J2EE connector from scratch (see Figure 1.2). This is a costly task since we need skilled developers who are able to develop J2EE connectors for the Java EE platform. The developers also need knowledge about the specific environment, i.e. the software and systems that are involved in the integration tasks, and they need time and resources to develop the connectors until they can use them productive.

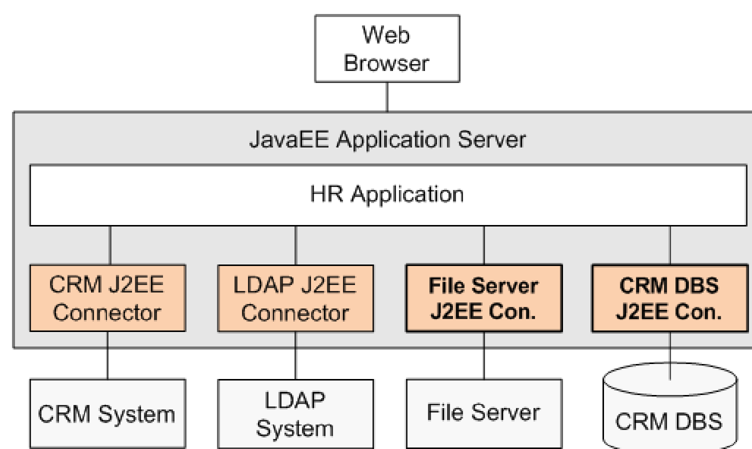


Figure 1.2: Developing new J2EE Connectors.

On the other hand, the FDBS-based HR application of company B already integrates the file server and the CRM DBS by means of SQL wrappers. The integration tasks of the file server SQL wrapper and the CRM DBS SQL wrapper are quite similar to that of the required file server J2EE connector and CRM DBS J2EE connector, e.g. reading files from the file server or submitting object requests to the CRM DBS. Therefore, it would be very appealing if we were able to avoid the

costly task of developing new J2EE connectors from scratch and if we could simply reuse the existing SQL wrappers. However, it is not that easy since J2EE connector technology and SQL wrapper technology bear considerable technological incompatibilities, e.g. regarding data model or operation model, so that it is impossible to directly employ the file server SQL wrapper and the CRM DBS SQL wrapper in the Java EE application server. In the end, the conventional solution is to develop the file server J2EE connector and the CRM DBS J2EE connector from scratch (as shown in Figure 1.2) although they have much in common with the file server SQL wrapper and the CRM DBS SQL wrapper. We are forced to go a long and laborious way although the existing SQL wrappers already solve similar integration tasks. But is this really necessary? Isn't there a way to benefit from the existing SQL wrappers?

In this work, we introduce and define **integration management technology (IM technology)** as a means to systematically deal with integration technologies and as a means to provide integration independence so that application developers are shielded from integration tasks such as the ones in the merger example. We develop an integration management system (IM system), i.e. the **virtualization tier (VT)**, which is based on IM technology. The VT allows to reuse adapters such as SQL wrappers or J2EE connectors and thereby reduces complexity and costs of integration tasks in IT infrastructures. Finally, the performance evaluation of our VT prototype shows that IM technology can work efficiently. Figure 1.3 sketches our integration management approach, i.e. the VT, and how it applies to our integration scenario: the VT reuses the file server SQL wrapper and the CRM DBS SQL wrapper and thereby allows the Java EE-based HR application to access the file server and the CRM DBS via the SQL wrappers instead of developing new J2EE connectors.

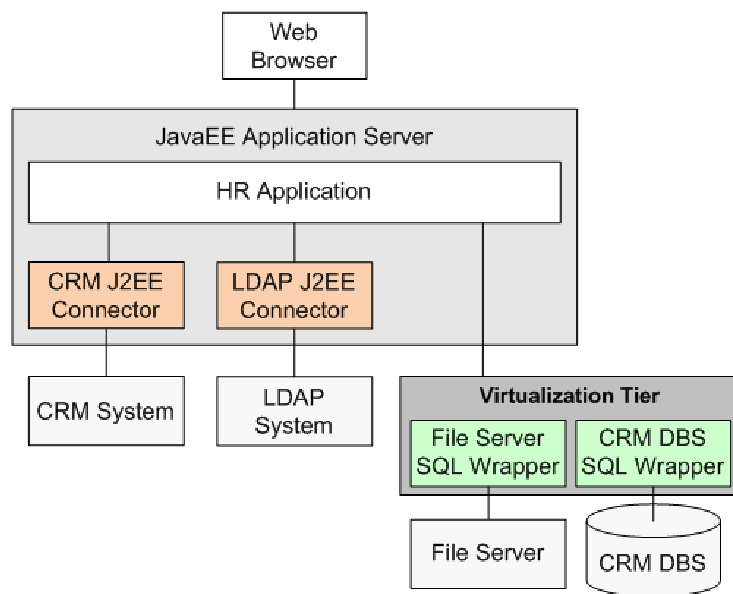


Figure 1.3: Reuse of SQL Wrappers.

In the next chapter (Chapter 2, Integration, also see [WM07d]), we discuss integration means in IT infrastructures. We give a detailed analysis of integration issues, e.g. such as the ones sketched in this chapter, and we introduce the virtualization tier and integration management as a systematic means to cope with integration issues. Thereafter, we show how data and operations of integrated remote systems are represented in the VT (Chapter 3, Data Model, also see [WM07d, WM07a]) and how access to data and operations in the VT is realized (Chapter 4, Processing Model, also see [WM07d, WM07a]). The next important questions are how data and operations in the VT are defined and how adapters are deployed into the VT (Chapter 5, Deployment, also see [WM07b, WM09]). Finally, the most important issues are how the VT can be applied to integration scenarios (Chapter 6, Applicability, also see [WM07c, WM09]) and whether the VT approach is efficient (Chapter 7, Performance, also see [WM09]).

Chapter 2

Integration

Companies can have up to thousands of different software systems and applications and they are usually spread over the whole company. The tasks and processes of a company are more and more tightly coupled with these software systems. This means that software systems have to be interconnected and orchestrated. Thus, the integration of software systems is a key issue in a modern and competitive company. The problem is that the software systems of a company have been bought, developed, installed and maintained in different places, at different times, by different people, for different purposes and they employ diverse technologies that are not compatible to each other. There are different integration technologies and platforms to cope with these challenging issues. They are subsumed under terms such as enterprise application integration, data integration or process integration. Their common goal is to bridge the heterogeneities of diverse software systems.

Middleware technology is a common means to achieve integration. A **middleware technology** specifies an architecture that can integrate diverse software systems (we call them **remote systems**) and that allows other software systems (we call them **client systems**) to uniformly access them (also see Figure 2.1). A **middleware system** is a software system that realizes middleware technology. Examples of middleware systems are Java EE application servers, federated database systems or message brokers. The integration tasks are usually performed by means of adapters. An **adapter*** is a software artifact that is used by a software system, e.g. a middleware system, to access another software system and/or vice versa. Examples of adapters are J2EE connectors, SQL wrappers or message broker adapters. An **adapter technology** specifies an architecture or a part of an architecture to overcome the heterogeneities of diverse remote systems by means of adapters, e.g. SQL/MED [SQL08] or the J2EE connector architecture [Sun03]. An **integration technology** specifies an architecture that comprises a middleware technology and an adapter technology that is employed by the middleware technology. Examples of in-

*Adapter functionality is used in diverse application domains, architectures and systems. Therefore, there are different names in use such as *adapter*, *connector*, *wrapper* or *gateway*. We use the term *adapter* since this it is not solely coined to a specific domain or architecture and since it is in wide-spread use.

tegration technologies are Java EE application server technology, federated database technology or message broker technology.

2.1 Integration Architectures

We categorize integration technologies into four categories of integration architectures as depicted in Figure 2.1. The first category is the **implicit integration architecture**. A client system simply interconnects with a remote system as needed. For example, a decision support application (*client system*) needs the price of an item and therefore calls the get_price(item_no) operation in a product data management (PDM) system (*remote system*) via RPC. The decision support application issues the RPC call directly in its application logic (*application module*). There is no explicit distinction between the integration task and the application logic itself. Clearly, this implicit integration task is not easily repeatable nor reusable for similar situations since the integration logic is not separated from the application logic in terms of a reusable software artifact.

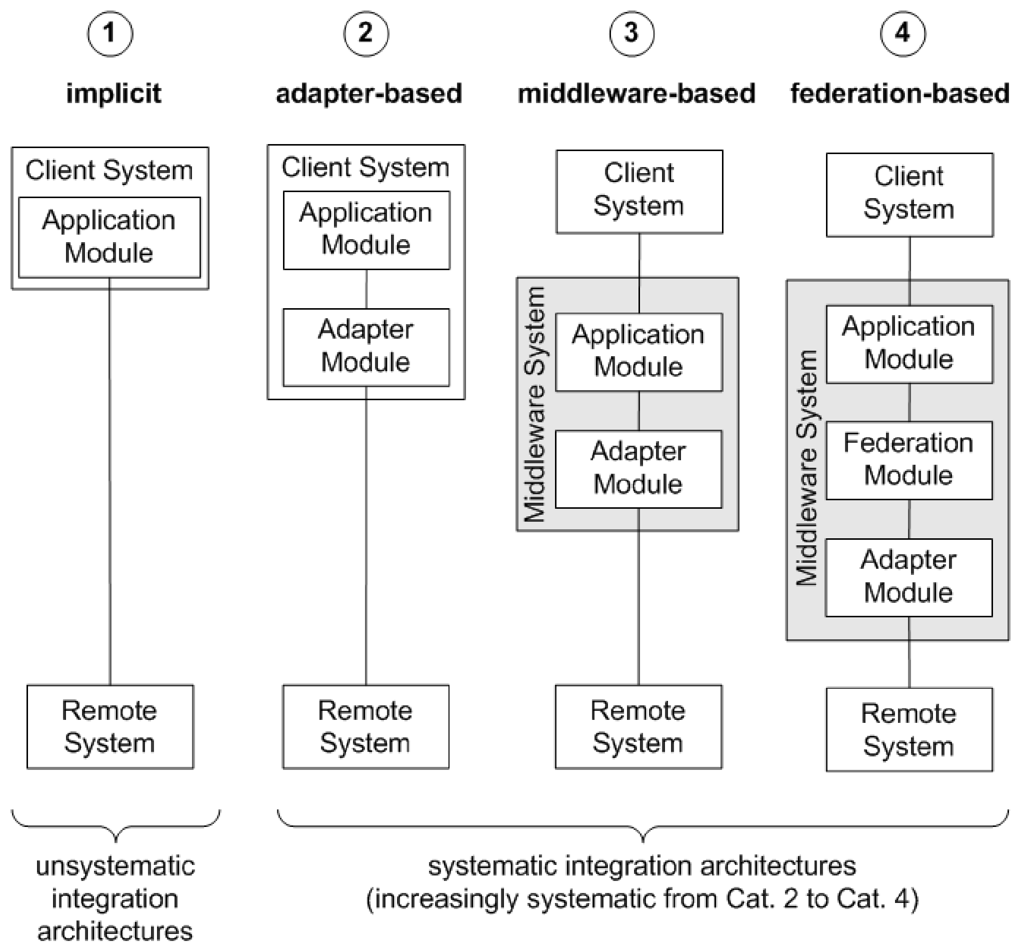


Figure 2.1: Integration Architecture Categories.

The second category is the **adapter-based integration architecture**, which makes use of an adapter module to perform an integration task. The adapter module can be used by different client systems to access the targeted remote system. For example, we evolve the decision support application and now put all PDM RPC calls into a separate module. The resulting *adapter module* offers an interface for the decision support system and hides from the RPC details. The *application module* of the decision support application (*client system*) then always uses the *adapter module* to access the PDM system (*remote system*). If a second client system wants to access the PDM system too, it uses the same adapter module, e.g. as a library, and calls the PDM operations via the interface of this module independent of any RPC details. Such adapter modules can be proprietary and then are used for only few applications within a specific application domain. Or they can comply to an explicitly specified adapter technology so that they can be in wide-spread use, e.g. JDBC drivers for uniformly accessing SQL DBS.

The third category is the **middleware-based integration architecture**. It includes the use of a middleware system that hosts adapter modules and that provides services for handling and executing them. For example, we further evolve our decision support system and add a Java EE application server (*middleware system*) to the scenario. The decision support application now runs as a Java EE application (*application module* plus a web browser as the *client module*) consisting of EJBs and JSPs and it relies on a PDM J2EE connector (*adapter module*) that integrates the PDM system (*remote system*) into the application server. The PDM J2EE connector allows to perform operations on the PDM system according to interaction patterns defined by the J2EE connector standard. These interaction patterns define how to access and integrate remote systems via a Java EE application server. Moreover, the J2EE connector standard also defines contracts between Java EE application server, J2EE connectors and Java EE applications such as connection pooling, thread management, authentication or transactions. The goal of a middleware system is to provide an environment where adapter modules can be handled and executed and where the middleware system provides additional services for adapter modules and application modules so that application modules and adapter modules do not need to implement them on their own. For example, the J2EE connector standard defines a connection pooling mechanism so that connections between connector and remote systems are handled and controlled by the application server. Otherwise, each application would have to handle these connections explicitly on its own.

The fourth and most sophisticated integration architecture category is the **federation-based integration architecture**, which offers a facility to combine data and operations of different remote systems in the middleware. This extends an integration task by a federation task: *adapter modules* allow to uniformly access remote systems and to homogeneously deal with their data and operations, and a *federation module* provides the facility to combine the retrieved data and the executed operations in a uniform manner, e.g. in terms of a homogeneous interface or a global schema. For example, we again evolve our decision support system and we now additionally require access to the data of an object-oriented database system

(OODBS), which we have to combine with the data that we can already retrieve from the PDM system. Thus, we have to implement some specific federation logic that performs this combination step, which we would do in the *application module* so far. However, this combination step is typical for integration tasks where more than one remote system is involved. Therefore, it is desirable to extract and generalize the federation logic and to put it into a separate *federation module* that can be used by different *application modules* with different *adapter modules* and *remote systems*. For example, a federated SQL DBS can process SQL queries that refer to different foreign tables which in turn refer to data and operations in different remote systems. The new version of our decision support system (*client system*) then can issue a federated SQL query, which refers to two foreign tables, one that represents the *get_price* operation in the PDM system and one that represent an object type in the OODBS. The SQL-DBS engine (*federation module*), resolves both foreign tables by issuing corresponding calls to the SQL wrappers (*adapter modules*), which in turn execute the requested operation in the PDM system (*remote system*) and retrieve the requested data in the OODBS (*remote system*). The SQL wrappers transform the retrieved data into tabular format and the SQL engine joins both tables for the final result. The decision support application receives this result and does not need to care about how the combination in the federation step works.

Our virtualization approach can potentially reuse adapters (*adapter modules*) of integration architecture categories 2, 3 and 4, i.e. it can potentially employ any systematic integration technology.

2.2 Adapter Technologies

The basic concept of an adapter is shown in Figure 2.2. There are two interaction categories: executing a request and receiving a response. Combinations of them result in interaction patterns, e.g. see the interaction patterns in WSDL [CMRW07] or REST [Fie00]. The most well-known interaction pattern is that of request-response, i.e. a request that yields a response. The request interaction is initiated by a client, e.g. the client of a middleware system (see Figure 2.2). The client issues a request that is specific to the middleware, e.g. an operation call in case of an application server. The adapter then has to transform the middleware-specific request into a suitable request for the remote system, e.g. an SQL query in case of a DBS. The response interaction works the other way round. In our example, the DBS processes the SQL query, creates an SQL result set and returns it to the adapter. The adapter transforms the remote system-specific response into a middleware-specific response, e.g. some Java objects, and transfers them to the middleware system.

There are many industry standards that aim at establishing standardized integration architectures. There are much more integration products and software suites providing integration technology solutions. There are numerous specific in-house integration approaches that are heavily influenced by special requirements of local application domains and IT environments. And there are lots of integration

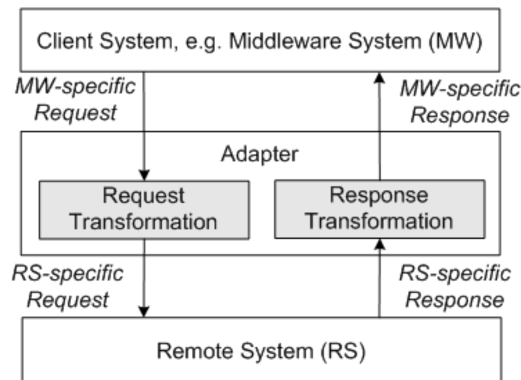


Figure 2.2: Basic Concept of an Adapter.

architectures that stem from research and academia and that contribute to the advent of integration technologies. Hence, adapter technologies can be quite different in their details since adapters are employed in different middleware systems and IT environments, they integrate different kinds of remote systems and they are used in different contexts for different purposes. Similarly, different middleware technologies are used in different environments and scenarios for different purposes. Thus, middleware systems and their associated adapter technologies may significantly differ in terms of applied technologies, programming languages, architecture styles, data models, access patterns, intended application domains, usage purposes, and so on. Therefore, there is a considerable number of adapter technologies and a company possibly has up to hundreds or even thousands of adapters for different middleware systems to integrate diverse remote systems and software services.

Next, we give a brief description of prominent examples of standard integration technologies to give an idea of the ubiquity of integration technologies in our today's IT landscape. Prominent examples of standard integration technologies (and the supported integration architecture categories as shown in Figure 2.1) are:

J2EE Connector Architecture (Cat. 3) The J2EE connector architecture is part of the Java EE platform and is tightly coupled with a Java EE application server that handles J2EE connectors. The standard defines a contract between the application server and the connector, based upon the service provider interface (SPI), which is concerned with tasks such as connection pooling, life-cycle management, security management, work management (thread management), and transaction management [Sun03]. The common client interface (CCI) is the basis of the contract between an application component in the application server and a connector so that the application component knows how to access the connector and how the connector behaves.

SQL Management of External Data (Cat. 4) The SQL standardization also comprises a part that is concerned with accessing external data sources. It is called SQL management of external data (SQL/MED) [SQL08]. The standard defines

how a DBS can act as a middleware system that uses adapters, i.e. SQL wrappers, to integrate data from external data sources. An external data source does not necessarily have to be a DBS, but it can be any remote system that delivers some data. A DBS is thereby extended to an FDBS that is able to cope with heterogeneous software systems.

Web Service Architecture (Cat. 4) The Web service architecture defines and relies on standards such as XML, XML schema, WSDL, SOAP, and UDDI [BHM+04]. Web services basically provide application-specific interfaces that are specified by means of a common meta model, i.e. WSDL. Each Web service is related to a specific implementation by means of a binding. The service implementation is not determined by the Web service architecture. However, this is the actual integration task that a middleware system performs by means of an adapter. Therefore, an Enterprise Service Bus (ESB) handles Web service requests and provides a middleware infrastructure and an adapter technology to realize Web service implementations.

Java Business Integration (Cat. 4) The Java Business Integration (JBI) initiative of the JCP is a specification that attempts to establish a standardized integration platform as a service-oriented architecture [Sun05]. The integration platform relies on Web service standards such as WSDL, SOAP or BPEL and supports a message-driven component model. The JBI component model obeys defined contracts (SPIs) and allows to plug-in and use different JBI components to build integration solutions. The JBI components that are concerned with accessing remote systems are called binding components. They represent the adapters of the JBI middleware.

Common Object Request Broker Architecture (Cat. 3) The Common Object Request Broker Architecture (CORBA) is a standard that aims at providing interoperability between different software systems. The CORBA standard defines an object request bus (ORB) that allows to remotely execute objects. Each object needs an interface that is commonly defined by means of the interface definition language (IDL) [OMG08]. CORBA objects are registered with the ORB in a namespace so that other CORBA objects can access them. Therefore, the ORB infrastructure is the middleware system and the CORBA objects can act as adapters that access remote systems.

Remote Procedure Call (Cat. 2) Remote Procedure Call (RPC) is a very basic means of remote communication that allows to execute operations on other computers and thereby can build distributed systems [Whi75].

Java Message Service (Cat. 3) Message brokers employ message adapters, e.g. JMS providers, that receive and submit messages and that transform messages into

suitable calls to software systems and vice versa. Message Queuing, e.g. the Java message service (JMS) [Sun02], is the basis of message brokers. It defines message exchange patterns and how messages are structured (header and body).

Java Database Connectivity (Cat. 3) The Java database connectivity (JDBC) defines adapters that access data sources. JDBC offers an interface to locally or remotely connect to a DBS and to submit SQL statements to the DBS [Sun06b].

.NET framework (Cat. 4) The .NET framework provides a broad range of services and connectivity with extensive integration capabilities, e.g. ADO.NET and Web services, and it serves as the basis for applications and other software on the Windows operating system [Mic10].

Industry products often, but not necessarily, rely on standards such as the ones mentioned above. But even if they do so, vendors usually evolve their products to provide additional functionality or other proprietary extensions. In the end, such products often are incompatible to each other even if they originally relied on the same standard. This kind of incompatibility problem is typical for products that are based on standards, especially if standards are developed in parallel to the products. For example, SQL, C or CORBA are prominent victims of this procedure. This becomes even worse if there are competing vendors or organizations that follow different interests so that they finally create competing standards that lead to further incompatibilities. There also are competing Web service approaches and standards, e.g. for Web service coordination, Web service choreography or Web service orchestration. There even are competing organizations such as OASIS and W3C that publish Web service standards. Often, newer integration products more or less employ Web services, but not all of them do so. Older integration products don't, but they are still in productive use. Especially, specific in-house developments may not support known standards and further contribute to the diversification of integration approaches. Research and academia also contribute to this diversification. Well-known research prototypes of integration technologies and integration architectures that also influenced the development of commercial products are, for example, TSIMMIS [CGMH+94], Garlic [RS97], Information Manifold [Lev98], XWRAP [LPH00], Nimble [DHW01], or Denodo [PRÁ+02]. Moreover, all products, architectures and systems concerned with integration functionality employ some integration technology, usually only one kind, seldom few and therefore they are incapable of dealing with other integration technologies' adapters. All these characteristics strongly contribute to the diversification and fragmentation of IT landscapes concerning their integration capabilities and thereby consequently lead to problems such as as the one exemplified by our initial integration scenario in Chapter 1.

Integration technologies and adapter technologies can be divided into different categories according to their purpose, their employed technologies, etc. Our integration management approach, i.e. the VT, considers adapter technologies in general

and deals with them from a global perspective. Global treatment of adapter technologies basically has to take into account the two opposite processing paradigms: *data-oriented processing* and *operation-oriented processing.* An operation-oriented request execution can be an API call, an RPC, an object method execution, a message submission, etc. A data-oriented request execution can be a complex request such as an SQL query or an XQuery or it can be a bulk read or write operation or other complex data-oriented operations based upon them. **Data-oriented adapter technologies** and **operation-oriented adapter technologies** are quite opposite to each other in terms of architecture style, operation model, access patterns, application domains or usage purposes. Therefore, they place different requirements on our integration management approach in terms of the data model that we use as the global data model in the virtualization tier (Chapter 3) and in terms of how the virtualization tier can be accessed (Chapter 4). Moreover, our experiments show that the different paradigms of these two adapter technology categories can heavily influence the efficiency of request executions (Chapter 7).

In this work, we use two adapter technologies to extensively evaluate our integration management approach: J2EE connectors and SQL wrappers. **J2EE connector technology** is a representative of operation-oriented adapter technologies and **SQL wrapper technology** is a representative of data-oriented adapter technologies. Both adapter technologies and corresponding middleware systems are opposite to each other in terms of the employed paradigms and technologies. An FDBS offers a built-in integration engine to execute federated SQL queries including an optimizer that provides for an efficient execution. The Java EE application server does not offer such a functionality by virtue, but each Java EE application must provide this functionality on its own (or via a suitable framework). The FDBS and the SQL wrappers execute SQL requests and deal with SQL data, i.e. tabular data. The application server can execute enterprise Java beans (EJBs) that implement application-specific data structures in terms of Java objects and the J2EE connectors use Java objects too. Consequently, FDBS clients issue SQL requests whereas Java EE clients have to rely on proprietary operations that are implemented by Java objects. Moreover, J2EE connectors and SQL wrappers rely on different server functionality, e.g. connection management, transaction management, security management, etc. and they offer different access mechanisms, e.g. SQL wrapper can negotiate about SQL query fragments, J2EE connectors can also handle incoming message calls. In summary, SQL wrappers and J2EE connectors and their respective middleware technologies represent diverse and even opposite technologies and they therefore provide a representative basis to show the applicability and the efficiency of our integration management approach.

2.3 Integration Issues

Let us come back to our initial example in Chapter 1 (see Figure 1.1) and let us take a closer look at it. The conventional solution is to develop the file server J2EE

connector and the CRM DBS J2EE connector from scratch as shown in Figure 1.2 since we only have the file server SQL wrapper and the CRM DBS SQL wrapper. This simply comes from the fact that there exists a number of different middleware systems and much more different remote systems so that the number of adapters resulting from all possible combinations of a middleware system and a remote system is quite large. This in turn means that only a small portion of all potential adapters do actually exist. Most of them have to be developed from scratch if they are needed as it is the case for the file server J2EE connector and the CRM DBS J2EE connector.

The problem of this approach is that developing an adapter generally is complex and error-prone. There are three main issues. First, the adapter developer has to know about the architecture and the different concepts of the adapter technology such as adapter framework, programming model and application programming interfaces (APIs), data model and data representation, processing model and communication protocols, error model, quality of service requirements, etc. Second, the adapter developer has to know about the middleware system that is used to deploy and execute the adapter. The middleware system additionally comprises concepts and mechanisms that are complementary or even different to the concepts and mechanisms of the adapter technology. Third, the adapter developer has to know about the remote system that the adapter has to integrate. The concepts and mechanisms that belong to the remote system can be quite different to the concepts and mechanisms of the middleware system and the adapter technology.

The different software systems, technologies and concepts that are involved in the development of an adapter strongly increase the complexity of the adapter development process. This imposes high requirements on the knowledge, skills and experience of an adapter developer. A software developer who implements a module in a software project or a software developer who implements a web application usually does not have the required capabilities. Moreover, an adapter is more prone to errors due to the different software systems, technologies and tools that are used for adapter development and that the adapter depends on. Therefore, testing the adapter implementation needs to be more extensive. The adapter implementation has to be tested under different conditions and configurations of the middleware system and the remote system, which requires additional administrative and organizational tasks. Errors can occur in relation to specific configurations and states of the different systems, hosts and operating systems that are involved in processing requests of the integration scenario. Maybe the combination of a specific state in the remote system and a specific state in the middleware system creates an unpredictable exception, maybe the load of the remote system host in combination with the thread pooling mechanism of the middleware server reveals a memory leak on the middleware host, and so on.

Debugging adapters is another issue, which is not as easy to perform as it is for most other software implementations since an adapter usually is not executed in its own operating system process, but as part of the middleware system. Debugging an adapter means debugging the overall middleware operating system process using the

debugging facilities of the middleware system. Moreover, debug-and-fix steps take much longer for adapter implementations than for most other software implementations since the involved software systems are distributed over two or more processes or even hosts and have to be administrated and configured separately.

Another argument against developing a new adapter is that if there already exists an adapter that integrates the desired remote system and if this adapter is an integral part of the existing system environment, it could be difficult if not almost impossible to develop a new adapter for the desired remote system. For example, the CRM application of the sales department is an old, legacy in-house implementation. Client applications use the message broker (MB) and thus the CRM application MB adapter to access the CRM application. The CRM application has been changed several times in some way. There is only outdated documentation and there is no one left who really knows about the CRM application and its internals. The knowledge about the interaction patterns and constraints that hold for the CRM application and its interfaces is coded in the CRM application MB adapter implementation and nowhere else. Additionally, bug fixes and some minor changes of the application logic or even extensions of the original functionality have not been realized in the CRM application itself. Instead, the bug fixes and changes have been realized as part of the CRM application MB adapter since there is personnel that can handle the message broker and its adapter implementations, but there is no personnel left that knows about the internals of the CRM application implementation. There is no reliable documentation about the interaction patterns, constraints, bug fixes and application logic changes that are buried in the CRM application MB adapter implementation and thus implementing a new J2EE connector for the CRM application would also require to substantially re-engineer the MB adapter and then to re-implement and test the modifications in the J2EE connector again. In summary, an adapter can encapsulate semantics such as consistency checking, integrity rules, application knowledge, domain knowledge, knowledge about complex interfaces and access patterns of the underlying remote system.

If we take a more global look at the problem of integrating remote systems into middleware systems, we additionally have to consider the overall costs. We already mentioned that it is not practically feasible to develop an adapter for each remote system and for each middleware system and of course we surely do not need an adapter for each possible combination of middleware system and remote system. However, solving the adapter problem by developing a new adapter each time one is needed places the overall complexity of $m * n$ adapters for m middleware systems and n remote systems (where $n \gg m$). From a company's strategic viewpoint this is an unsystematic way of tackling integration issues. A company just reacts to demands in current integration scenarios, but there is no global planning and no global integration strategy. This further complicates management of IT projects and proper calculation of IT budgets and future expenditures. For example, the HR application of our initial example is extended half a year later to additionally access the CRM application of the sales department. We have a MB adapter since the sales department runs a message broker that integrates most of its software systems.

However, the HR application is Java EE-based and needs a CRM application J2EE connector to access the CRM application (Figure 2.3). The consequence is that we have to develop the CRM application J2EE connector from scratch. Another

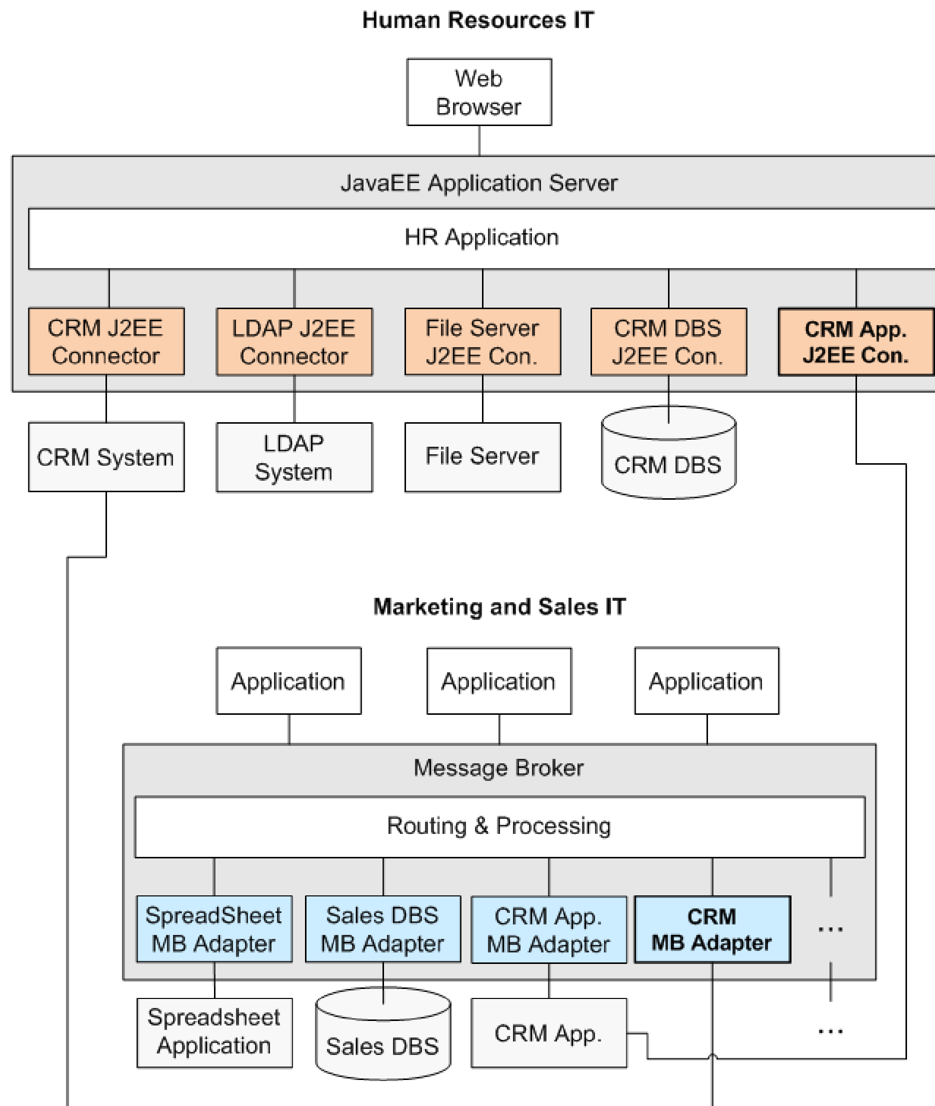


Figure 2.3: Extended Integration Scenario.

process of the company changes one year later and now the CRM system of the HR department has to be integrated into the sales department IT infrastructure. This means that we have to develop a CRM MB adapter from scratch (see Figure 2.3). And so on. In that way each integration issue is unsystematically solved, on its own and in a local, isolated situation, which is most likely sub-optimal from a global perspective.

Additional complications can occur when a remote system changes. Changes in a remote system mean upgrading the system to a new version, modifying the API or other characteristics of the system, which enforce modifications in the integrating

adapter. If there is only one adapter for that remote system and if this adapter is reused by other middleware systems and applications, we have to maintain and modify only this adapter. If we however created several other adapters for the same remote system, but for other middleware systems, we have to maintain and modify each of them, which means that there are several modification tasks instead of only one. For example, imagine that the CRM system in our example scenario in Figure 2.3 is modified, which means that both the CRM J2EE connector as well as the CRM MB adapter have to be modified.

Considering all these arguments it actually becomes very appealing if it would be possible to avoid developing new adapters. Generally, there are two means so far that could avoid developing new adapters from scratch: software generation and software reuse.

2.4 Generation and Reuse

Adapters are valuable because their development is expensive and because their value is gradually increasing during the maintenance and the evolution they undergo over time. There are numerous adapter generation approaches that usually provide some kind of abstract high-level specification language to generate adapters for remote systems, e.g. [BB97, BBH+08, BCG+05, GAD+05, GRVB98, HGMN+97, LHB+99, RPÁ+02, BCMP13]. However, all these approaches are seriously limited by their high abstraction level, which heavily restricts the addressed remote systems. Such specification languages either specify only parts of adapters or they do not support a wide range of interfaces, operation models, data models, error handling models, etc., but address only a specific application domain or scenario, e.g. web sources or SQL sources. Thus, approaches that aim at semi-automatically generating or customizing adapters are only suitable for restricted scenarios, but not as generic solutions. Consequently, adapters are usually developed in a complete development cycle so far.

Software reuse is another means that avoids complete software development cycles. Software reuse has a long tradition and ranges from the use of functions to the concept of software components. An overview of software reuse can be found in [Kru92, FT96, GAO09, GAO95]. Reuse of adapters sometimes is the only appropriate way if an adapter is the only available access mechanism that contains undocumented or lost knowledge about interaction patterns and constraints of the integrated remote systems. Maybe even bug fixes and application logic changes of the integrated remote system are realized in the adapter, but not in the remote system. Moreover, adapters generally perform the same task, i.e. integrating remote systems. Integration of a remote system by means of two different adapter technologies yields two different, incompatible adapters, but both adapters access the same remote system and the same interfaces in the same or at least in a similar way. Both adapters process requests, transform them into corresponding remote system access and transform responses to the adapter technologies' formats, models and

interfaces, respectively. This is different for both integration technologies, but the methodology of processing requests and responses and the way the remote system is accessed is similar for both adapters. In that sense, adapters perform similar integration tasks. Consequently, if we have an existing adapter that already integrates the desired remote system, we should reuse this adapter instead of developing a new one. The existing adapter has already been implemented and tested, it already is in productive use, and it has become a reliable and trusted part of a running system. If we can make use of this asset, why should we go a costly and laborious way and realize a similar integration task again?

The problem with reusing adapters is that adapters of different integration technologies are incompatible to each other so that we cannot directly reuse them on other middleware platforms. For example, we cannot use the CRM application MB adapter in Figure 2.3 with the Java EE application server because the Java EE application server employs different concepts, technologies and mechanisms than the message broker does, i.e. different management and configuration, different programming model and APIs, different data model and data representation, different processing model and communication protocols, different error model, different quality of service requirements, and so on. In other words, the message broker and its adapter technology is incompatible to Java EE application servers and J2EE connectors and vice versa.

In summary, existing approaches that generate adapters are not applicable to most integration issues and direct reuse of adapters is not feasible at all. Therefore, we propose a virtualization tier that is based on integration management technology and that provides a generic and systematic means to indirectly reuse diverse adapters.

2.5 Adapter Virtualization

Typical integration scenarios as exemplified by our example integration scenario show an inherent need for uniformly and systematically using different adapter technologies. This need can also be perceived in existing and ongoing technology and product evolution. Technology examples are Java EE [Sun06a], which defines application server extensions for Web services and CORBA interoperation. More and more middleware systems such as FDBSs or message brokers support Web services. IBM's WebSphere Application Server is a Java EE application server, but is also able to natively access WebSphere Business Integration Adapters, which are part of a message broker platform. Stonebraker [Sto02] discusses six classes of middleware systems and concludes that middleware systems of different classes increasingly tend to overlap in their functionality. Vinoski [Vin03] discusses the need for a "middleware for middleware" and the difficulties and limitations that are imposed even if Web service technology is employed. Such indicators show that there is a need for uniformly and systematically using different adapter technologies, but existing middleware extensions such as the aforementioned ones only support one or at most

few adapter technologies, respectively, and they are proprietary and unsystematically designed. To the extent of our knowledge there is no systematic approach that provides uniform and systematic access to different adapter technologies in a general manner although it would drastically influence IT landscapes and significantly increase cost-savings.

We designed a systematic integration management approach that can deal with different adapter technologies and that shields developers from integration tasks. Our approach employs a virtualization tier (VT) that allows to reuse adapters in different integration scenarios and with different middleware platforms. The VT virtualizes adapters and thereby uses and executes them as their respective middleware platforms would do. Virtualizing means that we provide the capability to uniformly handle and access different types of adapters so that application developers do not need to care about integration tasks, but can work independent of integration issues. For example, we deploy the CRM application MB adapter into the VT so that the HR application (cf. Figure 1.3) can access the CRM application by accessing the CRM application MB adapter in the VT (as sketched in Figure 2.4). Hence, there is no need for developing a new CRM application J2EE connector. The VT thereby alleviates and solves the drawbacks and problems of developing new adapters.

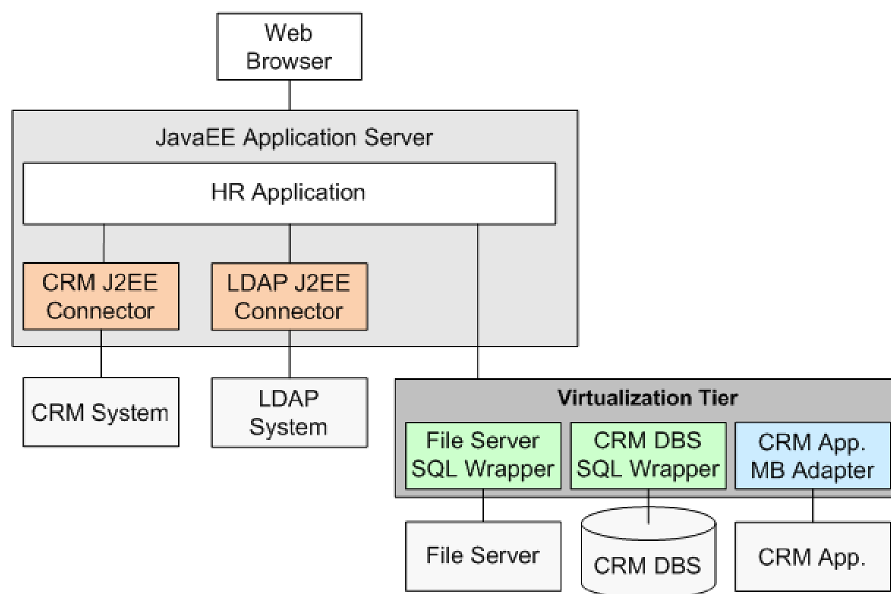


Figure 2.4: Reuse of the CRM App. MB Adapter.

Figure 2.5 shows the basic architecture of the VT. Each adapter technology needs an adapter manager that is responsible for managing and properly executing adapters of that adapter technology. Therefore, an adapter manager and a middleware system that both handle adapters of the same adapter technology have to implement the same core functionality. It is even possible to use that part of a middleware system implementation that is concerned with management and execution of adapters as a code basis for a corresponding adapter manager. This is actually

recommended since this functionality constitutes a considerable part of an adapter manager.

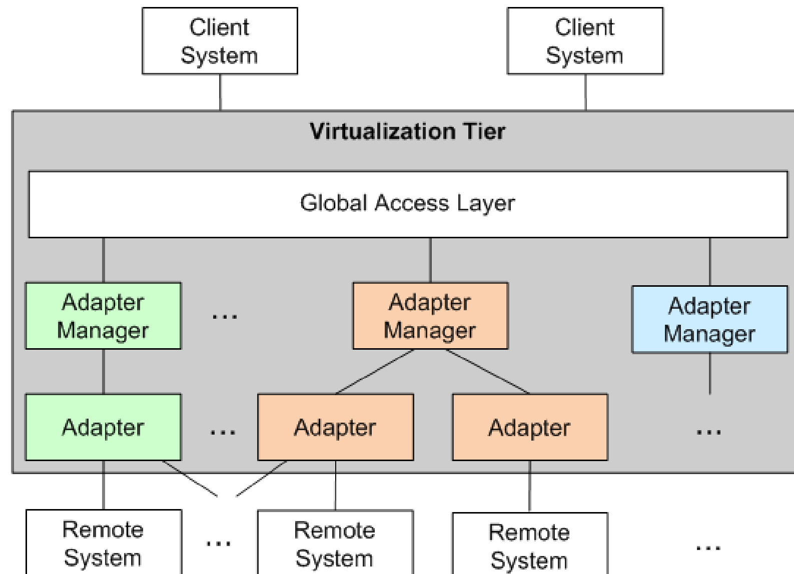


Figure 2.5: Virtualization Tier Architecture.

The global access layer of the VT provides uniform access to the different adapters. Figure 2.6 shows how the VT-based solution of the integration example in Figure 2.4 looks like. The VT employs the SQL wrapper manager to handle SQL wrappers, e.g. the file server SQL wrapper and the CRM DBS SQL wrapper, and it employs the MB adapter manager to handle MB adapters, e.g. the CRM application MB adapter. The J2EE connector manager is responsible for managing and executing J2EE connectors and so on. The HR application now can use the VT J2EE connector to access the VT, which in turn uses the different adapters that are deployed in the VT. The benefit of this solution is that both SQL wrappers and the MB adapter can be completely reused whereas the conventional solution shown in Figure 2.3 requires to develop three J2EE connectors from scratch. The message broker of the sales department uses the VT in the same way (see Figure 2.7): it employs the VT MB adapter to access the VT and thereby all adapters that are deployed in the VT, e.g. the CRM J2EE connector.

An important aspect of the VT approach is that existing middleware systems do not have to be modified and that the operation of existing applications and processes is not affected. The VT enhances middleware systems in a non-invasive manner offering an additional means of accessing remote systems. The enhancement is achieved by using a middleware's native adapter technology to access the VT, i.e. a VT middleware adapter. For example, the Java EE application server in our example scenario in Figure 2.6 uses a VT J2EE connector to access the VT. Figure 2.8 abstractly shows how existing and new middleware applications coexist. On the one hand, existing applications only use adapters in the middleware system that

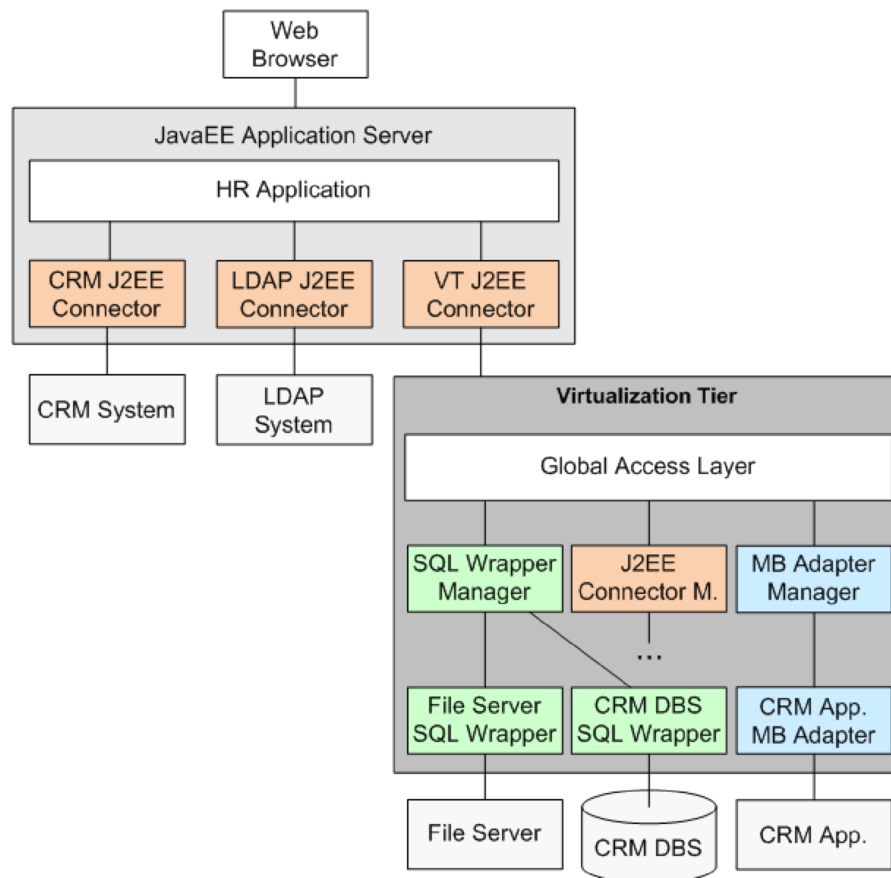


Figure 2.6: VT-Based Solution.

are directly accessing remote systems. On the other hand, new applications can additionally include the VT in their processing and additionally use other adapters.

Another benefit of the VT approach is that it increases the flexibility of an IT infrastructure since future changes and requirements concerning such integration tasks can be solved with the VT again. Abstractly seen, the VT can be considered as a middleware multiplexer that allows a middleware system to access any adapter that is deployed in the VT. If m middleware platforms have to access n remote systems *without* using the VT, we would potentially need $m * n$ adapters as shown in Figure 2.9. The VT reduces this complexity to $m + n$ adapters due to the uniform access that allows any middleware platform to use any adapter in the VT as shown in Figure 2.10 (m adapters for accessing the VT and n adapters for accessing the remote systems). Put in other words, a conventional integration approach that relies on a specific middleware system and a specific adapter technology requires a specific adapter to integrate a remote system whereas the VT allows to use the same middleware system, but any adapter that is available for that remote system. The VT is a systematic and integration technology-independent approach that is open for any middleware platform or application that wants to benefit from the VT.

A basic idea of the VT is that of a multiplexer that allows diverse middleware

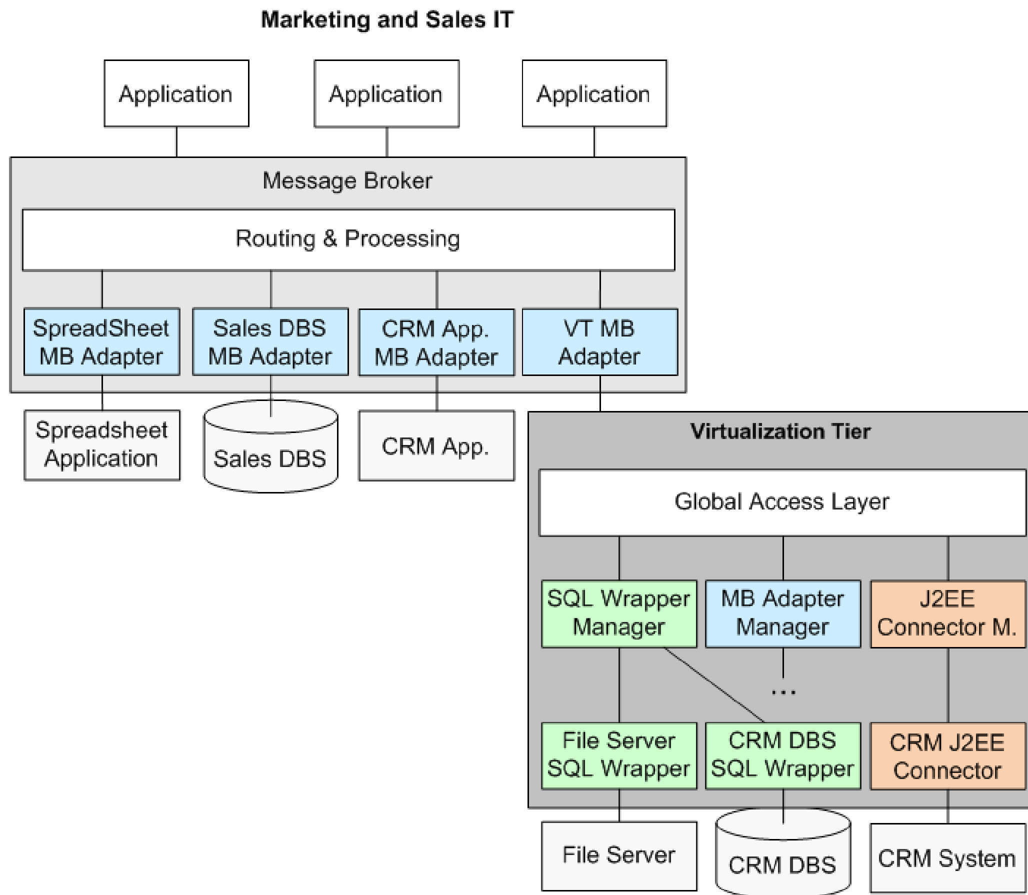


Figure 2.7: VT-Based MB Solution.

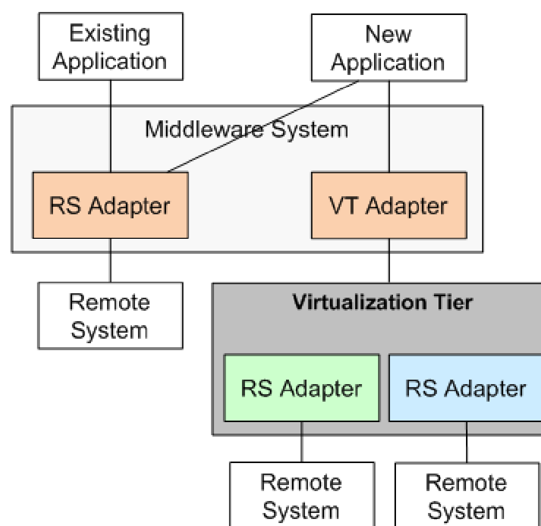


Figure 2.8: Middleware Enhancement.

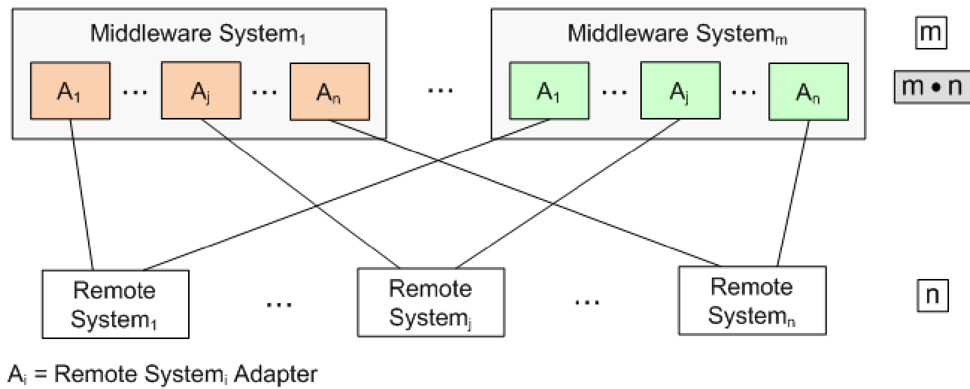


Figure 2.9: Conventional: Potentially $m * n$ Adapters Required.

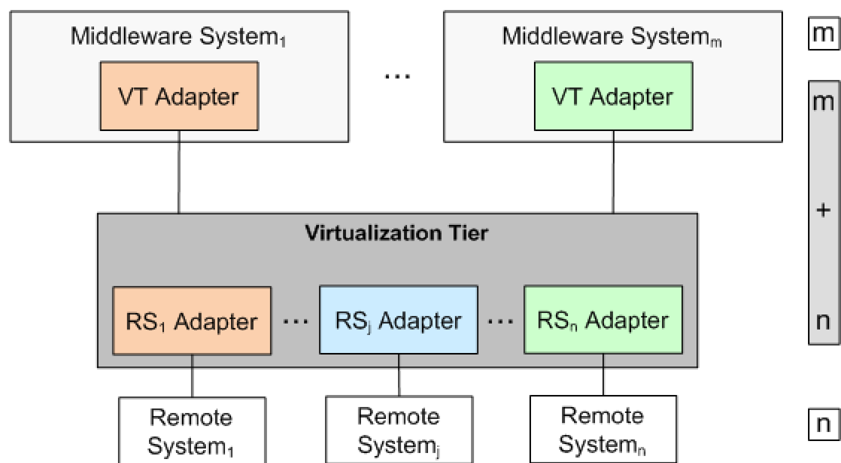


Figure 2.10: VT as Multiplexer: Only $m + n$ Adapters Required.

systems to access diverse adapter technologies. Therefore, the VT is executed as an independent and general-purpose service, i.e. in a separate server process, most suitably on a dedicated host. Another consequence from the central role that the VT plays is that we have to consider scalability issues. There are different distribution patterns conceivable for the VT to enable scalability. We did not perform an in-depth evaluation of different scalability approaches for the VT since techniques and methods for achieving scalability are well studied and can be analogously applied to the VT. A solution that is quite easy to achieve is to create VT instances on different hosts (see Figure 2.11). We only have to extract the VT repository where the VT object configurations and other configuration information is stored. The VT repository then runs on a separate host as a service for the different VT instances. The VT instances now access the VT repository for deployment and configuration information via network. Any information from the repository can be cached so that execution time is sped up. The separately hosted VT repository can additionally rely on conventional DBMS technology that employs distribution and replication to further increase scalability.

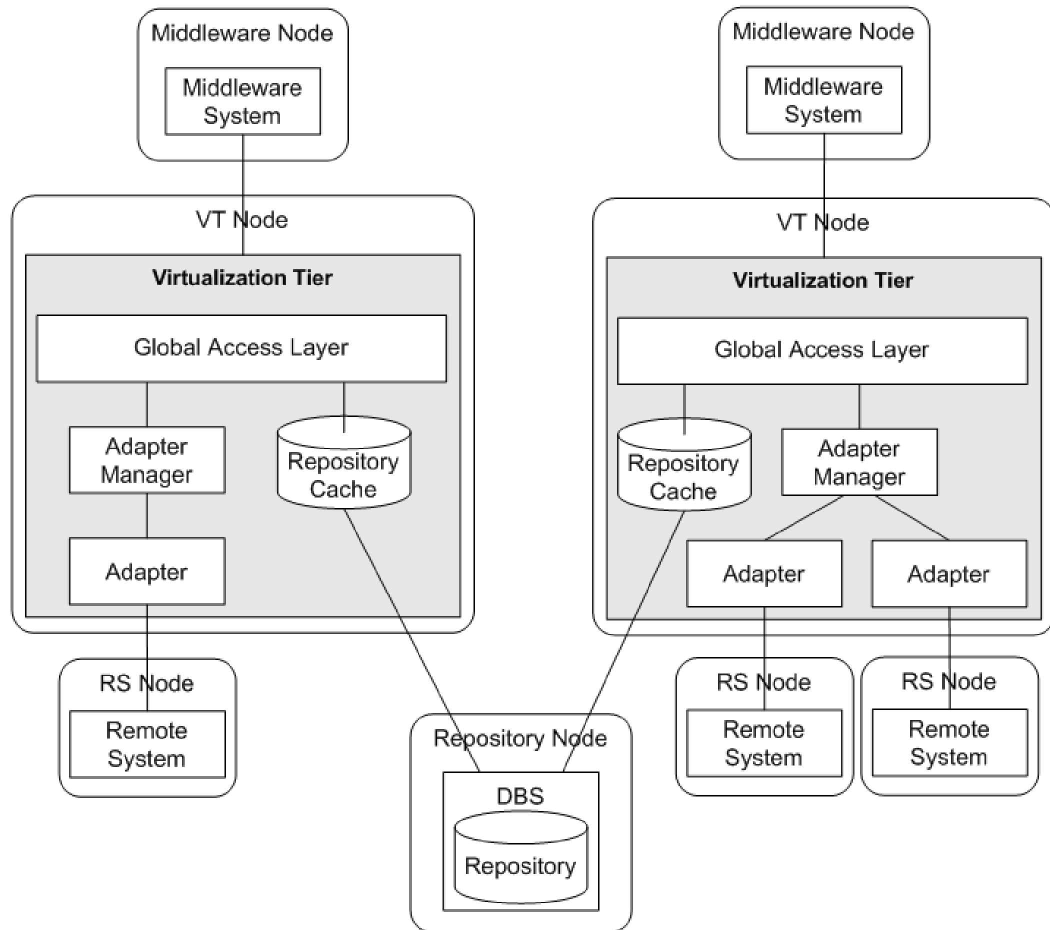


Figure 2.11: VT Scalability.

In the next section, we generalize from the VT approach. We introduce integration management technology and the class of integration management systems that shield from integration tasks and that provide integration independence.

2.6 Integration Management

IT infrastructures use middleware systems and adapter technologies to overcome heterogeneities and to integrate diverse software systems. Applications depend on such integration technologies to fulfill their functionality. Different middleware systems employ different adapter technologies and therefore an IT infrastructure depends on different integration aspects. For example, an IT infrastructure may contain a Java EE application server and J2EE connectors, an FDBS and SQL wrappers, and a message broker and message broker adapters. Applications may use one or more of them and thereby access different middleware systems, different adapters and different remote systems. An IT infrastructure also obeys different organizational, administrative and technical aspects that are involved in integration issues,

e.g. system authorities, system management responsibilities, licensing and software distribution. Integration management technology is a means to systematically deal with these integration issues. Figure 2.12 shows the basic architecture of integration management systems. An **integration management system (IM system)** is a software system that extracts integration issues from an IT infrastructure and its applications and that provides a single point of access, i.e. the *integration abstraction layer*, independent of any *integration tasks* that have to be performed beyond this point. We call the underlying technology **integration management technology (IM technology)**. An **integration task** in this context is a component that realizes one or more steps to interconnect remote system and integration management system and to overcome the heterogeneities between them. An IM system such as the VT provides uniform access means to diverse software systems and shields an IT infrastructure and its applications from integration aspects. We call this **integration independence**†.

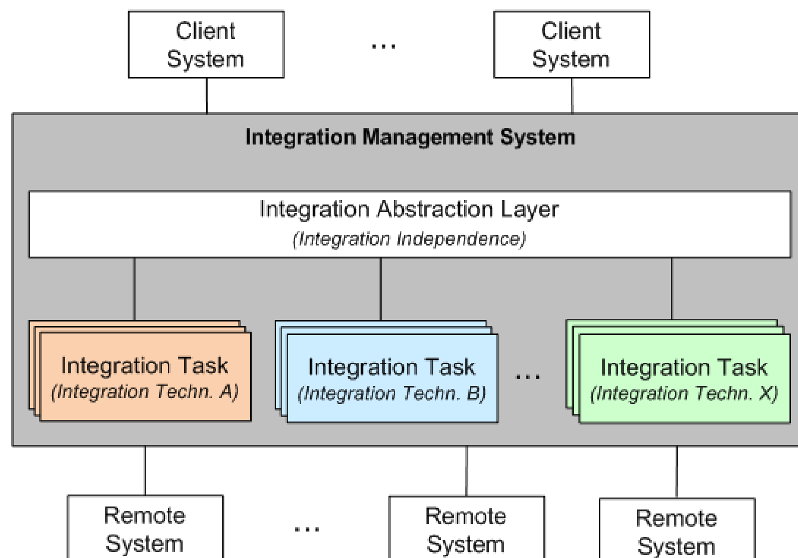


Figure 2.12: Integration Management System Architecture.

On the one hand, a *middleware system* employs an adapter technology to offer an *integration means*. On the other hand, the VT is an *IM system* that handles integration technologies to provide *integration independence*. IM technology abstracts from integration technologies in the sense that an IM system is independent of specific integration technologies. Therefore, the VT cannot be compared to a middleware system or an integration technology. A software system or an application that uses a middleware system depends on the corresponding integration technology. A software system or an application that uses the VT is independent of a specific integration

†The term *integration independence* is differently used, e.g. see [Noe05, McG06, HMKH10, Mil10]. We define the term *integration independence* in the sense of integration handling abstraction as the term *data independence* is defined in the sense of data handling abstraction [Cod70] or as the term *flow independence* is defined in the sense of flow handling abstraction [LR00]

technology and completely shielded from the integration tasks. Consequently, an IM system such as the VT is not intended to replace an integration technology and it can not be replaced by a middleware system and an integration technology. The task of an IM system is to deal with integration issues and other heterogeneity aspects of an IT infrastructure so that the development and maintenance of software systems and applications is alleviated and more homogenized.

In the rest of this work, we discuss and evaluate IM technology by means of the VT as a representative IM system. We take a closer look at the data model (Chapter 3) and at the processing model (Chapter 4) of the VT, we discuss deployment and configuration of integration tasks in the VT (Chapter 5), and we evaluate the applicability (Chapter 6) and the performance (Chapter 7) of the VT.

2.7 Summary

From a global perspective, the problem of integrating remote systems into middleware systems creates the overall complexity of $m * n$ adapters for m middleware systems and n remote systems. It is not practically feasible to develop an adapter for each remote system and for each middleware system because of this complexity and since developing an adapter generally is complex and error-prone. Therefore, we designed a systematic *integration management (IM)* approach that can deal with different adapter technologies and that shields developers from integration tasks. Our approach employs a *virtualization tier (VT)* that allows to reuse adapters in different integration scenarios and with different middleware platforms. The VT virtualizes adapters and thereby uses and executes them as their respective middleware platforms would do. Virtualizing means that we provide the capability to uniformly handle and access different types of adapters so that application developers do not need to care about integration tasks, but can work independent of integration issues. The VT can be considered as a middleware multiplexer that allows a middleware system to access any adapter that is deployed in the VT. The VT thereby reduces the complexity of $m * n$ adapters to $m + n$ adapters.

Chapter 3

Data Model

The adapter managers of the VT are responsible for properly managing and executing adapters (see Figure 2.5). The global access layer has to provide a suitable data model that can uniformly deal with data and operations of the remote systems that are integrated by the adapters in the VT.

3.1 VT Objects

The advantage of an object-oriented data model is that it can support data aspects by means of objects as well as behavior by means of object methods. This is a crucial aspect for the VT data model since it has to adequately support different adapter technologies, which may be data-oriented or operation-oriented. Put in other words, we need a means to represent and access remote data and we need a means to represent and execute remote operations. Additionally, we have to be able to access data in terms of bulk operations or complex requests, e.g. in a set-oriented way. Therefore, the VT data model has to support data structures and operations as well as a declarative, set-oriented query language. These requirements lead us to an object-oriented data model that can represent complex data structures as well as behavior and that is suitable for executing data-intensive queries as well as computational operations. The ODMG object model [CBB+00] meets the given requirements. The ODMG standard provides a semantically rich data model that supports object-oriented mechanisms such as inheritance, relationships and behavior and it also offers a declarative query language for processing data-intensive requests. It is programming language-independent so that it can be adopted to different programming languages (by means of so-called "bindings"), e.g. see [ABCB+01] for a discussion of the Java language binding. Moreover, the ODMG standard has been specified and accepted by a large community in industry as well as in academia and it has been successfully used in integration projects, e.g. Garlic [CHS+95], IRO-DB [GGF+95], and Lore [AQM+97]. It evolved and matured in more than 15 years and it has been adopted in other object models, e.g. JDO [Sun10], and software products, e.g. Hibernate [KBA+10]. Hence, the ODMG object model provides a firm basis and is most suitable to serve as the basis for the VT data model. We also use

the associated set-oriented, declarative object query language (OQL) of the ODMG standard as the basis for the VT query language (VTQL).

The ODMG object model is based on the OMG object model which is the object model of CORBA [OMG08]. Object instances obey types, i.e. classes or interfaces, and they comprise properties, i.e. attributes or relationships, as well as operations and exceptions. Operations are executed on behalf of the objects and they can raise exceptions. The VT data model is based on the ODMG object model and defines **VT objects** that can represent remote data as well as remote operations and that can be directly accessed in a navigational manner or by means of complex VTQL requests in a set-oriented, declarative manner. A VT object does not exist on its own, i.e. it does not exist independently, but is always correlated with some remote data or remote operation via an adapter that is deployed in the VT. This in turn means that any VT object access results in a corresponding access on a remote system. Data and operations of remote systems are reflected as VT objects and VT object methods in the VT. For example, a VT object class can represent a relational table or a Java class in a remote system. It is defined by means of a configuration chapter and can have one or more instances that represent data entities in the remote system. For example, a VT object instance can represent a tuple of a relational table or a Java object. The VT object concept is based on the interface and class constructs of the ODMG object model. If it is not necessary to distinguish between a VT object class and its instances or if it is clear from the context what is meant, we just use the term VT object. A **VT object operation** is an operation that is associated with a VT object class analogous to operations in the ODMG data model where an operation is a method of a class or interface. A VT object operation represents an operation in a remote system. If a VT object operation is executed, the corresponding remote operation is executed.

3.2 Semantics

The VT data model can basically represent remote data and remote operations by means of its structural properties, i.e. data structures. This is sufficient to properly access VT objects. The semantics of remote systems that are reflected as constraints on data or as implementations of operations usually cannot be completely represented in the VT (nor in a middleware system). Actually, it is even not necessary to represent any constraints of remote data or semantics of remote operations in the VT since VT objects only act as representations of remote data and remote operations and therefore the VT does not need to enforce constraints. Constraints and other semantics are already automatically enforced by the remote systems and maybe by the adapters that integrate the remote systems. From that point we only need the information about the constraints so that a VT client application can properly deal with VT objects, but we do not need to enforce the constraints itself in the VT. Nevertheless, the VT allows to specify information about the constraints of remote data or remote operations in terms of VT object constraints analogous to

ODMG constraints such as data types or relationships.

Any constraint checking comes at some costs, especially the more complex constraints become. If we redundantly enforce constraints in the VT in addition to the remote system, the processing of a VT request is generally slowed down to some extent. However, if this enforcement detects an error in the VT request, the request is not submitted to the remote system, but immediately rejected by the VT. Otherwise, the VT would not check the constraints of the VT request and it would submit the VT request to the remote system where the error is finally detected and the request is rejected. It would take more time to submit the request to the remote system than to check some constraints in the VT and thus it can be beneficial to pre-enforce constraints in the VT. It is a trade-off whether constraints should be pre-enforced in the VT or not. If constraint violations seldom occur, it is most probably not beneficial to redundantly check them in the VT. The VT administrator has to decide on whether a constraint is pre-enforced in the VT or not, i.e. whether the overhead of pre-enforcing a constraint does pay off. Therefore, constraints defined in the VT can be additionally tagged with *<unchecked>* so that these constraints are not pre-enforced by the VT, but only act for documentation purposes. A good reason why the VT should be able to define and enforce constraints is that the VT can be used to place additional, new constraints, e.g. constraints that cannot be expressed in a remote system or constraints that span multiple remote systems. Finally, it can be very useful to define constraints in the VT for documentation purposes only either as constraints that are tagged as *<unchecked>* or in terms of annotations that are attached to the elements of VT object definitions and that describe the semantics, e.g. "age must be between 18 and 99" for VT object attribute *age* or "this operation sums up the monthly turnovers of the given year" for VT object operation *computeSum(int year)*. These constraints are not enforced in the VT, but they serve VT administrators, VT client application developers and VT users as a global documentation so that they don't have to use specific remote system documentation, if there is someone at all. In that way, the VT can serve as a global documentation repository that is uniformly accessible and that delivers semantic information about the remote data and remote operations that are represented as VT objects in the VT.

3.3 Transactions

Every VT client application can access VT objects only within a transaction so that different VT client applications work isolated from each other. A VT client application must use a **VT transaction** even if a remote system does not support any kind of transactions. If a VT client application, e.g. a middleware system, accesses more than one remote system, the VT transaction is distributed among the participating remote systems. A remote system can either support distributed transactions (*<distributed>*), standard transactions (*<standard>*) or no transactions at all (*<none category="...">*). Remote systems that do not offer transactions are

not intended for competing access patterns (although concurrent access might be possible). Nevertheless, such remote systems can be used by the VT too. They are divided into three categories. The first category (*<none category="read_only">*) are remote systems that do not allow to modify data, i.e. they are read-only, e.g. some kind of data warehouse or a documentation or encyclopedia resource. The VT can access such a remote system without any restrictions, i.e. read-only. The second category (*<none category="single_user">*) are remote systems that are only used by the VT, i.e. there is no concurrent access. However, if the VT transaction rolls back on behalf of the VT client application or if an error occurs in the remote system during the execution of a VT transaction, the remote system is not automatically rolled back. Operations that have already been executed on the remote system must be explicitly compensated, e.g. by the VT client application or another administrative mechanism. It is the responsibility of the VT client application developer to consider such cases in the same way as an application developer would do if he is developing an application that directly operates on this remote system. The third category (*<none category="uncertain">*) are remote systems that are not only accessed by the VT, but by other systems too. Nevertheless, it seems not to be critical (for whatever reason) when two systems concurrently access such a remote system, e.g. a system with slowly changing data or a system that only collects data. For example a booking system shows that there is one place left in the plane, but when you try to book it, it is already booked by someone else or a monitoring system that logs events from other systems.

3.4 Object Identity

The **VT object manager** is part of the global access layer. It is responsible for properly handling VT object instances in the **VT object cache** of the global access layer. The VT object manager loads VT object instances via adapters from remote systems into the VT and it writes VT object instances via adapters from the VT into the corresponding remote systems. The VT object manager is also responsible for properly executing VT object operations via adapters on remote systems. Each VT object instance is uniquely identified in the VT, i.e. each VT object instance can be uniquely addressed and accessed. VT object instances can be compared either as VT object instances or as representatives of remote data entities. The former case compares whether two VT object instances are the same or not. A VT object instance is identified by means of the VT identifier (*vtid*). If two VT object instances have the same VT identifier, they actually are identical, i.e. there is only one VT object instance with this VT identifier. We call this kind of identity **VT identity**. The VT identity only holds for the VT. If we deal with a VT object instance as a representative of a remote data entity, it is not identified by means of the VT identity, but by means of the identity of the corresponding remote data entity. For example, a VT object instance that represents the tuple of a relational table is identified by the primary key of the table. A VT object instance that represents

a Java object could be identified by means of the main memory address where the Java object is allocated or by means of an internal object identifier. We call this kind of identity the **remote identity** of a VT object.

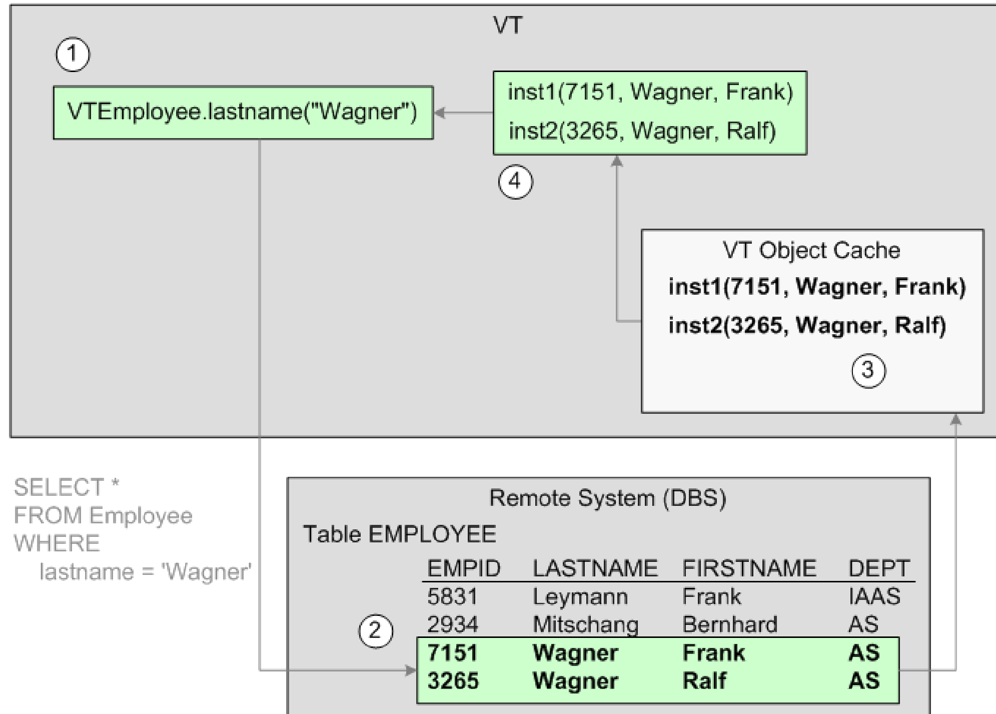


Figure 3.1: VT Object Mapping – First Request.

If a VT object instance has a remote identity, we call it a **unique VT object instance** (and its associated VT object class correspondingly **unique VT object class**). If a VT object instance does not have a remote identity, we call it a **non-unique VT object instance** (and its associated VT object class correspondingly **non-unique VT object class**). The VT uses the remote identity to ensure that a VT object instance is always correlated with the same remote data entity and that a remote data entity is represented only by one VT object instance. Consequently, there cannot be two or more VT object instances of the same VT object class that represent the same remote data entity. For example, Figure 3.1 shows how remote data is mapped to VT objects.* VT object class *VTEmployee* represents a relational table in a DBS. The table is named *Employee* and its primary key is *EMPID*. *VTEmployee* has two parameter sets: *VTEmployee.lastname(String lastname)* and *VTEmployee.dept(String dept)*.† Now, we perform a VT object read operation, i.e. *VTEmployee.lastname("Wagner")* (1), which locates two tuples in the *Employee* table as the requested remote data (2). The DBS returns the data and the VT

*We omit the adapter manager and the adapter since we only want to show the "final" correlation between remote data and VT objects independent of any intermediate steps in the adapter manager and in the adapter.

†Details about VT object operations and parameters can be found in Chapter 4.

object manager allocates two VT object instances of *VTEmployee* (3) and returns them as the result of the request (4). Next, we perform a second retrieval (see Figure 3.2), *VTEmployee.dept("AS")* (1), which locates three tuples in the *Employee* table as the requested remote data. The DBS returns the data, but in this case the VT object manager allocates only one *VTEmployee* instance (3) since the VT object manager uses the primary key values of table *Employee* as the remote identity of *VTEmployee* to determine whether a tuple of this table is already represented as a *VTEmployee* instance in the VT object cache. Indeed, the primary key values *7151* and *3265* already belong to two *VTEmployee* instances and therefore the VT object manager only creates one further *VTEmployee* instance. Finally, the VT object manager returns the requested *VTEmployee* instances (4).

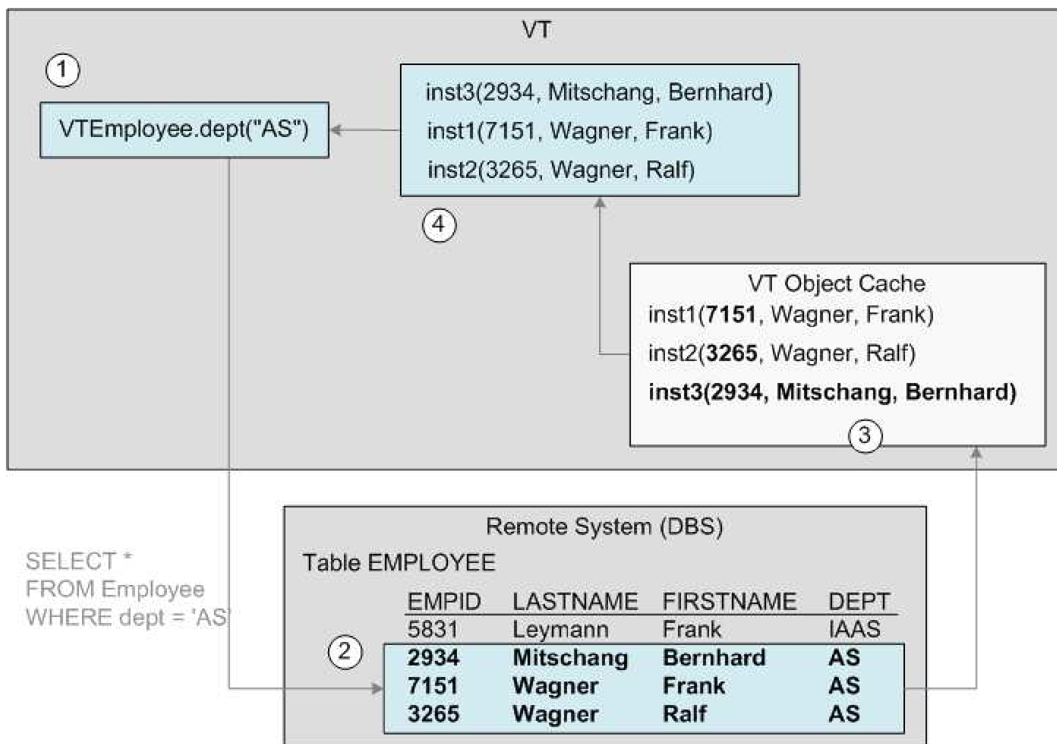


Figure 3.2: VT Object Mapping – Second Request.

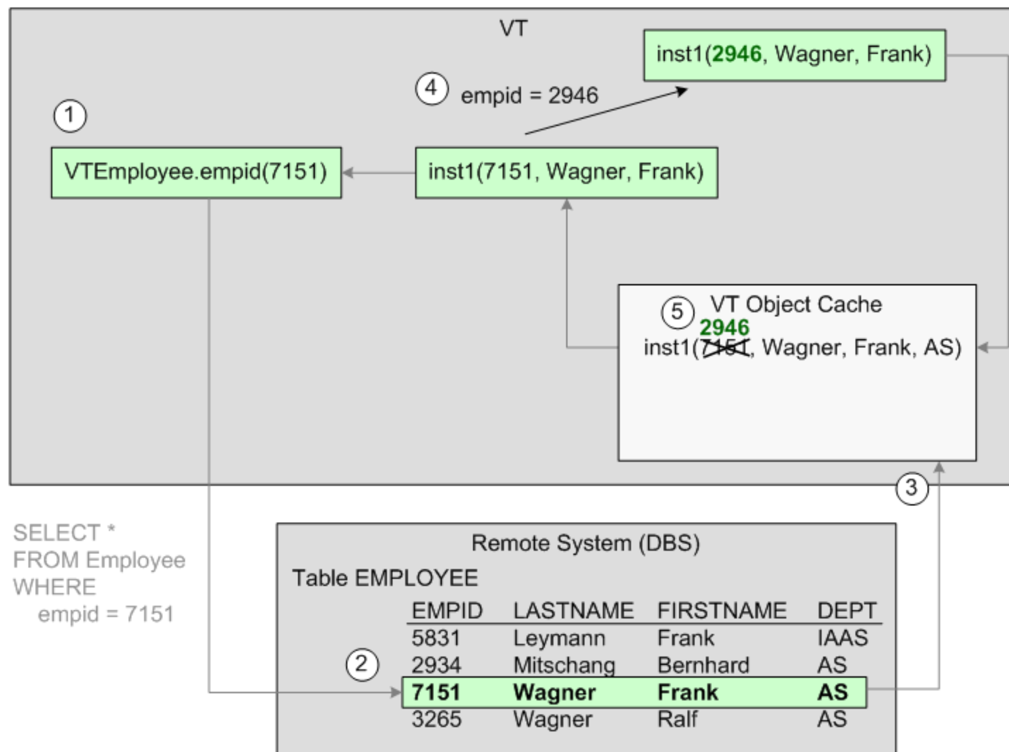
3.5 Non-Uniqueness

Changes of a VT object instance are reflected in the correlated remote data at the end of a VT transaction. For example, we retrieve one *VTEmployee* instance via *VTEmployee.empid(7151)* as shown in the update step in Figure 3.3, (1 - 3). Thereafter, we modify the *id* attribute of this *VTEmployee* instance (4, 5) and commit the VT transaction so that the modified value is written to the remote DBS. This works fine as long as we preserve the correlation between a VT object instance and its associated remote data entity, e.g. a tuple in a table. However, in

this case we changed the value of an attribute that is part of the remote identity, i.e. the *id* attribute, and therefore we cannot any longer correlate this *VTEmployee* instance with the associated tuple in table *Employee.* The solution is to keep the original remote identity of each VT object instance so that changes to the remote identity of a VT object instance can be still properly reflected in the correlated remote data entity. The original remote identity of a VT object instance is also used for properly identifying remote data entities that are retrieved later in the VT transaction. For example, the *id* attribute of *VTEmployee* instance *inst1* changed from *7151* to *2946.* In this case, we did not commit so far, i.e. our transaction is still running. We issue the next request that retrieves all employees of department *AS* (see retrieval step in Figure 3.3): *VTEmployee.dept("AS").* The VT object cache now uses the original *id* values of the VT object instances to determine whether a tuple has already been loaded into the VT earlier. Hence, the VT object cache does not compare the *empid* value of the *Employee* tuple *(7151, Wagner, Frank, AS)* to the current *id* value of *VTEmployee* instance *inst1*, i.e. *2946*, but to it's original *id* value, i.e. *7151*, thus realizing that the tuple is already correlated to *VTEmployee* instance *inst1* although its remote identity value has been changed in the VT (but not committed yet).

The situation becomes more complicated if a remote data entity does not have an identifier, e.g. the tuple of a relational table that does neither define a primary key nor a unique constraint. A remote data entity could be potentially represented more than once in the VT by different VT object instances of the same VT object class. Modifications on two or more of these VT object instances would then cause unexpected results. For example, Figure 3.4 shows how non-unique remote data entities are mapped to VT object instances without considering their non-uniqueness. VT object class *VTPayment* represents the relational table *Payment* in the DBS. There is no unique constraint and no primary key. *VTPayment* has two VT object read operations: *VTPayment.payedBy(String dept)* and *VTPayment.category(String category).* We now want to decrease all payments issued by department *IAAS* by *150* and thereafter we want to increase all payments of category *3* by *20* percent. First of all, the VT transaction uses *VTPayment.payedBy("IAAS")* to read two tuples from table *Payment* (1, 2). The VT object cache is empty and therefore two VT object instances are allocated for the two tuples (3). Finally, the VT transaction decreases the *amount* values of both *VTPayment* instances by *150* (4, 5).

The second part of the VT transaction executes *VTPayment.category(3)*, which reads two tuples from table *Payment*, too (6, 7). The two tuples are mapped to corresponding *VTPayment* instances (8) although one of the tuples has already been read by the previous request, i.e. *VTPayment.payedBy("IAAS").* Now we have four VT object instances representing three tuples in table *Payment.* Tuple *(2934, 900, IAAS, 3)* is represented by two different *VTPayment* instances, i.e. *inst2* and *inst4*. There is no means to determine whether a tuple of table *Payment* is already represented by a *VTPayment* instance or not because table *Payment* does neither have a primary key nor a unique constraint, which would be necessary to uniquely identify a tuple and an associated *VTPayment* instance. Even comparison of all



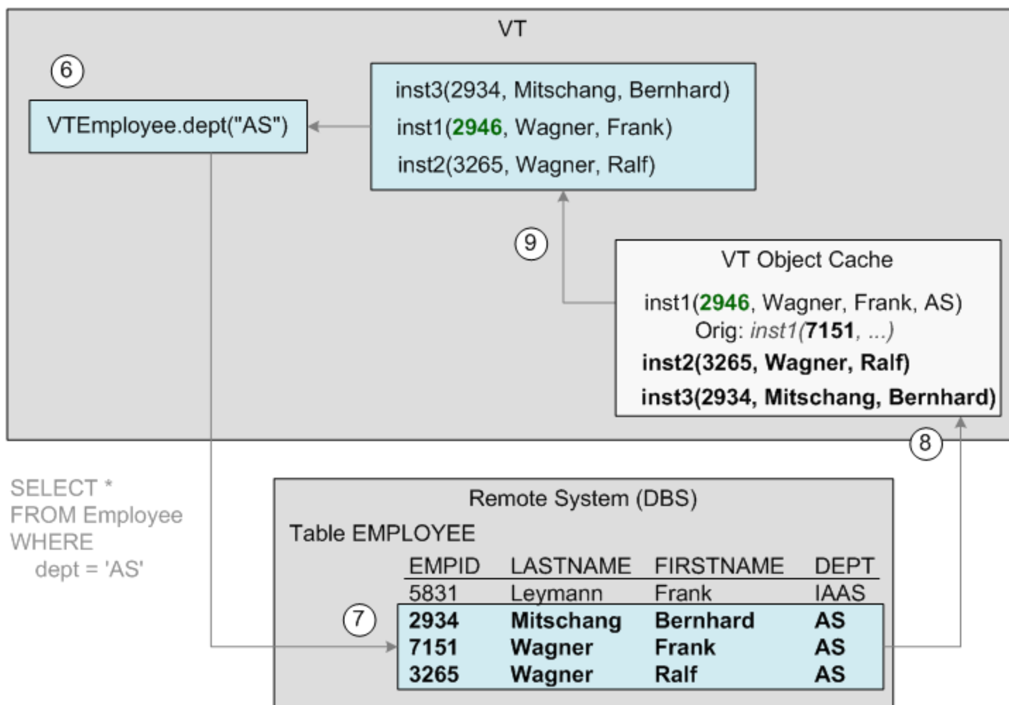


Figure 3.3: VT Object Mapping – Update.

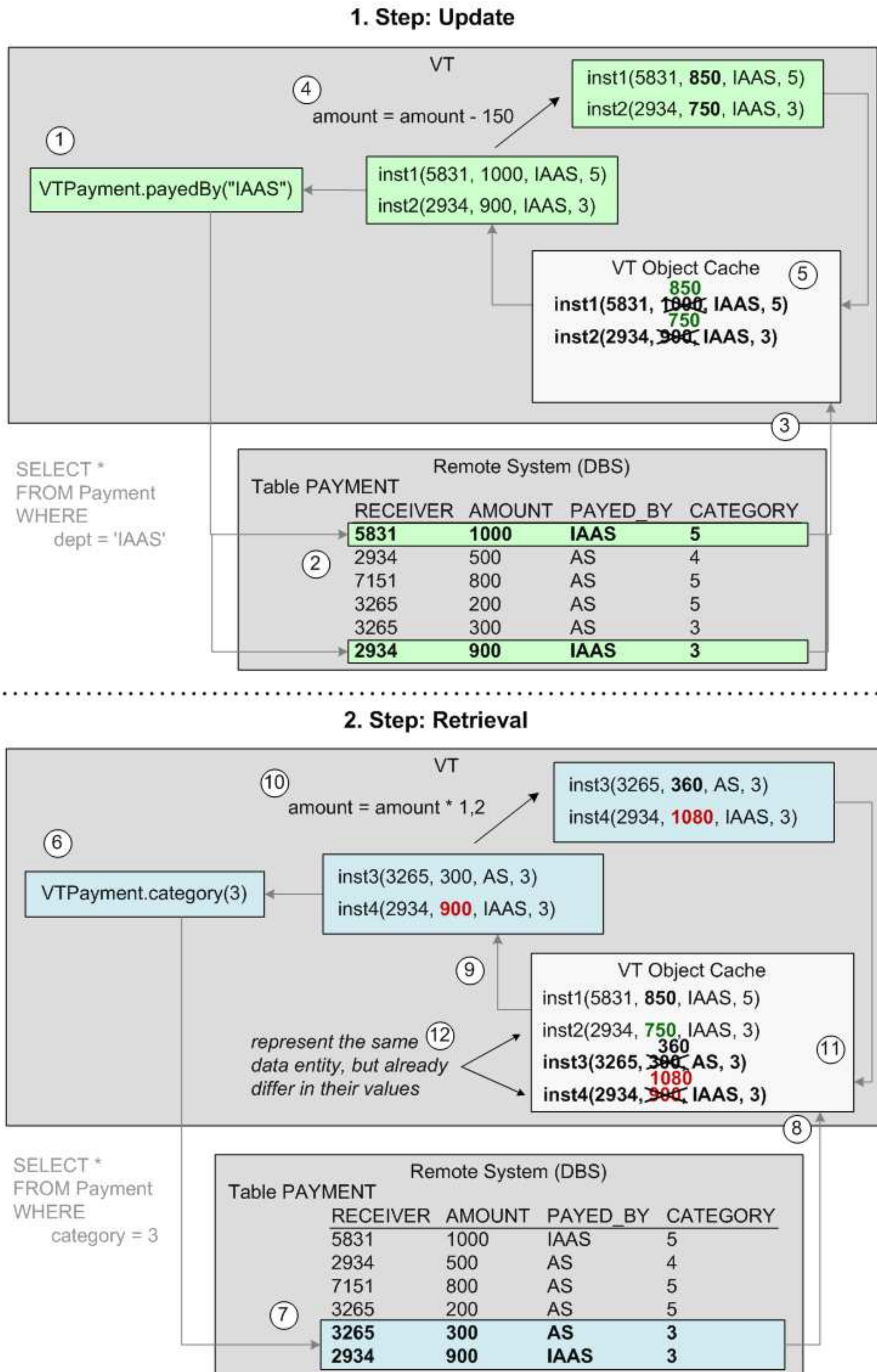


Figure 3.4: VT Object Mapping – Non-Unique (Mismatch).

values would fail since *VTPayment* instance *inst2* has already been modified and therefore now contains different values than *VTPayment* instance *inst4* does. If the VT transaction updates the two *VTPayment* instances returned by *VTPayment.category(3)*, i.e. *amount = amount * 1.2*, the outdated *amount* value of *inst4* is used to calculate it's new *amount* value. The correct solution would be to use the modified *amount* value of *inst2* to calculate the new *amount* value of *inst4* since *inst2* is the most recent representation of *Payment* tuple *(2934, 900, IAAS, 3)* at that point. However, the VT transaction commits *1080* as the new *amount* value of *Payment* tuple *(2934, 900, IAAS, 3)* instead of the correct result *900*. This example shows the necessity that a remote data entity is always represented by only one VT object instance at a time.

The reason of this incorrect behavior is that the VT is out of sync with the accessed DBS. There is no means of uniquely identifying a tuple in table *Payment* since there is no unique constraint and no primary key. The missing of a remote data identity can lead to decorrelation between remote data entities and VT object instances. Put in other words, the VT is not able to uniquely correlate a tuple of table *Payment* with its associated *VTPayment* instance all the time. Consequently, the VT needs to explicitly synchronize between the VT and the remote system if VT object instances have to be retrieved from the remote system and if VT object instances of the same VT object class have been modified before. This holds for VT object classes representing remote data entities that have no identity, e.g. Java classes without unique attributes or relational tables without unique constraints and primary keys. The synchronization can be realized in different ways. For example, changes can be explicitly synchronized with a remote system by writing each change to the remote system so that VT and remote system always stay synchronized. The usually more suitable solution is to keep the original values (see Figure 3.5, update step, 5) where each non-unique VT object instance retains its original version that has been mapped from the remote system. There is only one VT object, i.e. *inst2*, representing tuple *(2934, 900, IAAS, 3)* in table *Payment* (see Figure 3.5, retrieval step, 11) opposite to the example in Figure 3.4. In that way, VT objects in the VT object cache can be modified as desired and VT objects that are mapped from a remote system can be uniquely identified by means of the original versions of the VT objects in the VT object cache. The VT object cache replaces the original versions of the VT object instances by the current versions when the VT object instances are written to their correlated remote systems, e.g. when committing the VT transaction or for every write operation.

VT object classes representing remote data that owns an identity, e.g. a Java class with uniquely identifying attributes or a relational table with a primary key, are affected by the same behavior, if an attribute of a VT object instance that is part of the remote identity has been modified. In that case the correlation between the VT object instance and its remote data entity is lost either and must be explicitly preserved by correcting actions too. If attributes are modified that do not belong to the remote identity, operations are not critical.

One issue still remains. The missing of a remote identity (non-unique remote data

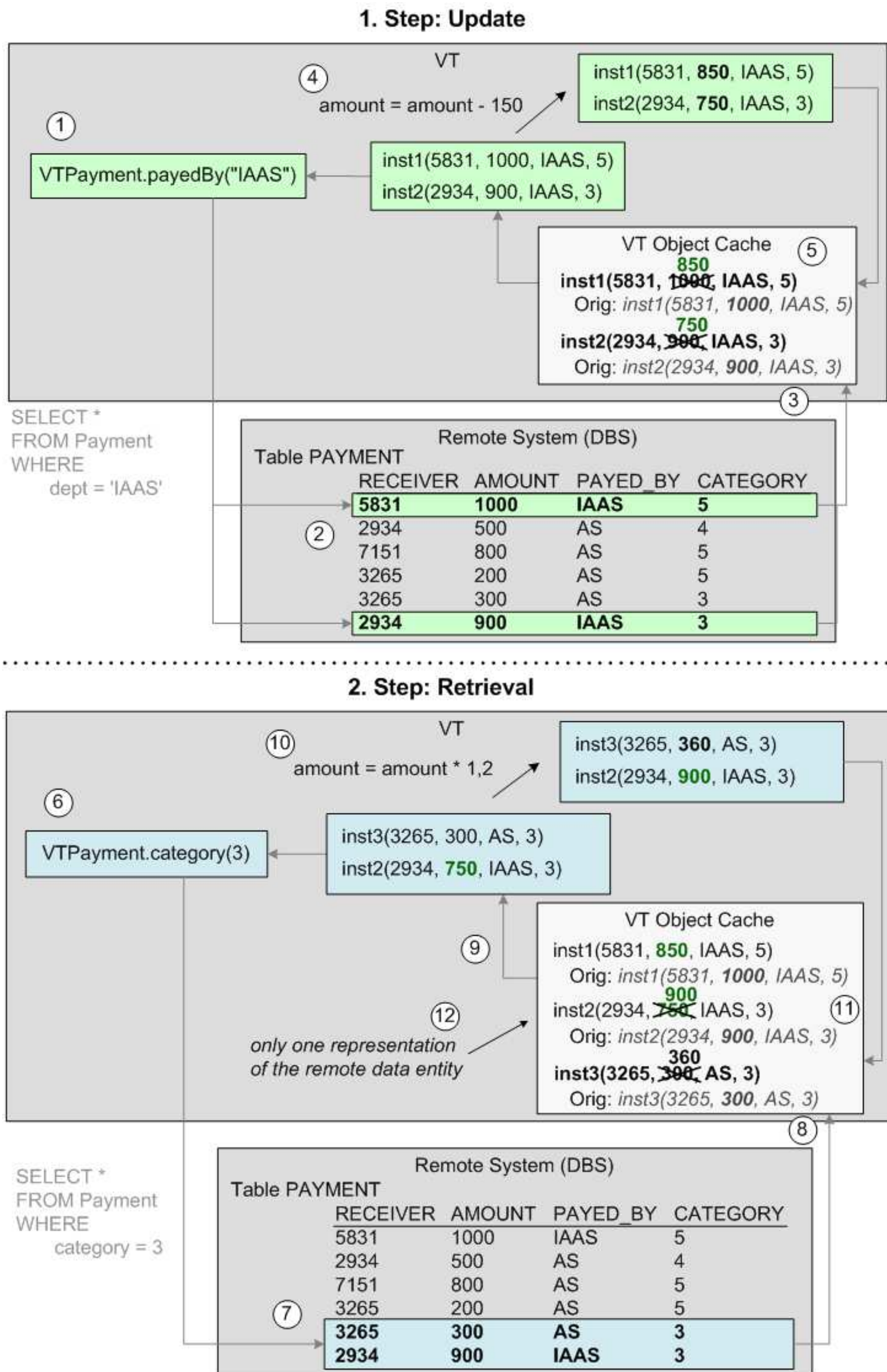


Figure 3.5: VT Object Mapping – Non-Unique (Correct).

entities, non-unique VT object instances) means that even all available properties of a remote data entity potentially are not sufficient for uniquely identifying it, i.e. there can be duplicates. A duplicate remote data entity of course cannot be unambiguously correlated with a VT object instance and vice versa. Access to a specific duplicate remote data entity is possible, if an order criterion is placed on the duplicates, e.g. retrieving duplicates into an array so that the array indexes make the duplicates distinguishable. The mapping from duplicate remote data entities to VT object instances and vice versa basically obeys the ambiguity of duplicates. The VT object cache checks for every non-unique remote data entity that a VT object operation retrieves whether a corresponding VT object instance has been instantiated earlier, i.e. whether the remote data entity has already been retrieved by another VT object operation earlier. If this is the case, the VT object cache uses the matching VT object instance as the representative of this remote data entity. Otherwise, it instantiates a new VT object instance based on the values of the remote data entity.

If a VT object operation returns a set of non-unique remote data entities that contains duplicate remote data entities, the VT object cache clearly has to provide a separate VT object instance for every duplicate remote data entity (see Figure 3.6) (3). If the VT object cache already contains VT object instances that represent the duplicate remote data entities, the VT object cache first maps the duplicate remote data entities to them, e.g. the first duplicate remote data entity to that duplicate VT object instance with the smallest *vtid*, the second to the duplicate VT object instance with the second smallest *vtid*, etc. and thereafter allocates new VT object instances if necessary, e.g. if there are only two duplicate VT object instances, the third duplicate remote data entity requires to allocate a new VT object instance. Updating or deleting one of the VT object instances results in updating or deleting one of the associated duplicate remote data entities (see Figure 3.6) (5 - 8). The actual correlation between a VT object instance and a specific duplicate remote data entity is not relevant (and also not possible, either) from the viewpoint of the VT since the missing of a remote identity, i.e. the non-uniqueness of a remote data entity, is intended by the remote system (or adapter) and thus it is the responsibility of the application, i.e. the VT client application in this case, to properly deal with non-unique remote data entities and non-unique VT object instances.

3.6 Object References

VT objects can reference other VT objects to build complex structures. A **VT object reference** is used as an attribute type in a VT object class to reference another VT object class. For example, VT object class *VTEmployee* contains an attribute called *dept* that references VT object class *VTDepartment*. If attribute *dept* is accessed in a *VTEmployee* instance it dereferences to the proper *VTDepartment* instance.

There are two different kinds of VT object references: explicitly resolvable VT

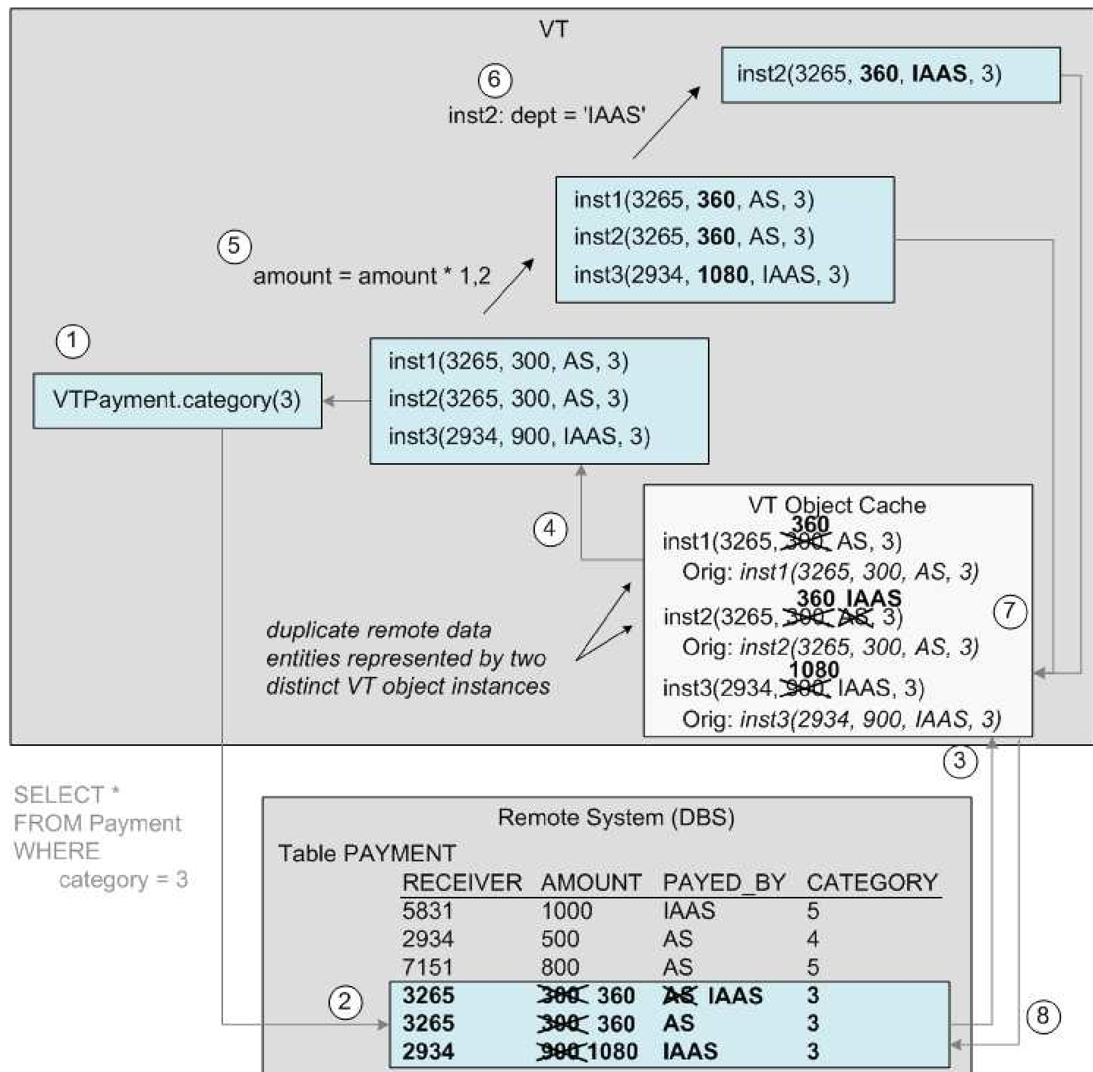


Figure 3.6: VT Object Mapping – Duplicates.

object references and non-explicitly resolvable VT object references. An **explicitly resolvable VT object reference** comprises a reference key and a reference operation. They are used to resolve the reference by means of a remote operation. The **reference key** consists of identifying attributes of the referenced VT object. The **reference operation** is a VT object read operation[‡] and uses the attribute values in the reference key as parameters to resolve the referenced VT object instance. Figure 3.7 shows an example. Attribute *dept* in *VTEmployee* instance *inst1* contains the name of the department, i.e. *"AS"*, as its reference key (1). Its reference operation is VT object read operation *getDept* of VT object *VTDepartment* (2). When a VT client application accesses attribute *dept* in *VTEmployee* instance *inst2*, the VT object manager uses the reference operation and the reference key value, i.e. *"AS"*,

[‡]Refer to Section 4.2.1 for details about VT object read operations.

to resolve the associated *VTDepartment* instance from the remote system via *VT-Department.getDept("AS")*. In this case, the DBS returns the requested department tuple (3) and the VT object manager creates a new *VTDepartment* instance based on these values (4). The *VTDepartment* reference of *VTEmployee* instance *inst2* (1) is now resolved to a VT-internal reference based on the VT identity of *VTDepartment* instance *inst3* (5) so that a VT client application now directly references *inst3* via *inst2* (6) in succeeding access operations on the department attribute of *inst2*.

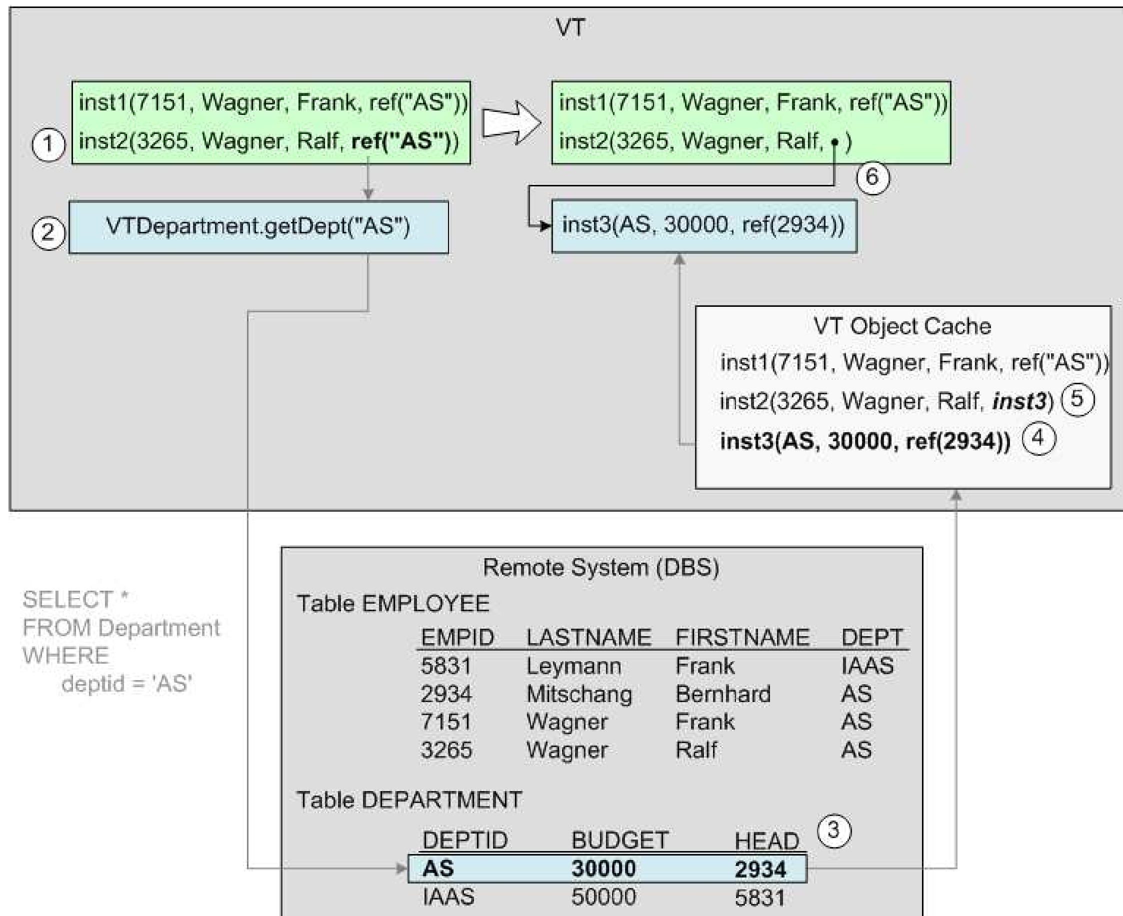


Figure 3.7: Dereferencing VT Objects.

The VT object manager uses the VT identity within the VT to uniquely identify VT object instances and VT object references. It uses the reference key and the reference operation of explicitly resolvable VT object references only in the mapping process when the reference is resolved from the remote system and when it is written back to the remote system. If the reference key consists of the remote identity of the referenced VT object, the VT object manager first looks for a VT object instance in the VT object cache with the same remote identity. If there is such an instance, the VT object manager uses this instance, otherwise the VT object manager resolves the reference from the remote system.

The reference operation of a VT object reference can be omitted. In that case, the VT object manager automatically uses the default unique VT object read operation of the referenced VT object. The prerequisite is that the reference key must correspond to the parameters of this VT object read operation. A unique VT object read operation of a VT object allows to uniquely identify and resolve a VT object instance of this VT object class, i.e. it returns only one VT object instance for a given reference key. If the reference operation is omitted and the default VT object read operation is used, the parameter of the operation usually is the remote identity of the VT object.

A **non-explicitly resolvable VT object reference** cannot be resolved by means of explicitly calling a remote operation, but is rather handled as an inherent part of a remote data entity. Hence, the reference must be resolved and transmitted together with the accessed VT object instance that contains the reference. This implicates that all VT object instances that are directly or indirectly referenced by a VT object instance have to be resolved if we only access this one VT object instance. For example, VT object class *VTNode* recursively references itself and thereby creates a tree structure (see example in Figure 3.8). If VT object class *VTNode* represents a Java class in the remote system and if the recursive references of *VTNode* represent Java references in the remote system, the Java references cannot be explicitly resolved because only the remote system implicitly knows the referenced Java objects and how they can be accessed, but the VT does not. This means that the directly and indirectly referenced *VTNode* instances and the corresponding Java objects have to be automatically resolved when the referencing, top-level Java object is resolved. For example, if we retrieve VT object instance *VTNode3*, the VT automatically retrieves the *VTNode* instances *3*, *5*, *6*, and *7* since the references in VT object class *VTNode* cannot be explicitly resolved.

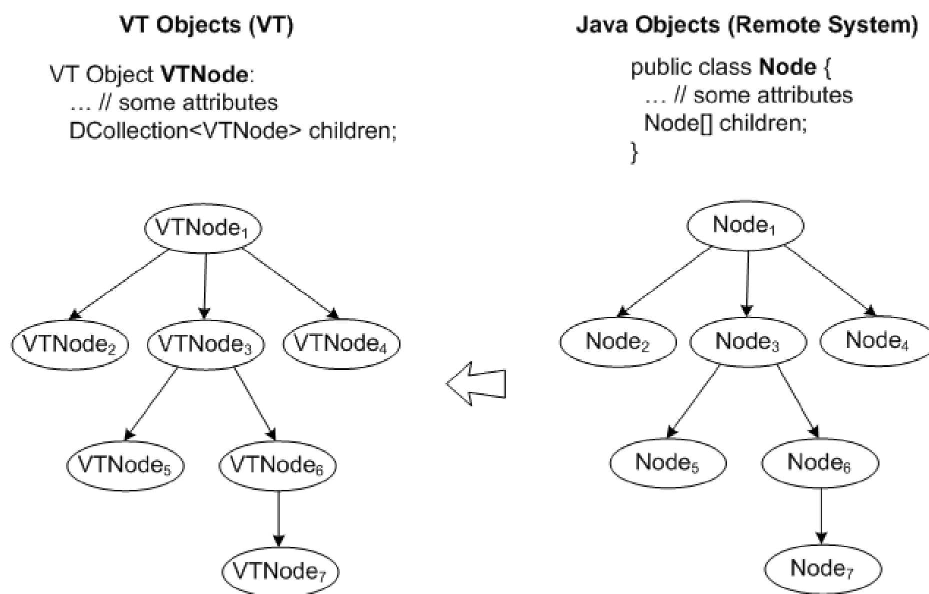


Figure 3.8: Non-Explicitly Resolvable VT Object References.

An explicitly resolvable VT object reference comprises a reference key and a reference operation or the *vtid*, i.e. the VT identity, if the reference has already been resolved. A non-explicitly resolvable VT object reference always contains the *vtid* of the referenced VT object instance since a non-explicitly resolvable VT object reference is automatically resolved when the referencing VT object instance is resolved. The concept of VT identity is also needed for transferring VT object instances to VT client applications without completely deserializing and serializing reference graphs of VT object instances.

3.7 Summary

The VT data model defines *VT objects* and *VT object operations* that represent data and operations in remote systems. VT objects can be compared either as VT objects or as representatives of remote data entities. The former case uses the *VT identity* for the comparison, the latter case uses the identity of the associated remote data entity, i.e. the *remote identity*. VT objects with a remote identity are *unique VT objects*, VT objects that do not have a remote identity are *non-unique VT objects*. The VT data model defines *VT object references* to build complex object structures. There are two different kinds of VT object references. An *explicitly resolvable VT object reference* comprises a *reference key* and a *reference operation* that are used to resolve the reference by means of a remote operation. A *non-explicitly resolvable VT object reference* cannot be resolved by means of explicitly calling a remote operation, but has to be resolved and transmitted together with the accessed VT object that contains the reference.

Chapter 4

Processing Model

There are two basic processing paradigms that we already identified in Section 2.2: *data-oriented processing* and *operation-oriented processing*. The VT represents remote data and remote operations as VT objects and allows to handle and process them in both ways. VT objects contain data as well as operations so that VT objects can be used in an operation-oriented manner by calling VT object operations and they can be used in a data-oriented way either by directly dealing with VT objects and their attributes or by means of VTQL, a set-oriented query language to retrieve and manipulate VT objects. In this chapter, we use the VT Java language binding to access VT objects via Java classes. VT client applications can use the VT Java language binding to programmatically access the VT.

4.1 Processing Transparency

The first example shows how **operation-oriented processing** works with the VT. The API of a CRM system offers a set of procedures written in C and corresponding C data structures, i.e. *struct Emp* and *struct Dept*:

```
struct Emp {                        struct Dept {
  int id;                             int id;
  string lastname;                    string name;
  string firstname;                   int budget;
  int salary;                       };
  struct Dept* dept;
};

struct Emp** get_emps();
struct Emp** get_emps(char* dept);
struct Emp* get_emp(int emp_id);
void add_emp(struct Emp* emp);
void add_emp(struct Emp* emp, char* msg);
void update_emp(struct Emp* emp);
void update_emp(struct Emp* emp, char* msg);
void replace_emp(int emp_id, struct Emp* emp);
void replace_emp(int emp_id, struct Emp* emp, char* msg);
```

```
void remove_emp(int emp_id, bool history);

struct Dept** get_depts();
struct Dept** get_depts(char* dept);
struct Dept* get_dept(int dept_id);
void add_dept(struct Dept* dept);
void add_dept(struct Dept* dept, char* msg);
void update_dept(struct Dept* dept);
void update_dept(struct Dept* dept, char* msg);
void replace_dept(int dept_id, struct Dept* dept);
void replace_dept(int dept_id, struct Dept* dept, char* msg);
void remove_dept(int dept_id, bool history);
```

We now integrate the CRM system into the VT by means of a J2EE connector and define suitable VT objects (via the VT Java language binding) to represent the given data and operations of the CRM system:

```
public class VTEmp {
  long id;
  String lastname;
  String firstname;
  long salary;
  VTDept dept;
}
```

```
public class VTDept {
  long id;
  String name;
  long budget;
}
```

```
public class VTCRMAccess {
  static DCollection<VTEmp> getEmps();
  static DCollection<VTEmp> getEmps(String dept);
  static VTEmp getEmp(long empId);
  static void addEmp(VTEmp emp);
  static void addEmp(VTEmp emp, String msg);
  static void updateEmp(VTEmp emp);
  static void replaceEmp(long empId, VTEmp emp, String msg);
  static void removeEmp(long empId, boolean history);
  ...
}
```

VT object class *VTCRMAccess* represents the remote operations, i.e. the C procedures, and *VTEmp* and *VTDept* represent the remote data, i.e. the C data structures *Emp* and *Dept*. Now, we can access the CRM system via the VT in an operation-oriented manner:

```
VTEmp emp = new VTEmp("Wagner", "Frank", 2000);
VTCRMAccess.addEmp(emp, "check was okay");
DCollection<VTEmp> emps = VTCRMAccess.getEmps();
...
emp.setSalary(emp.getSalary() * 1.2);
VTCRMAccess.replaceEmp(emp.getId(), emp, "good guy");
VTCRMAccess.removeEmp(emp.getId(), true);
```

The second example shows how **data-oriented processing** works with the VT. The remote system is a relational DBS with two tables, i.e. *Employee* and *Department*, which contain the same data as the CRM system of the operation-oriented example does:

```
CREATE TABLE EMPLOYEE (
  empid INT,
  lastname VARCHAR(30),
  firstname VARCHAR(30),
  salary INT,
  deptid INT
);
```

```
CREATE TABLE DEPARTMENT (
  deptid INT,
  name VARCHAR(20),
  budget INT
);
```

We integrate the DBS into the VT by means of an SQL wrapper and represent the two tables as suitable VT objects (via the VT Java language binding), i.e. *VTEmployee* and *VTDepartment*:

```
public class VTEmployee {
  long id;
  String lastname;
  String firstname;
  long salary;
  VTDepartment dept;
}
```

```
public class VTDepartment {
  long id;
  String name;
  long budget;
}
```

Now, we can access the DBS via the VT in a data-oriented manner by means of VTQL requests:

```
INSERT INTO VTEmployee
VALUES ('Wagner', 'Frank', 2000)
...
SELECT *
FROM VTEmployee
WHERE lastname = 'Wagner' AND firstname = 'Frank'
...
UPDATE VTEmployee
SET salary = salary * 1.2
WHERE lastname = 'Wagner' AND firstname = 'Frank'
...
DELETE FROM VTEmployee
WHERE lastname = 'Wagner' AND firstname = 'Frank'
```

The VT does not only offer operation-oriented processing and data-oriented processing, but additionally allows to handle and process VT objects independent of the processing paradigm of an adapter technology or remote system (see Figure 4.1). A client system, e.g. a middleware system, can choose the access style (*specific access style*), i.e. whether it wants to access the VT by means of VTQL requests or by means of VT object operations. The VT transparently processes the desired VTQL requests or VT object operations underneath (*transparent access style*) independent of the access styles of the underlying adapters and remote systems (*specific access*

styles). For example, the SQL-FDBS in Figure 4.1 chooses to transform SQL requests to VTQL requests and the Java EE application server chooses to transform Java method calls to VT object operation calls (*specific access styles*). The VT has to cope with both access styles (*transparent access style*) and in turn suitably accesses the required remote systems either in an operation-oriented manner or in a data-oriented manner (*specific access styles*). For example, if the VTQL request of the SQL-FDBS accesses a remote system via a J2EE connector, the J2EE connector still requires Java method calls, i.e. operation-oriented requests, but not a data-oriented request as it is given by the VTQL request. Therefore, the VT has to provide a means that can overcome the paradigm change from data-oriented processing to operation-oriented processing. Analogous considerations hold for the Java EE application server. For example, if the VT object operation calls of the Java EE application server access a remote system via an SQL wrapper, the SQL wrapper still requires SQL requests, i.e. data-oriented requests, but not operation-oriented requests as given by the VT object operation calls. Here, the VT provides *paradigm transparency* and internally performs a paradigm change from operation-oriented processing to data-oriented processing or vice versa if necessary.

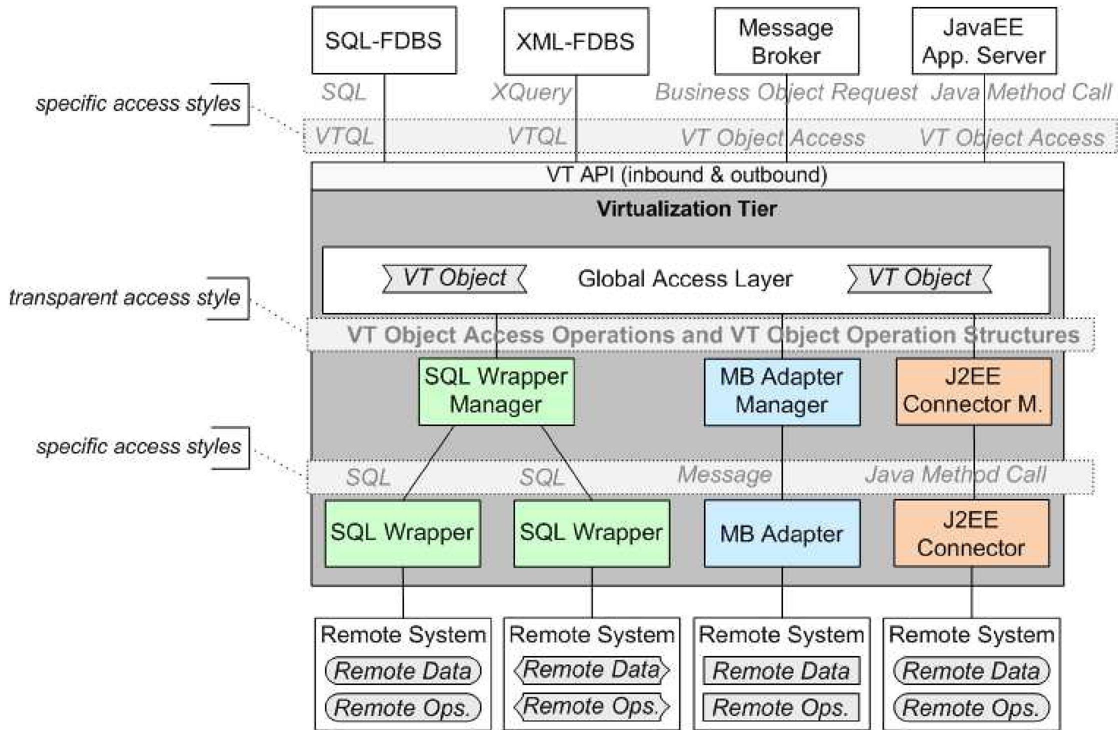


Figure 4.1: VT Objects and VT Access.

If the VT was not able to perform paradigm changes between operation-oriented processing and data-oriented processing, each client system would have to explicitly regard the processing paradigm of the currently accessed adapter. For example, each client system would have to issue VT object operation calls, if it wants to access a J2EE connector and each client system would have to issue VTQL requests, if it

wants to access an SQL wrapper. Therefore, the VT supports *transparent processing*, i.e. paradigm transparency, which means that the VT provides an operation-oriented view to data-oriented processing and a data-oriented view to operation-oriented processing. The VT realizes transparent processing by transforming VTQL requests and VT object operations into equivalent CRUD representations.

A **CRUD representation** is the bridge between VTQL requests and VT object operations and is realized by VT object access operations and VT object operation structures. Data-oriented processing relies on the CRUD principle, i.e. creating, reading, updating and deleting data. We represent the CRUD principle by means of VT object access operations. A **VT object access operation** (short: **VTO access operation**) is a standardized VT object operation for handling VT objects and their attributes in an operation-oriented manner. There are four types of VTO access operations according to the CRUD principle: **create**, **read**, **update**, **delete**.

Operation-oriented processing relies on operations and operation signatures. An operation signature consists of an operation name and input and output parameters. We represent remote operations by means of VT object operation structures. A **VT object operation structure** (short: **VTO operation structure**) is a data structure that consists of attributes analogous to data structures and attributes of VT objects. VTO operation structure attributes represent VTO access operation parameters that are not an inherent part of a VTO access operation type, i.e. they are not necessarily required for a VTO access operation of that type, but they belong to a specific VTO access operation only (more on VTO access operations and VTO operation structures in the next section).

We evolve the operation-oriented example, i.e. *VTEmp*, *VTDept* and *VTCRMAccess* and map the C procedures and the C data structures *Emp* and *Dept* to a CRUD representation in the VT using VTO access operations and VTO operation structures:

```
public class VTEmp {
  long id;
  String lastname;
  String firstname;
  long salary;
  VTDept dept;

  // VTO access operations
  @AccessOperation static DCollection<VTEmp> read();
  @AccessOperation static VTEmp create(DMap<String, Object> empValues);
  @AccessOperation void update();
  @AccessOperation void delete();

  // VTO operation structures
  @OperationStructure(type="read", card="single")
    static void getEmp(long empId);
  @OperationStructure(type="read", card="multiple")
    static void getEmpsOfDept(String dept);
  @OperationStructure(type="read", card="multiple")
    static void getAllEmps();
```

```
  @OperationStructure(type="create", card="single")
    static void addEmp();
  @OperationStructure(type="create", card="single")
    static void addEmp(String msg);
  @OperationStructure(type="update", card="single")
    static void updateEmp();
  @OperationStructure(type="update", card="single")
    static void replaceEmp(String msg);
  @OperationStructure(type="delete", card="single")
    static void removeEmp(boolean history);
  ...
}

public class VTDept {
  long id;
  String name;
  int budget;
}
```

VT object *VTCRMAccess* is no longer necessary since VT objects *VTEmp* and *VTDept* now contain standardized VTO access operations and VTO operation structures instead of the custom VT object operations in *VTCRMAccess*.

Next, we evolve the data-oriented example, i.e. *VTEmployee* and *VTDepartment*, and use VTO access operations and VTO operation structures so that we achieve a CRUD representation in the VT:

```
public class VTEmployee {
  long id;
  String lastname;
  String firstname;
  long salary;
  VTDepartment dept;

  // VTO access operations
  @AccessOperation static DCollection<VTEmployee> read();
  @AccessOperation
    static DCollection<VTEmployee>
    create(DCollection<DMap<String, Object>> empValues);
  @AccessOperation void update();
  @AccessOperation void delete();

  // VTO operation structures
  @OperationStructure(type="read", card="multiple")
    static void selectEmps();
  @OperationStructure(type="create", card="multiple")
    static void insertEmps();
  @OperationStructure(type="update", card="multiple")
    static void updateEmps();
  @OperationStructure(type="delete", card="multiple")
    static void deleteEmps();
  ...
}
```

```
public class VTDepartment {
  long id;
  String name;
  int budget;
}
```

VT objects *VTEmployee* and *VTDepartment* now additionally contain standardized VTO access operations and VTO operation structures. All VTO operation structures are empty since we do not need additional VTO access operation parameters.

Now, we have transformed an operation-oriented representation of VT objects as well as a data-oriented representation of VT objects into VT objects in CRUD representation so that we can transparently process them without considering their original representation in the remote systems. In this way, the VT can transparently handle different access styles from client systems, i.e. data-oriented requests as well as operation-oriented requests, and the next sections shows how this actually is done. In the rest of this chapter we refer to the code fragments in this section, i.e. VT objects *VTEmp*, *VTDept*, *VTEmployee*, *VTDepartment* and the associated CRM System API and the SQL tables.

4.2 VTO Access Operations

Each type of VTO access operation is mapped to a remote operation by means of a mapping pattern that is specific to the respective VTO access operation type, which is described in this section.

4.2.1 Read Operation

A read operation is always called via a *read* VTO access operation and a suitable VTO operation structure of type “read”, i.e. `type="read"`. For example, we first initialize VTO operation structure *VTEmp.getEmpsOfDept(”AS”)* and then we call VTO access operation *VTEmp.read()* to retrieve one or more *VTEmp* instances from the CRM system:

```
// initialize VTO operation structure of type "read"
VTEmp.getEmpsOfDept("AS");
// call read VTO access operation
DCollection<VTEmp> vtemps = VTEmp.read();
// process returned VTEmp instances
Iterator<VTEmp> iter = vtemps.iterator();
while  (iter.hasNext()) {
  VTEmp emp = iter.next();
  if  ((vtemp.getLastname().equals("Wagner"))
       && (vtemp.getFirstname().equals("Frank")))  {
     ...
  }
}
```

VTEmp defines three *read* VTO operation structures and thereby three variants of the *read* VTO access operation. The first *read* VTO operation structure is named *getEmp* and returns one *VTEmp* instance (`card="single"`), the second and the third one return a collection of *VTEmp* instances (`card="multiple"`). The three VTO operation structures are mapped from the three C procedures *get...* of the CRM system.*

The only inherent parameter of a *read* VTO access operation is the result parameter, e.g. *struct Emp** or *struct Emp*** in the *get...* procedures of the CRM system, which are mapped to one or more *VTEmp* instances in the VT. An **inherent VTO access operation parameter** is a parameter that is always required for the VTO access operation. For example, every *read* VTO access operation always requires a result parameter that returns either a collection of VT object instances of the calling VT object class or a single VT object instance. A VTO access operation can also have additional parameters. An **additional VTO access operation parameter** is a parameter that is correlated with a specific remote operation and therefore is defined as an attribute of the VTO operation structure that represents this remote operation. For example, the first *get_emps(...)* remote operation of the CRM system does not have a parameter (except the result parameter, which is inherent), but the second one does, i.e. *char* dept*. Therefore, we map the *dept* parameter to a corresponding VTO operation structure attribute in the VTO operation structure named *getEmpsOfDept*, i.e. *String dept*. Analogously, we define a VTO operation structure attribute *int empId* in the *read* VTO operation structure *getEmp(...)*, which is mapped from the parameter *int emp_id* of remote operation *get_emp*.

If we now want to retrieve all employees of department *AS* (see code fragment above), we first have to properly initialize VTO operation structure *getEmpsOfDept(...)*, i.e. by calling *VTEmp.getEmpsOfDept("AS")*. Then we can call the *read* VTO access operation, i.e. *VTEmp.read()*, which maps the previously initialized VTO operation structure, i.e. with value *"AS"*, to the corresponding remote operation call, i.e. *get_emps("AS")*. Finally, the VT maps the returned *Emp* array as the only inherent parameter of the *read* VTO access operation to a collection of *VTEmp* instances as the result of the *VTEmp.read()* call. Next, we can iterate over the returned collection of *VTEmp* instances and access a specific one, e.g. the one with *lastname = "Wagner"* and with *firstname = "Frank"*.

4.2.2 Create Operation

A create operation is always called via a *create* VTO access operation and a suitable VTO operation structure of type “create”, i.e. `type="create"`. For example, we first initialize VTO operation structure *VTEmp.addEmp("check was okay")*. Then we call *VTEmp.create(...)* with attribute values for the new *VTEmp* instance to be created and thereby also create a corresponding *Emp* data entity in the CRM system:

*The mapping and other details of VT object configurations are extensively discussed in Chapter 5.

```
DMap<String, Object> empVals = new DMap<String, Object>();
empVals.put("lastname", "Wagner");
empVals.put("firstname", "Frank");
empVals.put("salary", 2000);
...
VTEmp.addEmp("check was okay");
VTEmp vtemp = VTEmp.create(empVals);
```

VTEmp defines two *create* VTO operation structures and thereby two variants of the *create* VTO access operation. The *create* VTO operation structures are named *addEmp* and create one *VTEmp* instance (`card="single"`), respectively. The first VTO operation structure is empty, i.e. it contains no additional parameters, whereas the second VTO operation structure defines an additional parameter, i.e. VTO operation structure attribute *String msg*. Moreover, a *create* VTO access operation has two inherent parameters. A result parameter that returns one VT object instance or a collection of VT object instances. The other inherent VTO access operation parameter are the data values of the VT object instance or instances to be created. There are three alternatives. The first alternative defines one parameter for each VT object attribute. The second uses a *DMap* parameter to specify the VT object attribute values in an associative array where the index names conform to the attribute names. Both alternatives create one VT object instance. The third alternative takes a parameter that holds a collection of *DMap* objects so that one or more VT object instances can be created in one step. There are other alternatives, e.g. a parameter that contains another VT object instance of the same VT object class or a parameter that contains a collection of VT object instances of the same VT object class. The new VT object instances are then created as a copy of these VT object instances.

The two *create* VTO operation structures, i.e. *addEmp(...)*, are mapped to the two C procedures *add_emp(...)* of the CRM system. The empty VTO operation structure is mapped to remote operation *add_emp(struct Emp* emp)*. The second VTO operation structure with attribute *String msg* is mapped to remote operation *add_emp(struct Emp* emp, char* msg)* where *char* msg* is mapped from VTO operation structure attribute *String msg*. Parameter *struct Emp* emp* of both remote operations is mapped from the inherent VTO access operation parameter *DMap<String, Object> empValues* and the inherent VTO access operation result parameter maps the returned *VTEmp* instance from the *struct Emp* emp* parameter, too.

If we want to create a new employee data entity in the CRM system, we first create the necessary *DMap* input parameter (see code fragment above), then we initialize a *create* VTO operation structure, e.g. *VTEmp.addEmp("check was okay")* and finally we can perform the *create* VTO access operation: *VTEmp.create(empVals)*. The VT then performs an adapter interaction that maps both operation calls to a corresponding CRM system call, i.e. *add_emp(emp, "check was okay")* where *emp* is of type *struct Emp* and contains the values given by *empVals* in the VT. The VT also internally creates a *VTEmp* instance from the same values and returns it as the result of the *create* VTO access operation.

4.2.3 Update Operation

An update operation is always called via an *update* VTO access operation and a suitable VTO operation structure of type "update", i.e. `type="update"`. For example, we first initialize VTO operation structure *VTEmp.replaceEmp("good guy")*. Then we call *vtemp.update()* where *vtemp* is a *VTEmp* instance that has been previously changed in the transaction by means of a VT object operation, e.g. *vtemp.setSalary(vtemp.getSalary() * 1.2)*. Finally, the VT changes the associated *Emp* data entity in the CRM system too:

```
VTEmp.getEmpsOfDept("AS");
DCollection<VTEmp> vtemps = VTEmp.read();
Iterator<VTEmp> iter = vtemps.iterator();
while  (iter.hasNext()) {
  VTEmp vtemp = iter.next();
  if  ((vtemp.getLastname().equals("Wagner"))
       && (vtemp.getFirstname().equals("Frank")))  {
    vtemp.setSalary(vtemp.getSalary() * 1.2);
    VTEmp.replaceEmp("good guy");
    vtemp.update();
    break;
  }
}
```

VTEmp defines two *update* VTO operation structures and thereby two variants of *update* VTO access operations. The first *update* VTO operation structure is named *updateEmp* and does not contain attributes, i.e. there are no additional VTO access operation parameters, the second *update* VTO operation structure is named *replaceEmp* and has one attribute, i.e. *String msg*. An *update* VTO access operation has one inherent parameter to identify the remote data entity to be updated and to deliver the modified values of the VT object to the remote data entity. The whole information is contained in the VT object instance itself. For example, VTO operation structure *updateEmp* is mapped to remote operation *update_emp(struct Emp* emp)* and the inherent parameter *vtemp* is mapped to remote operation parameter *struct Emp* emp*. VTO operation structure *replaceEmp* is mapped to remote operation *replace_emp(int emp_id, struct Emp* emp, char* msg)* and the inherent parameter *vtemp* is mapped to the remote operation parameters *int emp_id* and *struct Emp* emp*. The first remote operation parameter identifies the remote data entity to be modified and the second remote operation parameter determines the modified values. The third remote operation parameter, i.e. *char* msg*, is mapped from VTO operation structure attribute *String msg*.†

If we want to modify employee data in the CRM system, we first of all have to retrieve the remote data that we want to modify as VT object instances (see code fragment above). Therefore, we perform a read request as described above, i.e. *VTEmp.getEmpsOfDept("AS")* and *VTEmp.read()*. Then we look up the *VTEmp* instance identified by *lastname = "Wagner"* and *firstname = "Frank"* and perform

†For details about mappings refer to Chapter 5.

the desired changes on the selected *VTEmp* instance, e.g. *vtemp.setSalary(vtemp.getSalary() * 1.2)*. Thereafter, we initialize an *update* VTO operation structure, e.g. *VTEmp.replaceEmp("good guy")*, and finally we perform the *update* VTO access operation, i.e. *vtemp.update()*. The VT uses the associated adapter and calls the corresponding CRM system operation, i.e. *replace_emp(37, emp, "good guy")*, where *37* is the *id* value extracted from the unaltered *vtemp* version and where *emp* is the mapping from the modified *vtemp* version.

4.2.4 Delete Operation

A delete operation is always called via a *delete* VTO access operation and a suitable VTO operation structure of type "delete", i.e. `type="delete"`. For example, we first initialize VTO operation structure *removeEmp*, i.e. *VTEmp.removeEmp(true)*. Then we call *vtemp.delete()* where *vtemp* is a *VTEmp* instance and the VT deletes the *VTEmp* instance from the VT object cache and the associated remote data entity from the CRM system:

```
VTEmp.getEmpsOfDept("AS");
DCollection<VTEmp> vtemps = VTEmp.read();
Iterator<VTEmp> iter = vtemps.iterator();
while  (iter.hasNext()) {
  VTEmp vtemp = iter.next();
  if  ((vtemp.getLastname().equals("Wagner"))
       && (vtemp.getFirstname().equals("Frank")))  {
    VTEmp.removeEmp(true);
    vtemp.delete();
    break;
  }
}
```

VTEmp defines one *delete* VTO operation structure, which is named *removeEmp* and which contains one attribute, i.e. *boolean history*, as an additional parameter for the *delete* VTO access operation. The only inherent VTO access operation parameter of a *delete* VTO access operation is the VT object instance to be deleted, i.e. a *VTEmp* instance in this case. The VTO operation structure is mapped to the *remove_emp* C procedure of the CRM system. The procedure has two parameters, *int emp_id* and *bool history*. The latter parameter is mapped from VTO operation structure attribute *boolean history*, the former parameter is mapped from the inherent VTO access operation parameter, i.e. *vtemp*. The inherent parameter is necessary to identify the remote data entity to be deleted. In this case, we only need the identifier, which the VT maps from *vtemp.id* to *emp_id*.

If we want to delete employee data from the CRM system, we first have to retrieve the VT object instances to be deleted via *VTEmp.getEmpsOfDept("AS")* and *VTEmp.read()* (see code fragment above). Then we select the *VTEmp* instance identified by *lastname = "Wagner"* and *firstname = "Frank"*, initialize the *delete* VTO operation structure, i.e. *VTEmp.removeEmp(true)*, and perform the delete operation, i.e. *vtemp.delete()*. The VT correspondingly calls the remote operation,

i.e. *remove_emp(37, true)*, where *37* is the *id* value extracted from *vtemp*. Finally, the VT object manager deletes *vtemp* from the VT object cache as well.

4.2.5 Operation-Oriented Data Usage

The CRUD representation of the DBS tables, i.e. *VTEmployee* and *VTDepartment*, integrated by the DBS SQL wrapper as shown in Section 4.1 allows us to access them in the same way, i.e. operation-oriented, despite the fact that the SQL wrapper only provides data-oriented access:

```
DMap<String, Object> empVals = new DMap<String, Object>();
empVals.put("lastname", "Wagner");
empVals.put("firstname", "Frank");
empVals.put("salary", 2000);
...
VTEmployee.insertEmps();
VTEmployee vtemp = VTEmployee.create(empVals);
...
VTEmployee.selectEmps();
DCollection<VTEmployee> vtemps = VTEmployee.read();
Iterator<VTEmployee> iter = vtemps.iterator();
while  (iter.hasNext()) {
  vtemp = iter.next();
  if  ((vtemp.getLastname().equals("Wagner"))
       && (vtemp.getFirstname().equals("Frank")))  {
    vtemp.setSalary(vtemp.getSalary() * 1.2);
    VTEmployee.updateEmps();
    vtemp.update();
    ...
    VTEmployee.deleteEmps();
    vtemp.delete();
    break;
  }
}
```

A more convenient way of accessing VT objects can be used if a VTO operation structure does not contain additional parameters, i.e. only inherent ones, and if its name is unique among the VTO operation structures of the VT object. Then the VTO operation structure initialization can be completely omitted when accessing VT objects:

```
DMap<String, Object> empVals = new DMap<String, Object>();
empVals.put("lastname", "Wagner");
empVals.put("firstname", "Frank");
empVals.put("salary", 2000);
...
VTEmployee vtemp = VTEmployee.create(empVals);
...
DCollection<VTEmployee> vtemps = VTEmployee.read();
Iterator<VTEmployee> iter = vtemps.iterator();
```

```
while  (iter.hasNext()) {
  vtemp = iter.next();
  if  ((vtemp.getLastname().equals("Wagner"))
       && (vtemp.getFirstname().equals("Frank")))  {
    vtemp.setSalary(vtemp.getSalary() * 1.2);
    vtemp.update();
    ...
    vtemp.delete();
    break;
  }
}
```

We can now achieve transparent processing by employing CRUD representations of remote data and remote operations and thereby enable operation-oriented processing as well as data-oriented processing. Right now, we have seen operation-oriented processing on VT objects in CRUD representation independent of the employed adapter technology and remote system. In the next section we show how data-oriented processing is achieved by means of VTQL requests on VT objects in CRUD representation.

4.3 VTQL Requests

VTQL is based on OQL, which in turn is based on SQL. OQL only provides a retrieval part, i.e. SELECT queries (also see [CBB+00]). We additionally include the three other CRUD principles in VTQL, i.e. create, update and delete requests, analogously to SQL. Thereby, VTQL requests provide a data-oriented way of accessing the VT.

4.3.1 Read Requests

A VTQL read request retrieves remote data entities using a VTO operation structure if necessary analogous to the operation-oriented way shown in Section 4.2.1. For example, data-oriented access to *VTEmp* (see definition on page 73) uses VTO operation structure *getEmpsOfDept(...)* to properly access the CRM system in a data-oriented way:

```
SELECT *
FROM   VTEmp.getEmpsOfDept("AS")
WHERE  lastname = 'Wagner' AND firstname = 'Frank'
```

Note that this is possible although the CRM J2EE connector only provides operation-oriented access! The FROM clause now contains the suitable VTO operation structure initialization, i.e. *VTEmp.getEmpsOfDept("AS")*, that is the basis for the *read* VTO access operation. The *read* VTO access operation is implicitly executed by the FROM clause so that one or more *VTEmp* instances are retrieved from the CRM system as output for the FROM clause. The selection and projection in the WHERE and SELECT clauses then perform as usual.

4.3.2 Create Requests

A VTQL create request creates one or more new remote data entities and corresponding VT object instances using a VTO operation structure if necessary. For example, data-oriented instantiation of *VTEmp* can use VTO operation structure *addEmp(...)* to properly access the CRM system in a data-oriented way:

```
INSERT INTO VTEmp.addEmp('check was okay')
VALUES ('Wagner', 'Frank', 2000)
```

The INSERT INTO clause contains the proper VTO operation structure initialization, i.e. *VTEmp.addEmp("check was okay")*, which is the basis for the *create* VTO access operation. The VALUES clause specifies the attribute values of the new *VTEmp* instance to be created so that the overall INSERT statement can be executed as a *create* VTO access operation.[‡]

4.3.3 Update Requests

A VTQL update request writes one or more modified VT object instances to the associated remote data entities. For example, data-oriented modification of *VTEmp* can use VTO operation structures *getEmpsOfDept(...)* and *replaceEmp(...)* to properly access the CRM system in a data-oriented way:

```
UPDATE VTEmp.getEmpsOfDept("AS")
SET    salary = salary * 1.2
WHERE  replaceEmp("good guy")
       AND lastname = 'Wagner'
       AND firstname = 'Frank'
```

The UPDATE clause does not contain the initialization of an *update* VTO operation structure, but requires a *read* VTO operation structure, which is responsible for retrieving the *VTEmp* instances to be changed in the CRM system, e.g. *VTEmp.getEmpsOfDept("AS")*. VTO operation structure *getEmpsOfDept(...)* allows the UPDATE clause to perform a corresponding *read* VTO access operation and to retrieve a set of *VTEmp* instances, which are further selected by the WHERE clause, i.e. *lastname = 'Wagner' AND firstname = 'Frank'*. The selected *VTEmp* instances are then changed corresponding to the SET clause, i.e. *salary = salary * 1.2*. The last step is the use of an *update* VTO operation structure in the WHERE clause, e.g. *replaceEmp("good guy")*, which provides the necessary information for the *update* VTO access operation that is finally performed by the overall UPDATE statement.

[‡]The values in the VALUES clause are mapped by the VT object manager to a corresponding *DMap* input parameter that is required for the *VTEmp.create(...)* operation.

4.3.4 Delete Requests

A VTQL delete request deletes one or more VT object instances and the associated remote data entities. For example, data-oriented deleting of *VTEmp* can use VTO operation structures *getEmpsOfDept(...)* and *removeEmp(...)* to properly access the CRM system in a data-oriented way:

```
DELETE FROM VTEmp.getEmpsOfDept("AS")
WHERE removeEmp(true)
      AND lastname = ’Wagner’
      AND firstname = ’Frank’
```

The DELETE FROM clause does not contain the initialization of a *delete* VTO operation structure, but refers to a *read* VTO operation structure, which is responsible for retrieving the *VTEmp* instances to be deleted from the CRM system, e.g. *VTEmp.getEmpsOfDept(”AS”)*. The VTO operation structure allows the DELETE FROM clause to perform a corresponding *read* VTO access operation and to retrieve a set of *VTEmp* instances, which are further selected by the WHERE clause, i.e. *lastname = ’Wagner’ AND firstname = ’Frank’*. Finally, a *delete* VTO operation structure is suitably initialized in the WHERE clause, i.e. *removeEmp(true)*, and the *delete* VTO access operation is performed by the overall DELETE statement.

4.3.5 Data-Oriented Data Usage

The CRUD representation of the DBS tables, i.e. *VTEmployee* and *VTDepartment*, integrated by the DBS SQL wrapper as shown in Section 4.1 of course allows us to access them in a data-oriented way, too:

```
INSERT INTO VTEmployee.insertEmps()
VALUES (’Wagner’, ’Frank’, 2000)

SELECT *
FROM VTEmployee.selectEmps()
WHERE lastname = ’Wagner’ AND firstname = ’Frank’

UPDATE VTEmployee.selectEmps()
SET salary = salary * 1.2
WHERE updateEmps()
      AND lastname = ’Wagner’
      AND firstname = ’Frank’

DELETE FROM VTEmployee.selectEmps()
WHERE deleteEmps()
      AND lastname = ’Wagner’
      AND firstname = ’Frank’
```

First, we create a new *VTEmployee* instance, then we retrieve all *VTEmployee* instances, next we perform an update and finally we delete one or more *VTEmployee* instances.

We can syntactically simplify the VTQL requests since *VTEmployee* has only one VTO operation structure for each VTO access operation type and since all VTO operation structures are empty. Therefore, we can omit the VTO operation structure names as well as the parameters so that we come up with a semantically equivalent, but syntactically simplified form of VTQL requests:

```
INSERT INTO VTEmployee
VALUES ('Wagner', 'Frank', 2000)

SELECT *
FROM VTEmployee
WHERE lastname = 'Wagner' AND firstname = 'Frank'

UPDATE VTEmployee
SET salary = salary * 1.2
WHERE lastname = 'Wagner' AND firstname = 'Frank'

DELETE FROM VTEmployee
WHERE lastname = 'Wagner' AND firstname = 'Frank'
```

Actually, the syntactically simplified form is identical to the original syntax shown in Section 4.1.

The result of the last two sections is that *VTEmployee* and *VTDepartment* as well as *VTemp* and *VTDept* can be accessed in an operation-oriented way as well as in a data-oriented way since they conform to the CRUD representation independent of the employed adapter technology or the underlying remote system. Now, we have shown how operation-oriented remote systems and adapter technologies as well as data-oriented remote systems and adapter technologies can be mapped to VT objects in CRUD representation. The CRUD representation of a VT object, i.e. its VTO access operations and VTO operation structures, enables transparent processing, i.e. operation-oriented processing as well as data-oriented processing on the same VT object, so that the access paradigm of a client system becomes independent from the access paradigm of an adapter or remote system.

4.4 Query Execution

VTQL requests can not always be directly mapped to corresponding requests on remote systems. This is the case if an adapter technology or a remote system does not support data-oriented processing. For example, the CRUD representation of our operation-oriented example on page 73, i.e. *VTEmp* and *VTDept*, does not allow to directly map a VTQL read request or a VTQL update request to the CRM system since we have C procedures that only perform single operation steps. Therefore, the read request on page 81 is executed in two steps. The VT first executes the FROM clause, i.e. *VTEmp.getEmpsOfDept("AS")* and the implicit *read* VTO access operation, which is mapped to C procedure call *get_emps("AS")*. The VT then compensates for the rest of the read request, i.e. the SELECT clause and the WHERE clause.

Similar considerations hold for the update request on page 82. It is executed in three steps. The VT first executes the UPDATE clause, i.e. *VTEmp.getEmpsOfDept("AS")* and the implicit *read* VTO access operation, which is mapped to C procedure call *get_emps("AS")*. The VT then compensates for the selection criteria in the WHERE clause, i.e. *lastname = 'Wagner' AND firstname = 'Frank'*, and performs the changes determined by the SET clause on the selected VT object instances, i.e. *salary = salary * 1.2*. Finally, the VT writes the modified data to the CRM system using the given *update* VTO operation structure, i.e. *replaceEmp("good guy")*, which is mapped to one or more *replace_emp(...)* procedure calls in the CRM system, e.g. *replace_emp(37, emp, "good guy")*.

On the other hand, the CRUD representation of our data-oriented example on page 74, i.e. *VTEmployee* and *VTDepartment*, allows to directly map the read request or the update request to the remote system since it is a DBS. The VT translates the read request and the update request on page 83 into corresponding SQL requests and can therefore directly submit them to the DBS so that the DBS internally executes them without any further execution or compensation in the VT.

Thus, we have to consider two important execution aspects. The first one is that the VT does not only need the capability to transform complex VTQL requests into complex data-oriented requests on remote systems, e.g. SQL requests or XQuery requests, but it also requires the capability to compensate for missing request execution capabilities in remote systems or adapter technologies. This means that the VT has to be able to determine which adapter and which remote system is able to execute which part of a VTQL request. This can be done either by defining the query capabilities in VT object configurations or by dynamically negotiating about VTQL queries and query fragments that an adapter and a remote system are willing to execute. The second execution aspect is that the VT has to be able to optimize VTQL requests that span more than one remote system. This means that the VT does not only need to decide on VTQL queries and query fragments for one remote system, but for two or more. The query evaluation and optimization mechanisms thereby become more complicated, which usually requires a sophisticated query optimization approach, typically a cost-based one, e.g. [HKWY97, LPL96]. Our experiments in Chapter 7 show that query negotiation, optimization and push-down of VTQL requests actually is very important since it can significantly speed up execution time.

Efficient query execution can also be significantly influenced by the internal representation of VT object instances. VT object instances are handled as explicit entities, which results in memory overhead and performance overhead as it is typical for object-oriented data models and languages. For example, the Java language binding of the VT represents VT object instances as Java objects, which contain additional internal object management information. Moreover, Java objects have to be separately handled in network communication, e.g. transfer of VT objects from the VT host to a client host, which can decrease performance. This is of special importance when we deal with large data sets, e.g. thousands of tuples in an SQL table. Therefore, we have to provide an efficient representation of VT objects so

that memory overhead and performance overhead is reduced. This is achieved by means of the VT cursor concept, which employs similar mechanisms as, for example, a JDBC driver does. A **VT object cursor** is a VT object that is created and managed by the VT and that efficiently handles data. Therefore, a VTQL read request returns a VT object cursor. For example, the VTQL read request on page 83 returns a collection of implicit *VTEmployee* instances that are represented by a VT object cursor, which is defined by VT object class *VTObjectCursor*:

```
public class VTObjectCursor {
  // iterator
  boolean next();
  // get and set
  Object get(String attrName);
  void set(String attrName, Object value);
  // operation structures
  void setOpStruct(String osName, DMap<String, Object> attrValues);
  // access operations
  void read();
  void create(DMap<String, Object> objValues);
  void delete();
  void update();
}
```

A VT object cursor contains the result set of a VTQL read request. It provides a *next()* operation to iterate over the implicit VT object instances of the result set. The *get(...)* operation and the *set(...)* operation allow to read and write the attributes of the VT object instance the cursor currently points to. The *setOpStruct(...)* operation provide a means to initialize the necessary VTO operation structures and the *read*, *delete*, *update* and *create* operations represent the VTO access operations that can be performed on the VT object instances. The *read* operation (re-)executes the cursor, the *delete* operation deletes the implicit VT object instance the cursor currently points to, the *update* operation writes the modified values of the current VT object instance, and the *create* operation inserts a new implicit VT object instance at the position the cursor points to.

A VT object cursor can be created in two ways. The data-oriented alternative is to issue a VTQL read request, e.g.:

```
VTObjectCursor vtempC;
vtempC = VT.execute("SELECT * FROM VTEmployee", true);
```

The operation-oriented alternative is to explicitly define a VT object cursor for the desired VT object class and to perform a *read* operation, which is executed as a *read* VTO access operation:

```
VTObjectCursor vtempC;
vtempC = new VTObjectCursor(VTEmployee.class, true);
vtempC.read();
```

```
while (vtempC.next()) {
  if (vtempC.get("lastname").equals("Wagner"))
      && (vtempC.get("firstname").equals("Frank")) {
    vtempC.delete();
    DMap<String, Object> empVals = new DMap<String, Object>();
    empVals.put("id", 2937);
    empVals.put("lastname", "Wagner");
    empVals.put("firstname", "Frank");
    empVals.put("salary", 2000);
    vtempC.create(empVals);
    ...
    vtempC.set("salary", vtempC.get("salary") * 1.2);
    vtempC.update();
  }
}
```

The cursor, i.e. *vtempC*, contains the result of the *read* operation: *vtempC.read()*, and *vtempC.next()* allows to iterate over the implicit *VTEmployee* instances in the cursor result set and to modify (*vtempC.set("salary", vtempC.get("salary") * 1.2)* and *vtempC.update()*), insert (*vtempC.create(empVals)*) or delete (*vtempC.delete()*) implicit VT object instances.

4.5 Summary

The VT supports both basic processing paradigms, i.e. *data-oriented processing* and *operation-oriented processing*. The VT represents remote data and remote operations as VT objects and allows to handle and process them in both ways. VT objects contain data as well as operations so that VT objects can be used in an operation-oriented manner by calling VT object operations and they can be used in a data-oriented way either by directly dealing with VT objects and their attributes or by means of *VTQL*, a set-oriented query language to retrieve and manipulate VT objects. Moreover, the VT does not only offer operation-oriented processing and data-oriented processing, but additionally allows to handle and process VT objects independent of the processing paradigm of an adapter technology or remote system. A client system, e.g. a middleware system, can choose the access style, i.e. whether it wants to access the VT by means of VTQL requests or by means of VT object operations. The VT transparently processes the desired VTQL requests or VT object operations underneath independent of the access styles of the underlying adapters and remote systems. The VT internally performs a paradigm change from operation-oriented processing to data-oriented processing or vice versa if necessary and thereby provides *paradigm transparency*. The VT realizes paradigm changes by transforming VTQL requests and VT object operations into equivalent CRUD representations, which are realized by means of *VTO access operations* and *VTO operation structures*

Finally, the VT has to be able to transform complex VTQL requests into complex data-oriented requests on remote systems, e.g. SQL requests or XQuery requests,

and it has to be able to compensate for missing request execution capabilities in remote systems or adapter technologies. The VT also has to be able to optimize VTQL requests that span more than one remote system. Therefore, the VT requires a sophisticated query optimization approach to support query optimization, query negotiation and push-down of VTQL requests.

Chapter 5

Deployment Model

Now, we know how the VT data model looks like and how operations and requests are processed in the VT. Next, we show how VT objects are defined and mapped to remote data and remote operations, how adapters are deployed and handled in the VT and how the VT can be applied to integration scenarios in terms of configurations and deployments.

5.1 VT Object Configurations

A **VT object configuration** defines how remote data and remote operations are represented as VT objects and how VT objects and the correlated remote data and remote operations are accessed. The VT uses VT object configurations to properly execute operations and requests on VT objects, e.g. VT object operations and VTQL requests. A VT object configuration consists of four configuration chapters (also see Figure 5.1):

- The **adapter information chapter** defines information about an adapter. The associated adapter manager uses an adapter information chapter to deploy and access the adapter properly, e.g. where to find the adapter libraries or which parameters to apply to the adapter.
- The **system information chapter** defines information about a remote system. An adapter needs a system information chapter to properly access the remote system, e.g. authentication information or connection management information.
- The **object information chapter** defines information about the remote data and the remote operations that are correlated with the VT object defined by the object definition chapter, e.g. which API operation to call or which database table to access in the remote system.
- The **object definition chapter** defines VT object attributes, VT object operations, e.g. VTO access operations, as well as VTO operation structures.

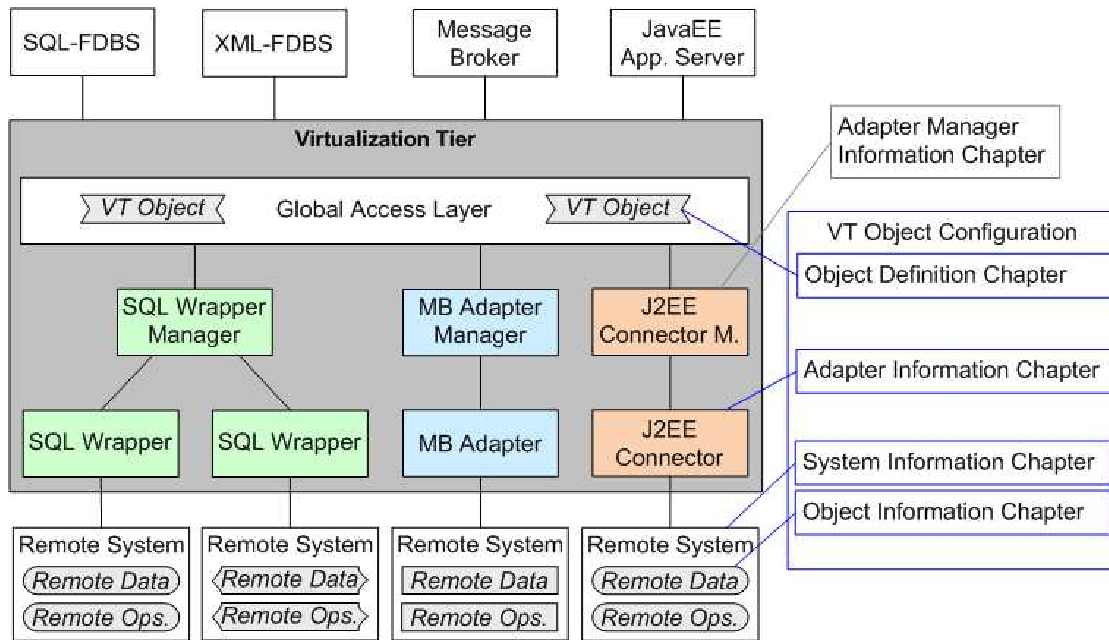


Figure 5.1: VT Object Configuration.

> The object definition chapter also defines how the attributes, operations and operation structures are mapped to the remote data and the remote operations that are defined in the object information chapter.

Additionally, each adapter manager is initially configured by an **adapter manager information chapter**, which specifies information on how to initialize the adapter manager and global properties that hold for all adapters that are associated with this adapter manager. Configuration chapters depend on each other as depicted in Figure 5.2. An adapter manager (*adapter manager information chapter*) is responsible for handling and executing adapters (*adapter information chapter*). An adapter integrates a specific remote system, e.g. a software system or a service, which can be instantiated on different hosts or with different parameters so that the adapter can access different instances of the same remote system (*system information chapter*), e.g. two instances of a DBS on different hosts. Finally, a remote system can contain a number of remote data entities or remote operations (*object information chapter*) that are correspondingly represented as VT objects (*object definition chapter*).

In the next subsections we take a closer look at VT object configurations and their configuration chapters. We use the CRM DBS and VT object *VTEmployee* for the SQL wrapper example (case *W*) and we use the CRM system with the C procedures and data structures and VT object *VTEmp* for the J2EE connector example (case *C*). Figure 5.3 shows the configuration chapter hierarchy of our example scenario.

The SQL wrapper manager and the J2EE connector manager are associated with one adapter, one remote system and two remote data entities, respectively. Config-

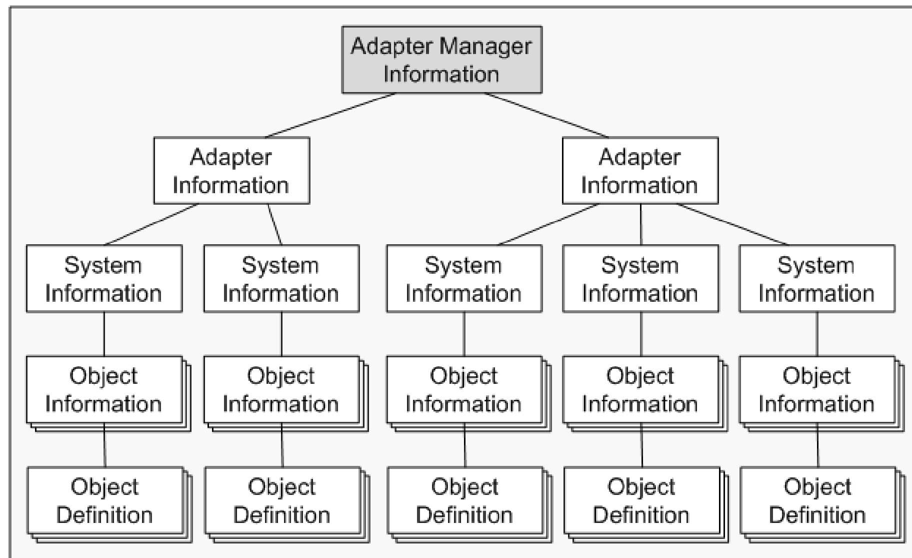


Figure 5.2: Configuration Chapter Hierarchy.

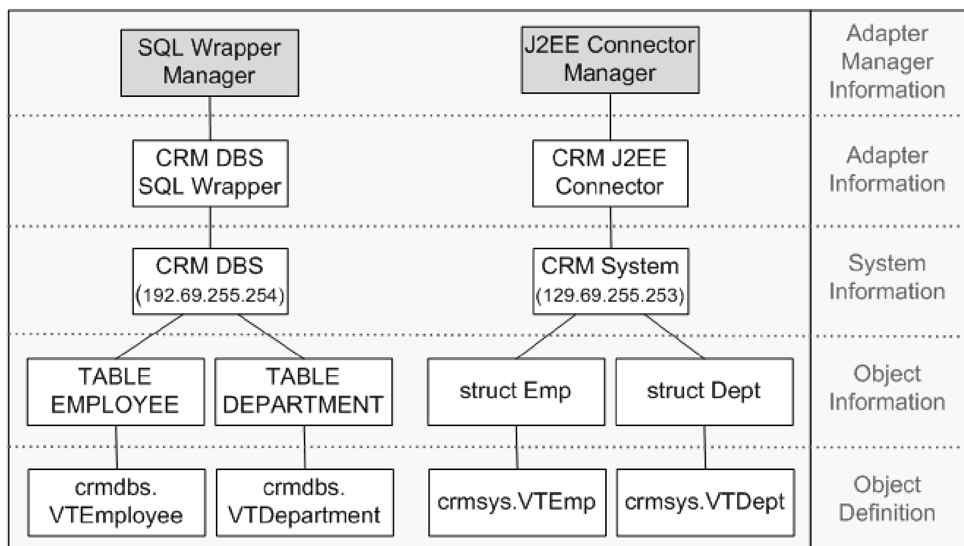


Figure 5.3: Example Configuration Chapter Hierarchy.

uration chapters are expressed in XML. However, we use a graphical representation since it is more compact and easier to understand than the underlying XML representation. Each adapter manager provides a separate configuration schema, i.e. XML schema, defining the properties that can be used in the different configuration chapters. For example, the SQL wrapper manager provides the *options* and *option* properties that are used in the different SQL wrapper statements and the J2EE connector manager provides the *interaction_spec*, *input_record* and *output_record* properties that represent the J2EE connector interaction parts.

5.1.1 Adapter Manager Information Chapter

The SQL wrapper manager information chapter shown in Figure 5.4 specifies that it can execute SQL wrappers that conform to the DB2 II 9 API. The J2EE connector manager information chapter shown in Figure 5.5 specifies that it can execute J2EE connectors of version 1.0. The SQL wrapper manager generally negotiates about VTQL requests and potentially allows push-down of VTQL fragments (*vtql*). The J2EE connector manager categorically rules out negotiating about VTQL requests (*no_vtql*). Hence, the VT has to compensate for VTQL requests that are targeted at remote systems integrated by J2EE connectors.

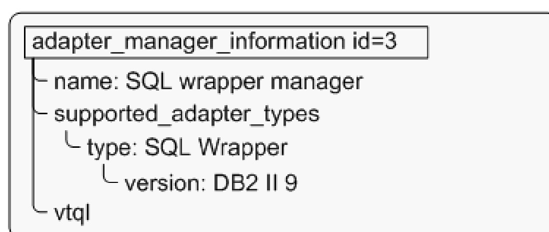


Figure 5.4: SQL Wrapper Manager Information Chapter (case W).

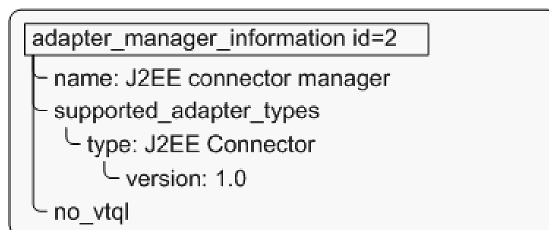


Figure 5.5: J2EE Connector Manager Information Chapter (case C).

5.1.2 Adapter Information Chapter

The SQL wrapper information chapter of the CRM DBS SQL wrapper shown in Figure 5.6 specifies the wrapper name, the employed wrapper base library and an option that defines the wrapper hook. The SQL wrapper information chapter provides the information that is necessary to deploy the SQL wrapper into the VT. This information is the same as it is given by the original DB2 II CREATE WRAPPER statement (see Figure 5.7), which is used to deploy the same SQL wrapper into its native middleware system, i.e. DB2 II.

The J2EE connector information chapter of the CRM J2EE connector shown in Figure 5.8 specifies the connector name and the interfaces and implementation classes that are required for the interaction between application server (or J2EE connector manager) and J2EE connector. Further information concern transaction behavior and security issues. The J2EE connector information chapter provides

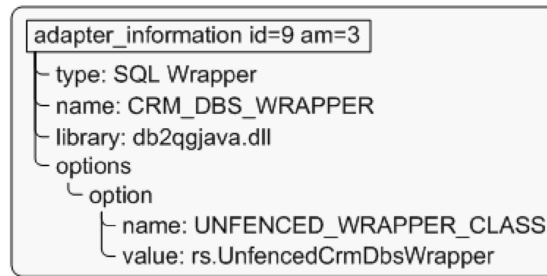


Figure 5.6: SQL Wrapper Information Chapter (case W).

```
CREATE FOREIGN WRAPPER CRM_DBS_WRAPPER
LIBRARY 'db2qgjava.dll'
OPTIONS (
  UNFENCED_WRAPPER_CLASS 'rs.UnfencedCrmDbsWrapper'
)
```

Figure 5.7: SQL CREATE WRAPPER Statement (case W).

the information that is necessary to deploy the J2EE connector into the VT. This information is the same as it is specified by the original J2EE connector deployment in a Java EE application server. Figure 5.9 shows the J2EE deployment descriptor of the CRM J2EE connector. There is additional deployment information about the remote system, which is handled by the system information chapter in the next subsection.

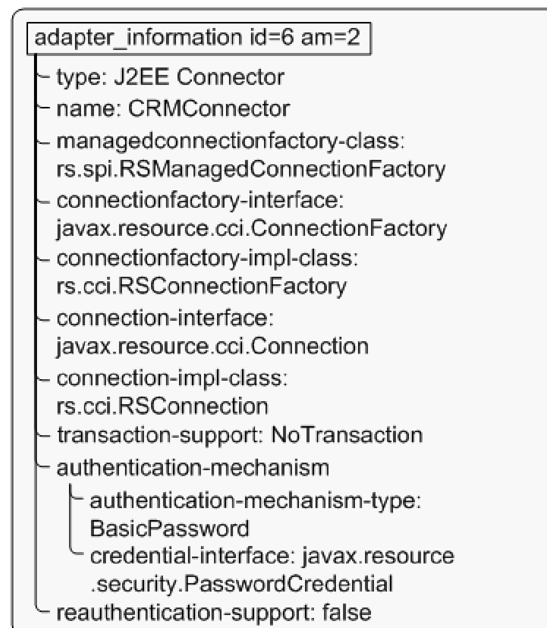


Figure 5.8: J2EE Connector Information Chapter (case C).

```
<managedconnectionfactory-class>
  rs.spi.RSManagedConnectionFactory
</managedconnectionfactory-class>
<connectionfactory-interface>
  javax.resource.cci.ConnectionFactory
</connectionfactory-interface>
<connectionfactory-impl-class>
  rs.cci.RSConnectionFactory
</connectionfactory-impl-class>
<connection-interface>
  javax.resource.cci.Connection
</connection-interface>
<connection-impl-class>
  rs.cci.RSConnection
</connection-impl-class>
<transaction-support>
  NoTransaction
</transaction-support>

<config-property>
  <config-property-name>
    IPAddress
  </config-property-name>
  <config-property-type>
    java.lang.String
  </config-property-type>
  <config-property-value>
    129.69.255.253
  </config-property-value>
</config-property>
<config-property>
  <config-property-name>
    Port
  </config-property-name>
  <config-property-type>
    java.lang.String
  </config-property-type>
  <config-property-value>
    6789
  </config-property-value>
</config-property>
<authentication-mechanism>
  <authentication-mechanism-type>
    BasicPassword
  </authentication-mechanism-type>
  <credential-interface>
    javax.resource.security.PasswordCredential
  </credential-interface>
</authentication-mechanism>
<reauthentication-support>
  false
</reauthentication-support>
```

Figure 5.9: J2EE Connector Deployment (case C).

5.1.3 System Information Chapter

The system information chapter of the SQL wrapper scenario shown in Figure 5.10 specifies the name of the remote system and the host where the CRM DBS instance runs and where the CRM DBS SQL wrapper has to connect to. This information is the same as it is given by the original DB2 II CREATE SERVER statement shown in Figure 5.11.

The system information chapter of the J2EE connector scenario shown in Figure 5.12 analogously specifies the name of the remote system and the host where the CRM system instance runs and where the CRM J2EE connector has to connect to. The same information is also specified by the original J2EE connector deployment descriptor of the CRM J2EE connector shown in Figure 5.9.

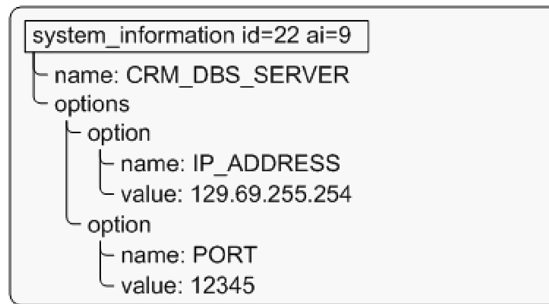


Figure 5.10: CRM DBS Information Chapter (case W).

```
CREATE FOREIGN SERVER CRM_DBS_SERVER
FOR WRAPPER CRM_DBS_WRAPPER
OPTIONS (
  IP_ADDRESS '129.69.255.254',
  PORT '12345'
)
```

Figure 5.11: SQL CREATE SERVER Statement (case W).

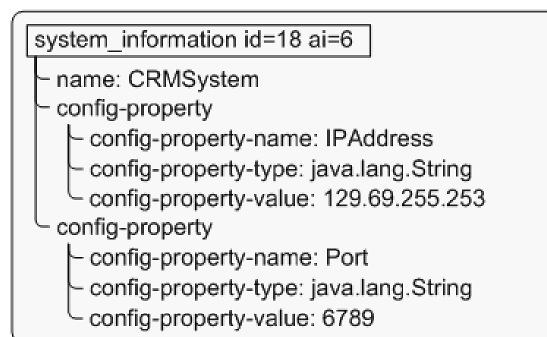


Figure 5.12: CRM System Information Chapter (case C).

5.1.4 Object Information Chapter

The object information chapter of the SQL wrapper scenario shown in Figure 5.13 specifies the name and the columns of one of the tables that the CRM DBS SQL wrapper provide. The *options* elements determine how the CRM DBS SQL wrapper maps the given FDBS table to the corresponding table in the CRM DBS, i.e. *CRM_EMPLOYEE* maps to *EMPLOYEE* (not shown here: *CRM_DEPARTMENT* maps to *DEPARTMENT*). This information is the same as it is given by the original DB2 II CREATE TABLE statements shown in Figure 5.14.

The object information chapter of the J2EE connector scenario shown in Figure 5.15 specifies Java class *rs.RSEmp* (not shown here: Java class *rs.RSDept*), and the possible Java method calls given by the *interaction* elements that can be used to properly access the CRM system via the CRM J2EE connector. The same information is also specified by the original J2EE connector deployment of the CRM J2EE connector and its associated Java classes shown in Figure 5.16.

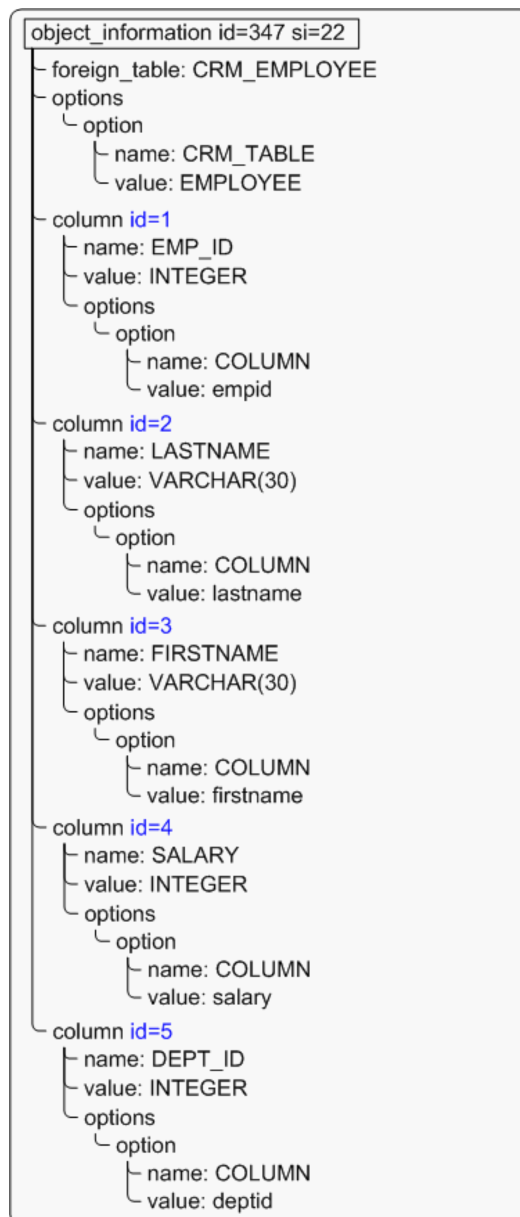


Figure 5.13: CRM_EMPLOYEE Information Chapter (case *W*).

```
CREATE FOREIGN TABLE CRM_EMPLOYEE (
  EMP_ID INTEGER OPTIONS (COLUMN 'empid'),
  LASTNAME VARCHAR(30) OPTIONS (COLUMN 'lastname'),
  FIRSTNAME VARCHAR(30) OPTIONS (COLUMN 'firstname'),
  SALARY INTEGER OPTIONS (COLUMN 'salary'),
  DEPT_ID INTEGER OPTIONS (COLUMN 'deptid')
) FOR SERVER CRM_DBS_SERVER
OPTIONS (
  CRM_TABLE 'EMPLOYEE'
)
```

Figure 5.14: SQL CREATE FOREIGN TABLE Statement (case *W*).

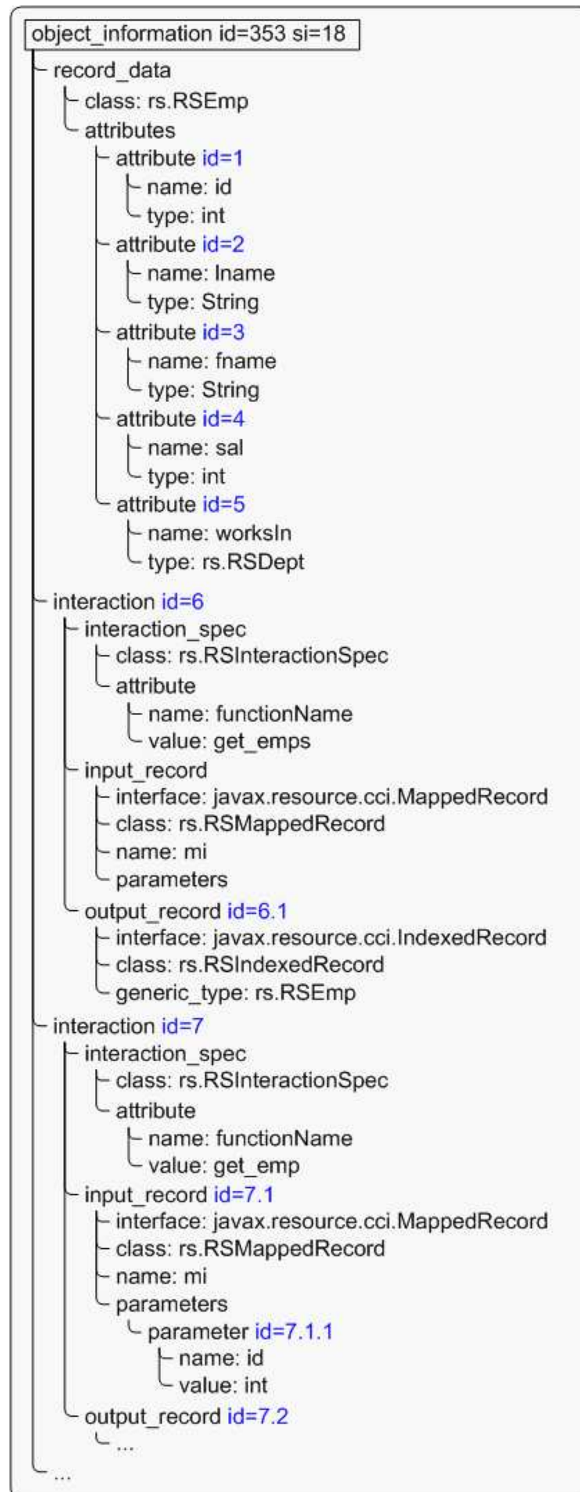


Figure 5.15: RSEmp Information Chapter (case C).

5.1.5 Object Definition Chapter

Adapter information chapters, system information chapters and object information chapters contain the deployment information that is required when an adapter is

```
package rs;
public class RSEmp {
  public int getId() { ... }
  public void setId(int id) { ... }
  public String getLname() { ... }
  public void setLname(String lname) { ... }
  public String getFname() { ... }
  public void setFname(String fname) { ... }
  public int getSal() { ... }
  public void setSal(int sal) { ... }
  public RSDept getWorksIn() { ... }
  public void setWorksIn(RSDept worksIn) { ... }
}
```

```
package rs.cci;
public class RSIndexedRecord
implements IndexedRecord
{
  public int size() { ... }
  public Object get(int index)
  ...
}
```

```
package rs.cci;
public class RSMappedRecord
implements MappedRecord
{
  ...
}
```

```
package rs.cci;
public class RSInteractionSpec implements InteractionSpec
{
  ...
  public void setFunctionName(String functionName) { ... }
  public String getFunctionName() { ... }
}
```

Figure 5.16: Java Classes of the J2EE Connector (case *C*).

deployed into its native middleware system. Now, we use object definition chapters to map remote data and remote operations to VT objects. This mapping is intended to syntactically homogenize data and operations of diverse remote systems and to uniformly access them via VT objects. Further mapping issues, especially structural or semantic mappings can be additionally applied on top of the defined VT objects (also see Section 5.4). A VT object class consists of attributes, i.e. data structures, VT object operations, e.g. VTO access operations, and VTO operation structures.

The object definition chapter of the SQL wrapper scenario shown in Figure 5.17 defines VT object class *crmdbs.VTEmployee* (not shown here: VT object class *crmdbs.VTDepartment*). All VTO operation structures (beneath element *access*) are empty, i.e. they don't have parameters, which means that the VTO access operations do not require further parameterization. The mapping identifiers (*mid*) determine, which elements of the associated object information chapter are mapped to the elements in the object definition chapter. More complex mappings are explicitly expressed by the *mapping* element, e.g. mapping the foreign key relationship between *CRM_EMPLOYEE* and *CRM_DEPARTMENT* to reference attribute *dept* of type *VTDepartment* in *VTEmployee*. The reference attribute, i.e. attribute *dept*, also contains an empty *ref_op* and *ref_key* element, which means that the default

read VTO operation structure (element *default*) and the default identity (element *identity*) are used to resolve the reference.

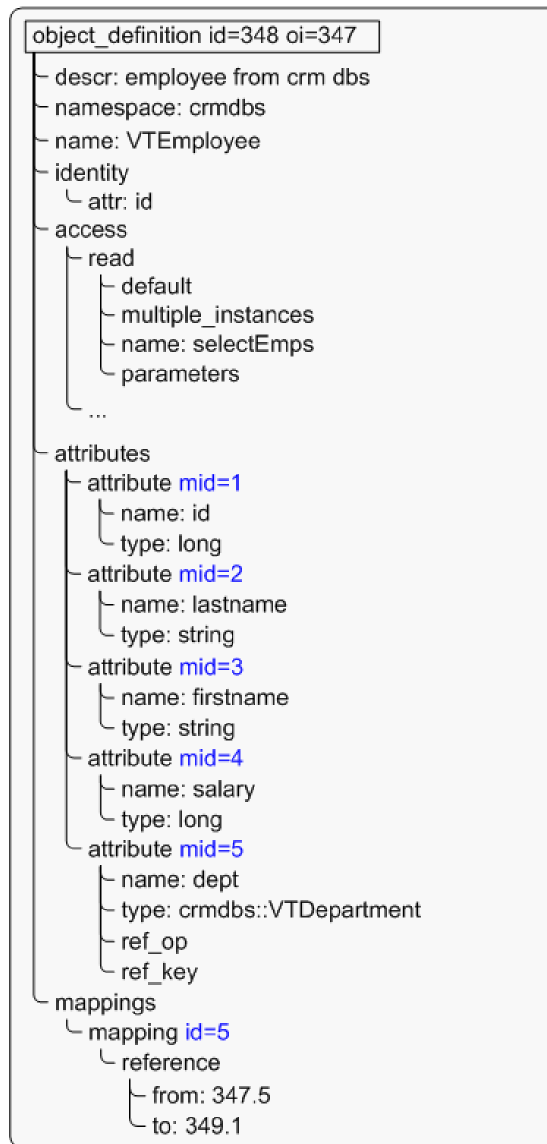


Figure 5.17: VTEmployee Object Definition (case *W*).

The object definition chapter of the J2EE connector scenario shown in Figure 5.18 defines VT object class *crmsys.VTEmp* (not shown here: VT object class *crmsys.VTDept*). Its attributes map to the attributes of the Java class in the associated object information chapter by means of the mapping identifiers (*mid*). The VTO operation structures (element *access*) conform to the definitions of the interaction elements in the associated object information chapters and the mappings in the *mapping* elements determine the exact mapping of the parameters. For example, VTO operation structure *getAllEmps* does not have attributes and only maps the output record of the *interaction* element in the associated object information chapter

to the result parameter of the *read* VTO access operation. VTO operation structure *getEmp* works analogously but additionally contains an attribute, i.e. *empId*, which is mapped to the input record in the associated *interaction* element, i.e. the parameter with name *id*.

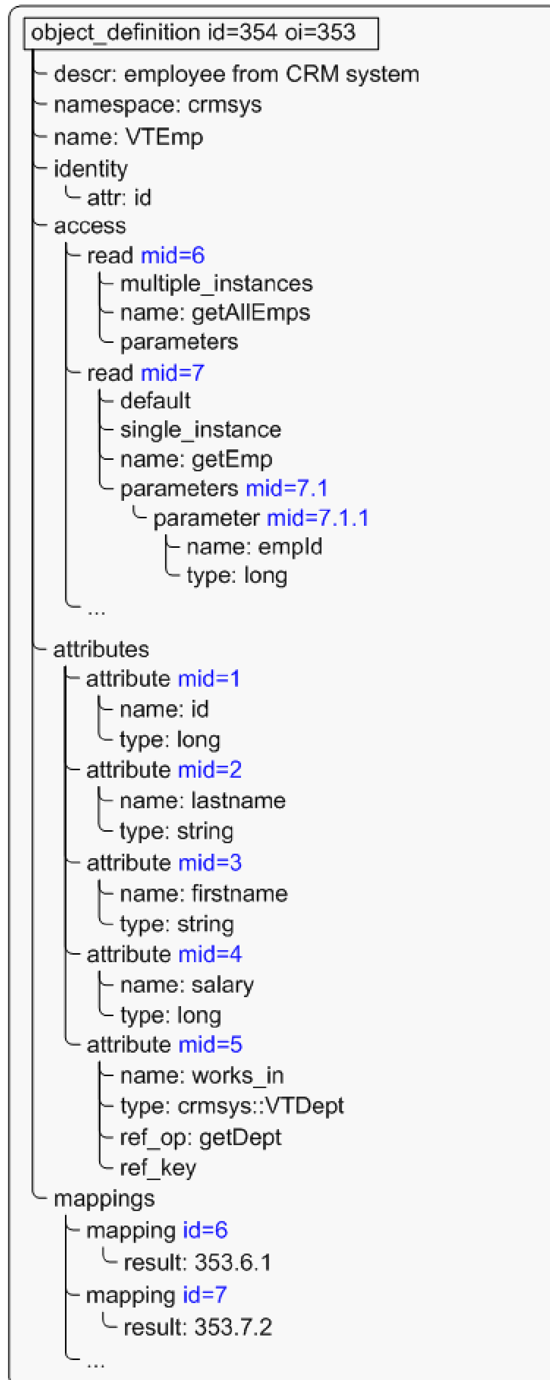


Figure 5.18: VTEmp Object Definition (case C).

5.2 Example Requests

Next, we discuss two example requests of our example scenario, e.g. see Figure 2.6, to show how the overall processing in an integration scenario works. One example request uses the CRM DBS SQL wrapper in the VT and the other one uses the CRM J2EE connector. The first request processing is illustrated in Figure 5.19.

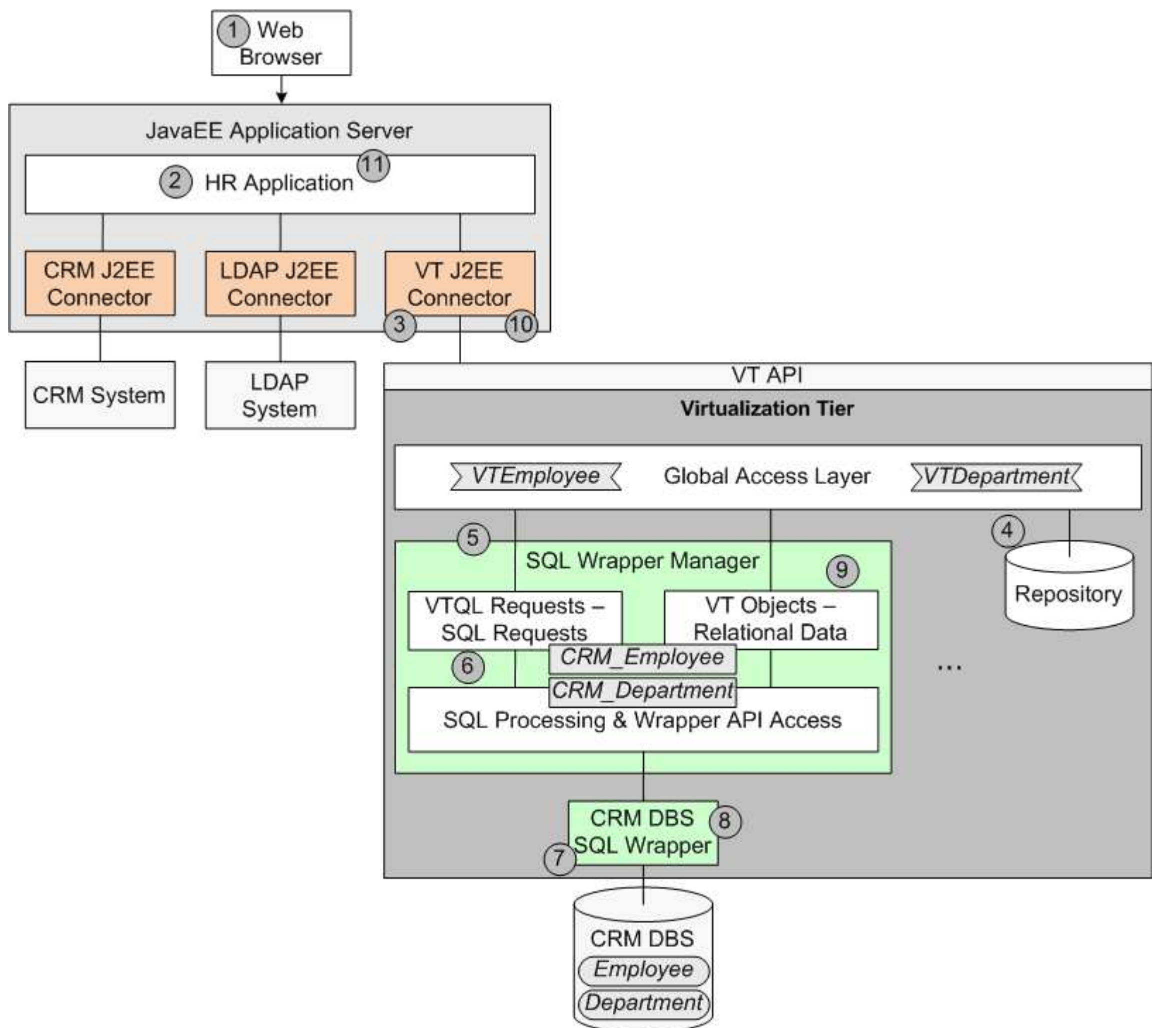


Figure 5.19: Request Processing (SQL Wrapper Example).

1. A user opens a web browser and logs in to the HR application via a web page. The user wants to access some employee information about Frank Wagner and submits a corresponding HTML form request to the HR application.

2. The HR application calls an EJB method to resolve the desired employee information. The EJB in turn issues an interaction request to the VT J2EE connector. An interaction request is a Java method call with input parameters

and output parameters. There are two different means of interactions: either to issue a VTQL request or to call a VT object operation. If we want to call a VT object operation, the EJB would have to issue an interaction with the following parameters:

- Input parameter *access_op* and value *read* for the VTO access operation.
- Input parameter *obj_name* and value *VTEmployee* for the VT object to access.
- Output parameter *emps*, which contains all *VTEmployee* instances.

This will yield a quite inefficient processing since the *read* VTO access operation does not use additional parameters and therefore the interaction will retrieve all employee information from the CRM DBS and returns them to the HR application, which in turn has to select the desired *VTEmployee* instance.

An efficient means is to use a VTQL request so that we drive an interaction with one input parameter and one output parameter:

- Input parameter *vtql_req* and value

  ```
  SELECT *
  FROM   VTEmployee
  WHERE  lastname = 'Wagner' AND firstname = 'Frank'
  ```

- Output parameter *emps* now contains only the desired tuples since we do not have to process and transfer the whole *CRM_Employee* table from the CRM DBS via the VT up to the application server. Consequently, the HR application does not need to perform any intermediate processing.

3. Next, the VT J2EE connector submits the VTQL request given by this interaction request.

4. The VT checks the repository for the VT object configuration of *VTEmployee* and then identifies the SQL wrapper manager as the responsible adapter manager (via object definition chapter, object information chapter, system information chapter and finally adapter information chapter, also see Section 5.1).

5. The VT hands over the VTQL request to the SQL wrapper manager, which actually is capable of handling it (which would not be the case for the J2EE connector manager so that the VT would have to compensate for it).

6. The SQL wrapper manager now translates the VTQL request into an SQL query accessing foreign table *CRM_Employee*:

```
SELECT *
FROM CRM_EMPLOYEE
WHERE LASTNAME='Wagner' AND FIRSTNAME='Frank'
```

7. The CRM DBS SQL wrapper translates the SQL query into a suitable request for the remote system, which incidentally is an SQL DBS, i.e. the CRM DBS. Therefore, we have to translate the SQL query into an SQL query for the CRM DBS, which is quite similar since we applied a direct and natural mapping from the SQL wrapper to the CRM DBS in the VT object configuration of *VTEmployee*:

```
SELECT *
FROM EMPLOYEE
WHERE lastname='Wagner' AND firstname='Frank'
```

8. The CRM DBS executes the SQL query and returns the selected employee tuples to the CRM DBS SQL wrapper, which can directly map the *EMPLOYEE* tuples to tuples of *CRM_EMPLOYEE* as the result of the SQL query issued by the SQL wrapper manager in step 6.

9. Next, the SQL wrapper manager transforms the tuples of *CRM_EMPLOYEE* into VT object instances according to the *VTEmployee* object definition chapter as requested in step 5 and returns them to the VT.

10. The VT transfers the VT object instances to the calling VT J2EE connector, which transforms them into suitable Java objects in output parameter *emps* of the J2EE connector interaction in step 3.

11. Finally, the EJB processes the employee Java objects and the enterprise application creates a suitable HTML response document answering the initial user request.

If we access the file server via the file server SQL wrapper (also see Figure 2.6), the example request would work in the same way except that the file server SQL wrapper would transform the SQL query issued by the SQL wrapper manager into corresponding file access operations.

The second request processing uses the CRM J2EE connector in the VT and works analogously although quite different to the first request processing with the SQL wrapper (see Figure 5.20). Actually, it could be a better option to directly deploy the CRM J2EE connector into the Java EE application server instead of the VT, we just use it for a discussion of a typical example processing and show the general usage of the VT and its mapping and processing architecture.

1. The user now wants to access some other employee information about Frank Wagner, which is located in another remote system, i.e. the CRM system, and which we integrated via the CRM J2EE connector into the VT. Therefore, the user submits a corresponding HTML form request to the HR application.

2. The HR application again calls an EJB method to resolve the desired employee information.

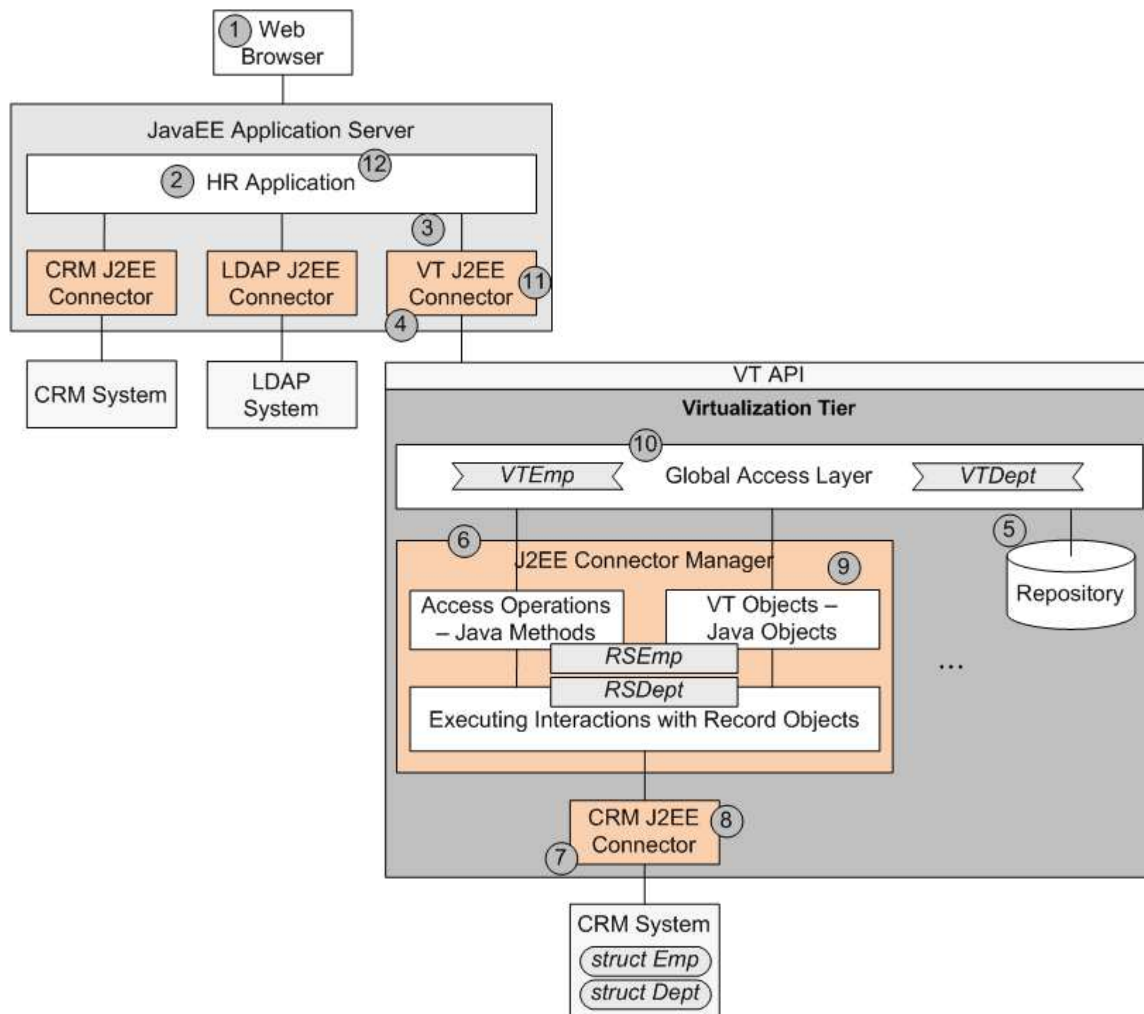


Figure 5.20: Request Processing (J2EE Connector Example).

3. The EJB in turn issues an interaction request to the VT J2EE connector that contains a suitable VTQL request in the *vtql_req* interaction input parameter:

```
SELECT *
FROM VTEmp.getAllEmps()
WHERE lastname = 'Wagner' AND firstname = 'Frank'
```

   Output parameter *emps* this time contains *VTEmp* instances which are retrieved from the CRM system.

4. The VT J2EE connector extracts the VTQL request from the interaction and submits it to the VT.

5. The VT checks the repository for the VT object configuration of *VTEmp* and then identifies the J2EE connector manager as the responsible adapter manager (again, via object definition chapter, object information chapter, system

information chapter and finally adapter information chapter, also see Section 5.1). This time, the VT does not hand over the VTQL request to the J2EE connector manager since J2EE connectors cannot handle VTQL requests (also see Figure 5.5). Therefore, the J2EE connector manager can only submit the *read* VTO access operation and the *read* VTO operation structure to the CRM J2EE connector: *VTEmp.getAllEmps()* and *VTEmp.read()*.

6. The J2EE connector manager translates the *read* VTO access operation into a suitable interaction request according to the object information chapter (see Figure 5.15) with *functionName get_emps*, an empty input record and an output record that contains *RSEmp* Java objects.

7. The CRM J2EE connector in turn translates the interaction request into a C procedure call for the CRM system: *get_emps()*.

8. The CRM J2EE connector translates the returned C data structures, i.e. *struct Emp*, into *RSEmp* Java objects and returns them via the interaction output record to the J2EE connector manager.

9. The J2EE connector manager transforms the *RSEmp* Java objects into corresponding *VTEmp* instances and hands them over to the VT.

10. The VT now first has to compensate for the rest of the VTQL request, i.e. the WHERE clause.

11. Then the VT transfers the remaining VT object instances to the calling VT J2EE connector, which transforms them into suitable Java objects in output parameter *emps* of the J2EE connector interaction in the Java EE application server.

12. Finally, the EJB processes these employee Java objects and the enterprise application creates another HTML response document answering the initial user request.

Also refer to Chapter 4 for further details about the different means of processing requests in the VT.

5.3 Deployment Process

The skills required to specify VT object configurations, i.e. to define VT objects and to deploy adapters into the VT, comprise knowledge about the VT, i.e. about the VT data model and the VT processing model, as well as knowledge about the involved adapter technologies. Clearly, the complexity of the whole deployment process must be reduced to make the VT practically manageable. Therefore, the goal is to divide the deployment process into two steps performed by different persons (as depicted in Figure 5.21): adapters are deployed in the first step (1) and suitable VT objects

are defined in the second step (2). Deploying an adapter into the VT requires the same deployment information as if the adapter is deployed into its native middleware system (see Section 5.1).

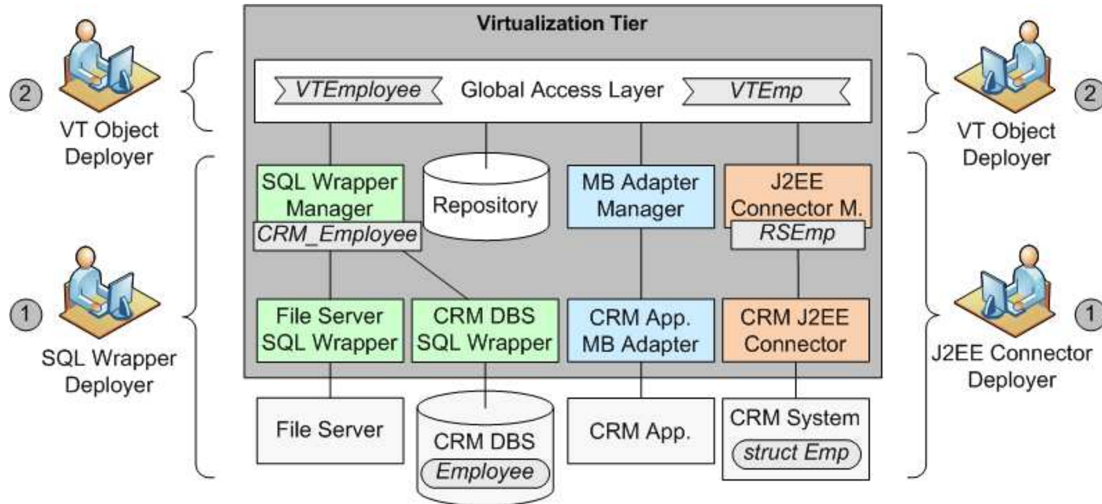


Figure 5.21: Deployment Responsibilities.

An **adapter deployer** is a person who is responsible for deploying adapters into a middleware system, e.g. a J2EE connector deployer or an SQL wrapper deployer. An adapter deployer has to be familiar with the middleware system, i.e. he knows how to deploy adapters and how to define middleware-specific data and operations that are representing data and operations in the integrated remote systems. Therefore, the first step of the deployment process is performed by the corresponding adapter deployer. The adapter deployer deploys an adapter into the VT (*adapter information chapter*), correlates a remote system with the adapter (*system information chapter*) and specifies suitable middleware-specific data definitions and operation definitions (*object information chapter*). The information specified by the adapter information chapter, the system information chapter, and the object information chapter corresponds to the information that an adapter deployer has to specify if he wants to deploy an adapter, e.g. an SQL wrapper or a J2EE connector, into the corresponding middleware system, e.g. an FDBS or a Java EE application server.

For example, an FDBS administrator deploys the CRM DBS SQL wrapper into the FDBS (see right part of Figure 1.1) and defines information about the CRM DBS and the integrated CRM DBS tables by means of the SQL statements in Figures 5.7, 5.11 and 5.14. The FDBS administrator is most suitable to act as an SQL wrapper deployer in the VT scenario as shown in Figure 5.22 where he has to specify the same information as in the FDBS scenario, i.e. he deploys the CRM DBS SQL wrapper into the VT (*adapter information chapter* in Figure 5.6) and defines information about the CRM DBS (*system information chapter* in Figure 5.10) and the CRM DBS tables to be integrated into the VT (*object information chapter* in Figure 5.13).

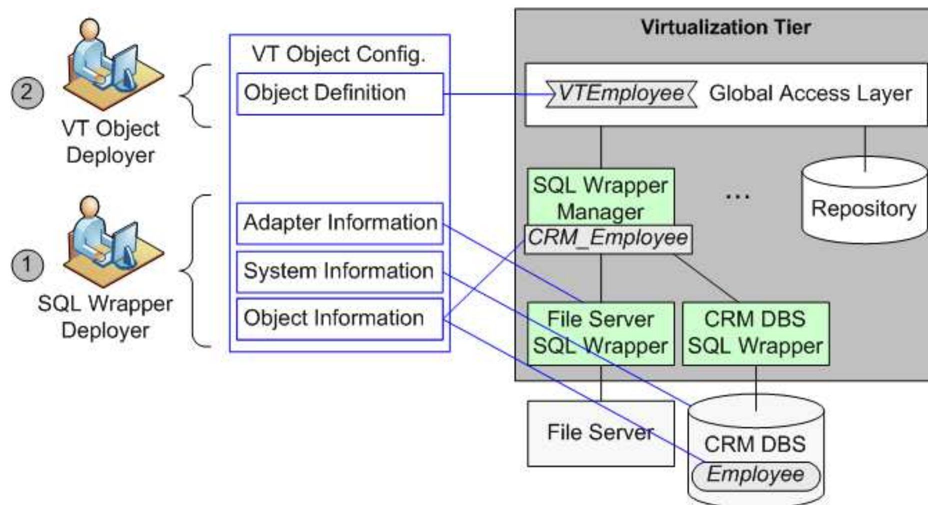


Figure 5.22: Deployment of an SQL Wrapper.

The J2EE connector scenario works analogously. The Java EE administrator deploys the CRM J2EE connector into the Java EE application server (see left part of Figure 1.1) by means of the deployment information shown in Figures 5.9 and 5.16. The Java EE administrator is most suitable to act as a J2EE connector deployer in the VT scenario as shown in Figure 5.23 where he has to specify the same information as in the Java EE scenario, i.e. he deploys the CRM J2EE connector into the VT (*adapter information chapter* in Figure 5.8) and defines information about the CRM system (*system information chapter* in Figure 5.12) and the procedures and data structures of the CRM system to be integrated into the VT (*object information chapter* in Figure 5.15).

Again, the information that is used for an adapter deployment in the VT, i.e. the configuration chapters, is the same information that is used for an adapter deployment in the respective middleware system. An adapter deployer does not need further skills or knowledge beyond that. He only uses the adapter deployment facility of the VT to enter the same information as he would do with the middleware system.

A **VT object deployer** performs the second step of the deployment process, i.e. to define VT objects by means of *object definition chapters*. She creates VT object definitions according to the middleware-specific data definitions and operation definitions specified by adapter deployers during the first step. For example, the VT object deployer creates the VT object definitions in Figures 5.17 and 5.18 according to the object information chapters defined by the adapter deployers (see Figures 5.13 and 5.15). The VT object deployer does not have to specify VT object definitions from scratch. Adapter managers comprise functionality to automatically generate basic VT object definitions from given object information chapters. The VT object deployer can then start on such a basic VT object definition and further customizes it if necessary. Thereby, the VT object deployer does not need to know much about the different adapter technologies.

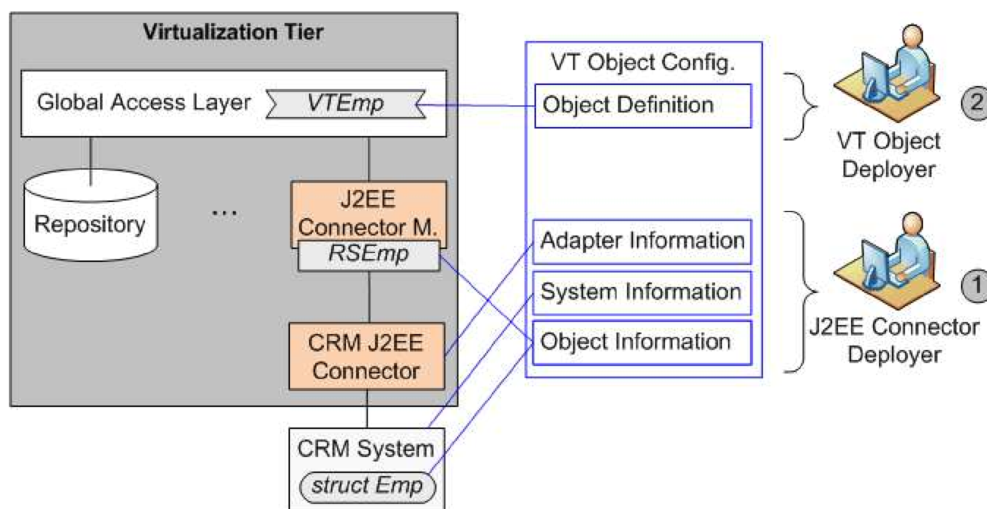


Figure 5.23: Deployment of a J2EE Connector.

In this way, the deployment process in the VT is significantly alleviated and becomes practically manageable: the adapter deployer is doing his job as usual. He is only concerned with the middleware-specific part of VT object configurations, i.e. adapter deployment and definition of middleware-specific data and operations, but he does not need further knowledge about the VT. The VT object deployer is only concerned with the VT-specific part, i.e. VT object definitions, but she does not need to know about adapter technology-specific deployment tasks.

5.4 Mapping Issues

The mapping from object information chapters to object definition chapters as exemplified in Section 5.1 is intended to provide a uniform representation of remote data and remote operations in the VT. More complex mappings that are typically performed by structural or semantic mapping approaches are not subject of VT object configurations. Future work could enhance the mapping between object information chapters and object definition chapters to additionally employ algorithms and approaches in the area of schema matching, schema mapping and schema integration (see Figure 5.24). A schema integration facility on top of the existing VT object configurations allows to apply additional mapping configurations. The global access layer provides the original VT objects as usual according to the mappings in the VT object configurations. A **schema integrator** can create additional mapping configurations based on the original VT objects. The mapping configurations rely on advanced schema integration techniques and map the original VT objects to new VT object definitions in global schemas and finally in export schemas (views). VT client applications then can deal with VT objects more suitable and more comfortable in a global schema or in an export schema. An overview of advanced schema matching approaches and mapping issues can be found in [RB01, Noy04].

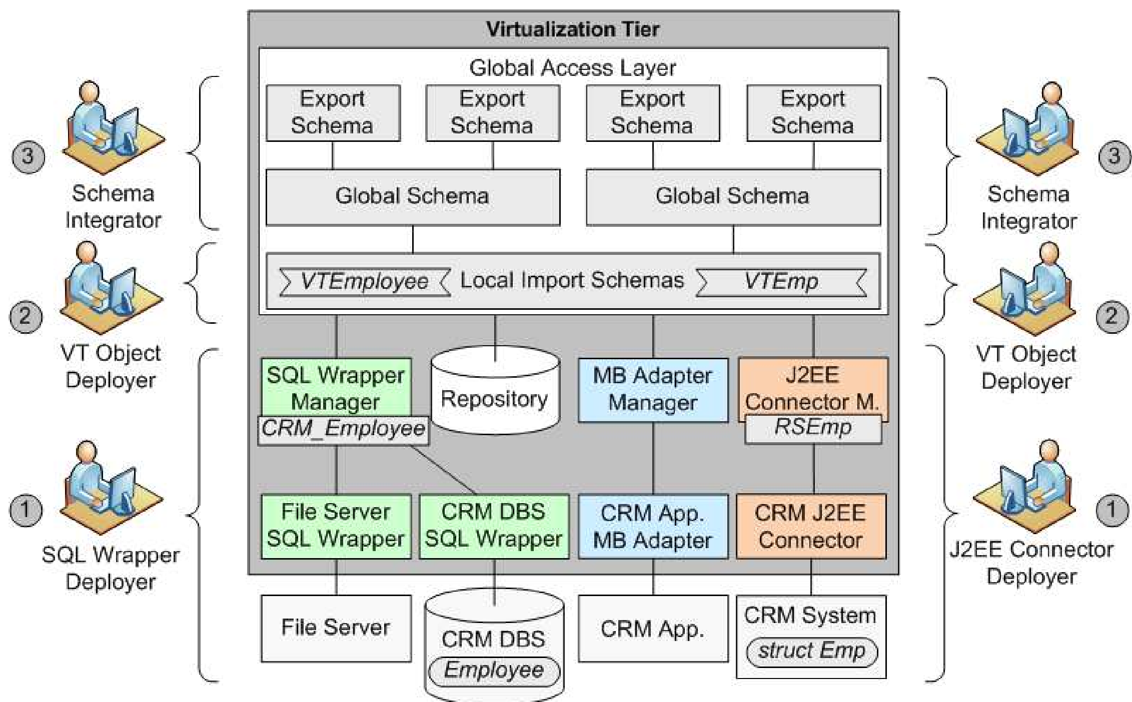


Figure 5.24: Extended Deployment – VT Object Mapping Issues.

5.5 Reuse of Middleware Infrastructure

The co-existence of middleware infrastructures and VT bears optimization potential in terms of eliminating redundancies by applying different levels of reuse.

5.5.1 Reuse of Adapter Deployments

So far, we discussed that the content of adapter deployments in the VT is the same as the content of adapter deployments in the respective native middleware system. Therefore, adapter deployers can easily deploy adapters into the VT since they only need to specify the same information about the adapters as if they deploy the adapters in the respective middleware system. An important conclusion from this point is that we create and manage redundant adapter deployment information, once in the original middleware system and once in the VT. If we deploy an adapter only in the VT, we don't have this problem, but if we want to reuse an adapter that is already deployed and used in a middleware system, we have to come up with a second deployment of this adapter in the VT. If we deploy the same adapter a second time, changes to one adapter deployment have to be applied to the other adapter deployment too. This process is not automated and thus error-prone and it requires additional organization and communication across different administration responsibilities. It also requires additional work and thus more personnel. Therefore, it would be better to have only one adapter deployment. But if we already have adapter deployments in a middleware system and if we also want to reuse these

adapters in the VT, we also need two or more adapter deployments. For that reason, we provide a more effective and efficient way of handling redundant adapter deployment information.

Automatic Extraction & Transformation

Let us come back to our integration scenario in Figure 1.1 where the Java EE application server has to access the file server and the CRM DBS. There are only SQL wrappers, the necessary J2EE connectors are not available. The originally proposed solution shown in Figure 2.6 requires that the FDBS administrator deploys the file server SQL wrapper and the CRM DBS SQL wrapper a second time, i.e. into the VT. However, we do not want to do the same work a second time as already discussed above. We just want to take the deployments from the FDBS and transfer them in a suitable manner to the VT such that all necessary configuration chapters are automatically created.

The general process of transferring adapter deployments is shown in Figure 5.25 where the deployment information is extracted from the FDBS, transformed into configuration chapters, i.e. adapter information chapters, system information chapters, object information chapters, and automatically generated object definition chapters that are derived from the object information chapters. Finally, the generated configuration chapters are applied to the VT resulting in proper adapter deployments of the file server SQL wrapper and the CRM DBS SQL wrapper. The dashed box on the left side of Figure 5.25 represents the original deployment information of the SQL wrappers, remote systems and SQL tables. The dashed box on the right side represents the configuration chapters in the VT, which are derived from the original deployment information on the left side. This basically allows the Java EE application server to access the same entities on the file server and in the CRM DBS as the FDBS does.

Deployment Transformation Wizards

The deployment transformation process is realized by means of *deployment transformation wizards*, which are associated with adapter managers. An adapter manager provides a generic plug-in API so that deployment transformation wizards can be associated with an adapter manager. The VT repository stores all registered wizards and a VT administrator can choose among them to start a transformation process. In our example, an FDBS deployment transformation wizard is associated with the SQL wrapper manager. The transformation process here works as follows:

- A VT administrator starts the wizard, enters the IP address and the port number of the FDBS, a login name and a password, and some other information which is necessary for properly accessing the FDBS.

- The wizard then connects to the FDBS, looks up the FDBS catalog and displays the SQL wrappers deployed in the FDBS as well as the registered remote

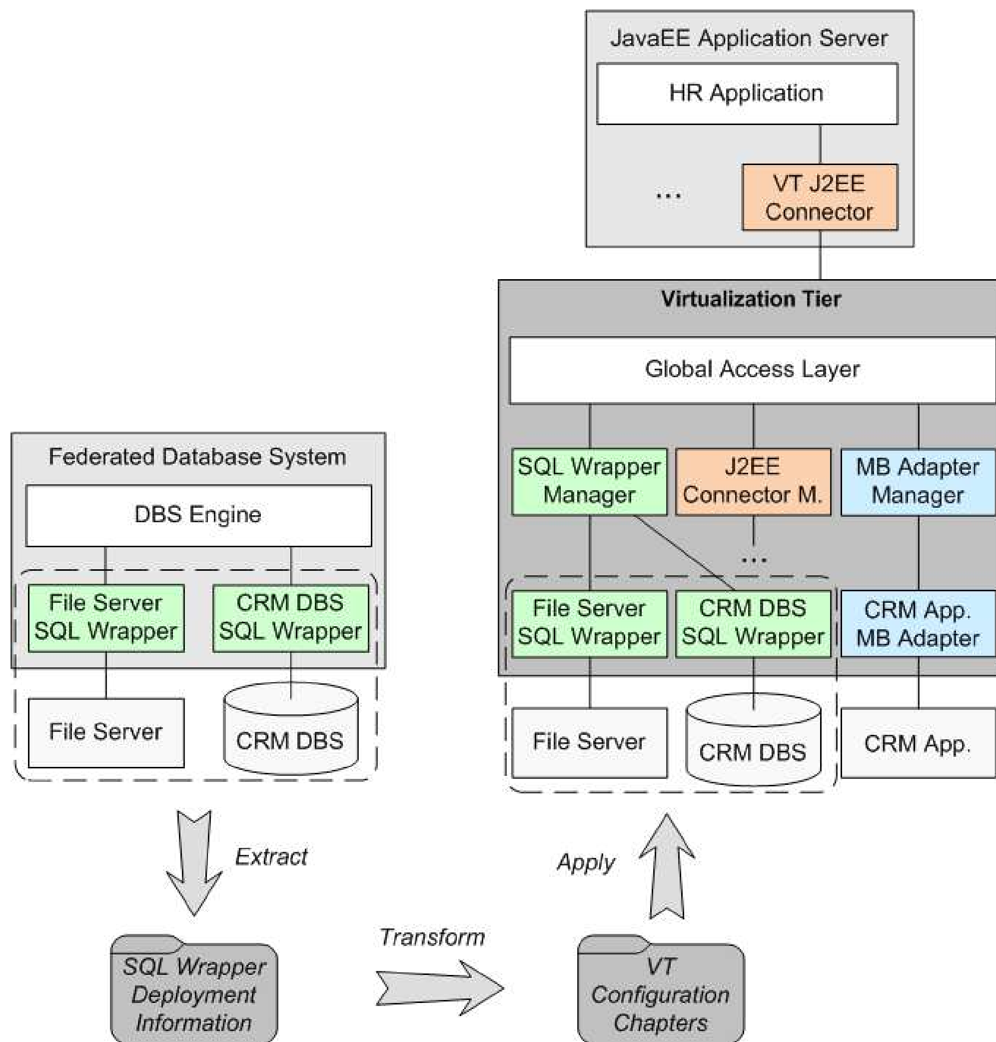


Figure 5.25: Reusing Adapter Deployments.

systems and the tables that represent data and operations of the remote systems (which have been defined by SQL statements like the ones shown in Figures 5.7, 5.11 and 5.14).

- The VT administrator selects the entries of the file server SQL wrapper and the CRM DBS SQL wrapper, the file server and the CRM DBS as well as some tables that represent files on the file server and tables in the CRM DBS. Finally, the VT administrator starts the extraction process.

- The wizard retrieves the deployment information of the two SQL wrappers, the deployment information of the two remote systems and the deployment information of the selected tables from the FDBS catalog.

- The wizard transforms the retrieved FDBS catalog data into suitable configuration chapters (like the ones shown in Figures 5.6, 5.10 and 5.13).

- The wizard transfers the wrapper libraries of the two SQL wrappers and other resources that are required for proper SQL wrapper execution, e.g. remote system API libraries, to the VT host.

- The wizard deploys the configuration chapters into the VT. This deploys the two SQL wrappers into the VT, registers the two remote systems in the VT and creates VT object definitions analogous to the tables in the FDBS.

Another example of a transformation wizard is the J2EE connector manager with a registered Java EE application server deployment transformation wizard that extracts J2EE connector deployments from a Java EE application server, transforms them into configuration chapters and finally deploys them into the VT.

Changes in correlated adapter deployments, i.e. in a middleware system or in the VT, can be monitored by deployment transformation wizards and thereby maintained in the same way as deployments are generated. An adapter deployment in the VT that has been extracted from an adapter deployment in a middleware system then can be automatically modified by the deployment transformation wizard if the adapter deployment in the middleware system is modified (and vice versa). The result of this transformation process is that redundant adapter deployments are automatically generated and maintained. They do not need to be manually handled any longer. The adapter deployer can completely stay in the middleware environment and use the native middleware's deployment facility. He does not need to use the VT deployment utility to deploy an adapter from scratch into the VT because a VT administrator can use a suitable deployment transformation wizard to automatically import adapter deployments from the middleware system. In contrast to the automatic transformation process, the solution in Figure 2.6 requires new adapter deployments even if corresponding deployments already exist in a middleware system. This introduces the disadvantages discussed above: manual, error-prone changes in redundant adapter deployments, additional organization and communication across different administration responsibilities, additional administrative work and personnel. Automatic extraction avoids these disadvantages and provides a more effective and efficient way of handling redundant adapter deployments.

5.5.2 Reuse of Middleware Infrastructure

If an adapter is automatically deployed into the VT by means of a deployment transformation wizard, still two adapter instances of the same adapter are available in the IT infrastructure, one adapter instance in the original middleware system and one adapter instance in the VT. That is, the VT has to incorporate functionality for executing the adapter, which basically is the same functionality that is required for executing the adapter in the middleware system. Put in other words, we use redundant middleware infrastructure, one in the original middleware system and one in the VT.

Redundant Middleware Infrastructure

For example, the scenario in Figure 5.25 allows to access a file server SQL wrapper instance via the FDBS (shown on the left), but it also allows to execute the other file server SQL wrapper instance via the VT (shown on the right). The same holds for the CRM DBS SQL wrapper. This kind of redundant middleware infrastructure is not always intended from a global viewpoint. Intended redundant middleware infrastructure is used for purposes such as high availability, replication or distribution of computation, which is not the case here. The usage of redundant middleware infrastructure in cases like the given integration scenario leads to some disadvantages:

- higher software costs if functionality is implemented twice,
- higher hardware costs since more hardware is needed for executing the additional software, i.e. the redundant functionality,
- higher maintenance costs since this software and hardware also has to be maintained.

Hence, we don't want to provide a second middleware infrastructure for executing adapters if we already have one. It is beneficial to reuse the existing middleware infrastructure for executing adapters. For example, we would like to use the FDBS for executing the file server SQL wrapper and the CRM DBS SQL wrapper. Therefore, we employ an FDBS VT adapter to integrate the FDBS into the VT as shown in Figure 5.26. The dashed box indicates the middleware infrastructure that we reuse in this scenario. The VT uses the FDBS VT adapter to access the FDBS, which in turn executes the file server SQL wrapper and the CRM DBS SQL wrapper since they are deployed only in the FDBS, but no longer redundantly in the VT. In contrast, Figure 5.25 shows the former solution where the VT uses the SQL wrapper manager to execute the file server SQL wrapper and the CRM DBS SQL wrapper directly, i.e. in the VT.

The VT adapter technology is not necessary for the VT to work as an IM system, but it provides some additional flexibility for integration issues such as the integration of middleware systems. Thus we can develop future adapters as VT adapters that directly fit into the VT and that directly support the VT architecture, which makes integration processes more efficient and uniform. Therefore, we use the VT adapter technology to natively and thereby more efficiently integrate any kind of remote system, e.g. other middleware systems, into the VT. VT adapters natively cope with VT objects, VT object operations and VTQL requests, which is not necessarily supported by all other adapter managers, e.g. the J2EE connector manager does not execute VTQL requests, but the VT has to compensate for them.

VT Adapter Deployments

Reusing the FDBS infrastructure leads to a different adapter deployment than directly reusing the SQL wrappers. The VT adapter manager and the FDBS VT

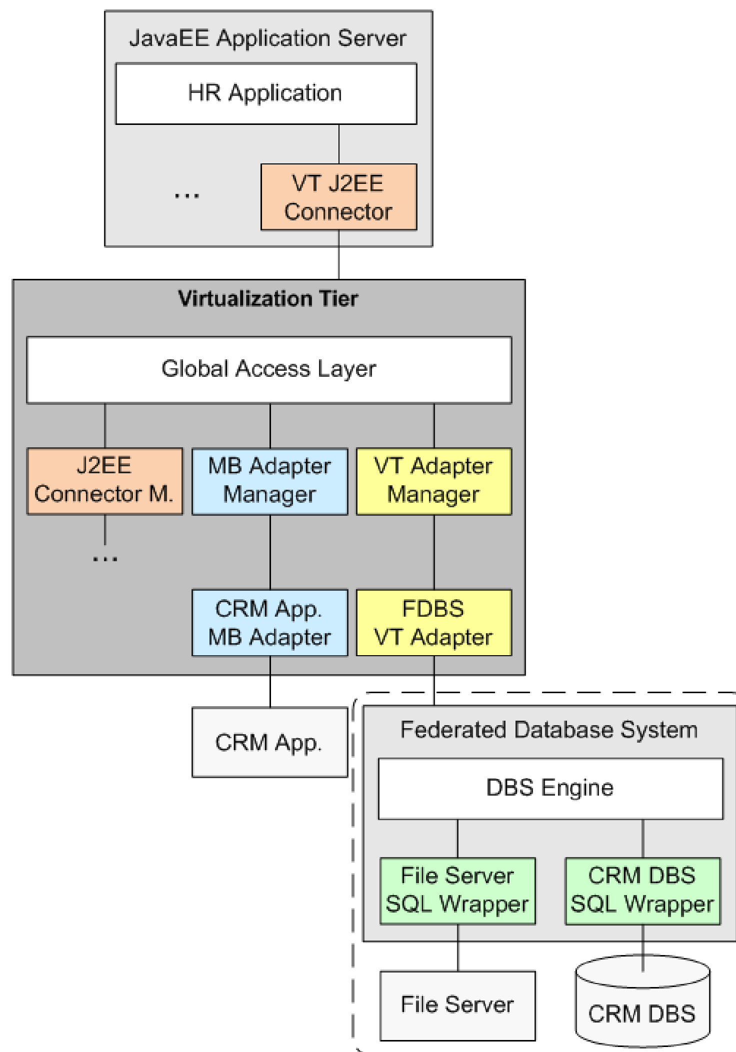


Figure 5.26: Reusing an FDBS Middleware Infrastructure.

adapter use different configuration chapters than the SQL wrapper manager and the corresponding SQL wrappers do. The VT adapter manager and the FDBS VT adapter do not need to know about how SQL wrappers are executed because the FDBS is executing the SQL wrappers. Thus, the VT administrator creates an adapter information chapter for the FDBS VT adapter as shown in Figure 5.27. The file server and the CRM DBS as well as the files on the file server and the tables in the CRM DBS are no longer visible to the VT, but only to the FDBS, which is responsible for properly accessing them via the file server SQL wrapper and the CRM DBS SQL wrapper and which represents them as foreign tables in the FDBS. Therefore, the system information chapters and object information chapters of the integration scenario in Figure 5.25 are not applicable here. Instead, the VT only accesses the FDBS so that we need a system information chapter for the FDBS as shown in Figure 5.28 and it accesses the foreign tables in the FDBS, but the files on

the file server and the tables in the CRM DBS are not any longer visible for the VT. Figure 5.29 shows one of the corresponding object information chapters. It specifies information about the foreign table in the FDBS that represents the *EMPLOYEE* table in the CRM DBS. However, the part of the foreign tables in the FDBS that is concerned with information about the CRM DBS, especially the *options* (see Figure 5.14), are not relevant for the FDBS VT adapter since it directly accesses the FDBS and its SQL tables. The FDBS VT adapter therefore only considers the SQL-related part of the foreign table definitions which can be considered as normal table definitions as exemplified in Figure 5.30. Actually, this is the information that is represented in the FDBS object information chapter in Figure 5.29.

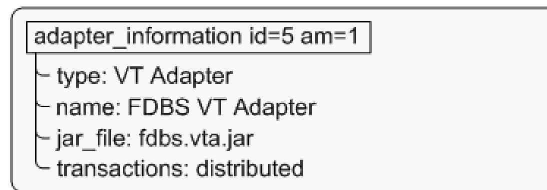


Figure 5.27: FDBS Adapter Information Chapter.

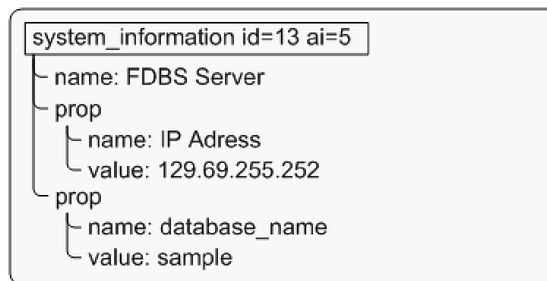


Figure 5.28: FDBS System Information Chapter.

The object definition chapter in Figure 5.31 is mapped from the object information chapter in Figure 5.29 and is almost the same as the object definition chapter that is mapped from the CRM DBS SQL wrapper in the VT (see Figure 5.17). The only difference is that the latter VT object definition is based on the foreign table defined by the CRM DBS SQL wrapper in the VT, whereas the former VT object definition is based on the table provided by the FDBS. This is reflected in a different namespace and description in both object definition chapters.

We also need a different FDBS deployment transformation wizard than the one associated with the SQL wrapper manager in the previous section. The new FDBS deployment transformation wizard is associated with the VT adapter manager. It extracts only information about the FDBS tables and transforms them into corresponding object information chapters. It does not care about whether a table is a regular table in the FDBS or whether a table is a foreign table that represents data and operations of an integrated remote system, e.g the file server or the CRM DBS. It just accesses tables according to their columns and constraints.

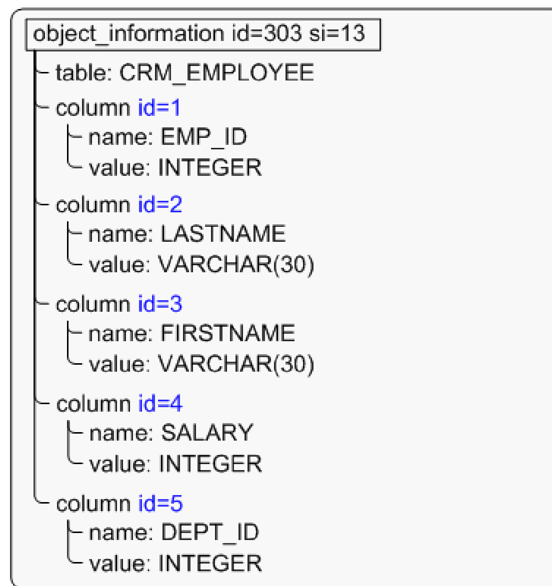


Figure 5.29: FDBS Object Information Chapter.

```
CREATE TABLE CRM_EMPLOYEE (
  EMP_ID INTEGER,
  LASTNAME VARCHAR(30),
  FIRSTNAME VARCHAR(30),
  SALARY INTEGER,
  DEPT_ID INTEGER
)
```

Figure 5.30: FDBS Table Representation.

The system information chapter is created on the basis of the information about the FDBS that the VT administrator entered into the wizard. The adapter information chapter does not change for this FDBS type at all. However, if we deploy the SQL wrappers directly into the VT, we need different system information chapters as well as different adapter information chapters. Another benefit of integrating the FDBS into the VT is that the heterogeneity of the remote systems is hidden by the FDBS. The VT only needs the information about the tables in the FDBS, e.g. table name, column names and column types (object information chapter) and information about the FDBS itself (system information chapter). The VT administrator does not need to consider details about the remote systems and their data and operations, e.g. files in the file server or tables in the CRM DBS, any longer since they are completely handled by the FDBS and the FDBS administrator.

Other middleware systems are analogously integrated into the VT. For example, the message broker of the integration example in Figure 2.3 is integrated by means of an MB VT adapter so that the Java EE application server can finally also access the CRM application and other remote systems integrated by the message broker via the VT (see Figure 5.32). The SQL wrappers and the MB adapters remain in their respective middleware system. The middleware infrastructure, i.e. the FDBS, the message broker and the deployed adapters, is reused as far as possible.

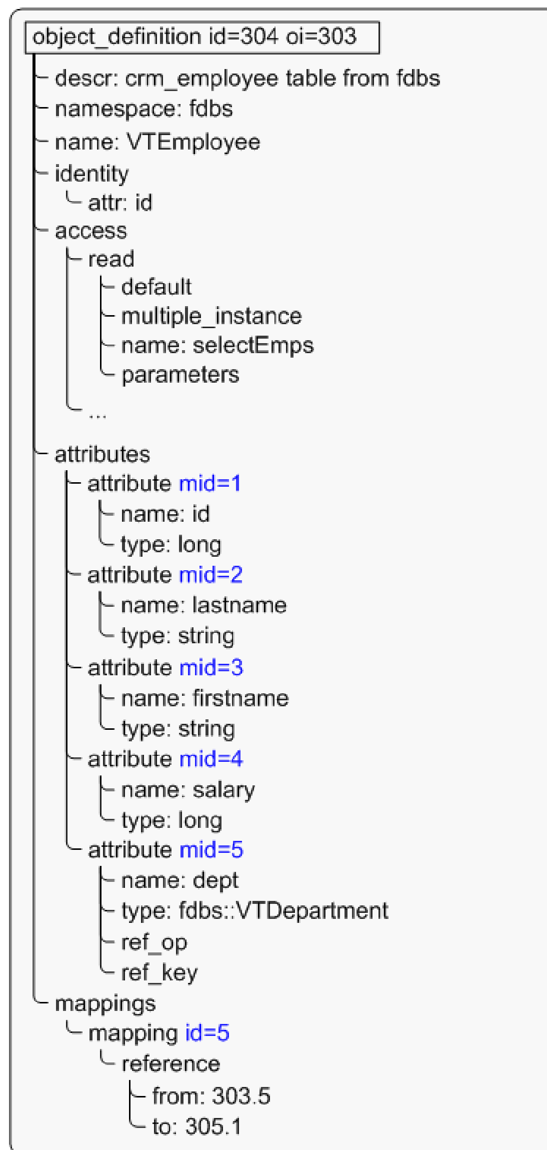


Figure 5.31: FDBS Object Definition Chapter.

Now we can reuse adapter deployments in the VT and we can even reuse whole middleware systems, i.e. available middleware infrastructure concerning adapter execution, so that we have only one adapter execution infrastructure and no longer redundant infrastructure parts. An adapter is only deployed into the corresponding middleware system, but not into the VT. The VT only holds the deployment information that is necessary for accessing the middleware system and its data and operations via a suitable VT adapter. The main benefit is that we can employ VT adapters for other middleware systems so that we can finally homogenize the existing middleware infrastructure.

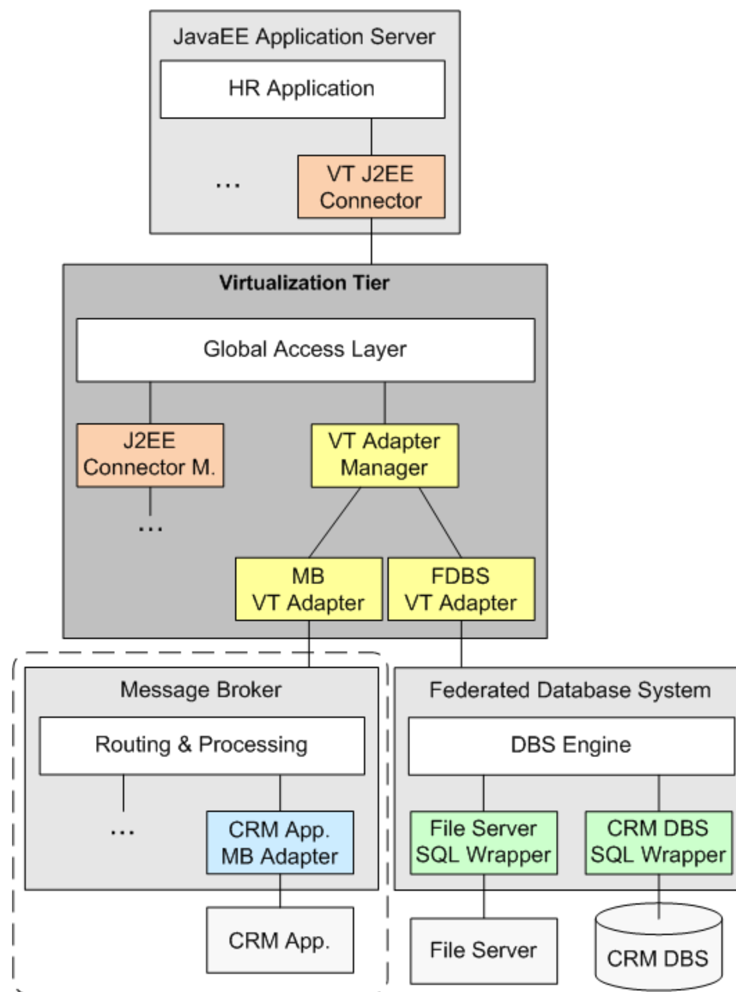


Figure 5.32: Reusing a Message Broker Infrastructure.

5.6 Summary

A *VT object configuration* defines how remote data and remote operations are represented as VT objects and how VT objects and the correlated remote data and remote operations are accessed. A VT object configuration consists of four configuration chapters: *adapter information chapter*, *system information chapter*, *object information chapter*, and *object definition chapter*. Additionally, each adapter manager is initially configured by an *adapter manager information chapter*. The deployment process is divided into two steps that are performed by different persons: adapters are deployed by an *adapter deployer* and suitable VT objects are defined by a *VT object deployer*. In this way, the deployment process in the VT is significantly alleviated and becomes practically manageable: the adapter deployer is doing his job as usual, i.e. adapter deployment and definition of middleware-specific data and operations, and the VT object deployer is only concerned with the VT-specific part, i.e. VT object definitions.

The co-existence of middleware infrastructures and VT bears optimization potential in terms of eliminating redundancies by applying different levels of reuse: reuse of adapter deployments by means of deployment transformation wizards and reuse of middleware infrastructures so that adapters are executed only in one environment, respectively.

Chapter 6

Applicability

In this chapter, we discuss and evaluate architecture patterns that can be applied to different kinds of integration scenarios and we show how Web service infrastructures can considerably benefit from IM technology, i.e. by means of the VT.

6.1 Architecture Patterns

There are several architecture patterns using IM technology to resolve heterogeneities in middleware infrastructures, i.e. adapters and middleware systems. We will discuss two conventional architecture patterns that solely employ adapters and middleware systems and three architecture patterns that are based on the VT and that we already introduced in Section 5.5. Figure 6.1 gives an overview of the architecture patterns. The conventional architecture patterns have a high degree of adapters to be potentially developed whereas the VT-based architecture patterns require only a low degree of adapters, which is the main benefit. The number of layers in the overall processing stack of the VT-based architecture patterns is slightly higher, but introduces only negligible performance overhead as shown in the performance evaluation in Chapter 7.

	Architecture Pattern	# Layers	# New Adapters
Convent.	Implementation Pattern	3	(m-1) * n
	Connection Pattern	4	(m-1) * m
VT-Based	Adapter Reuse Pattern	4	m
	Deployment Reuse Pattern	4	m
	Middleware Reuse Pattern	5	m+m

for m middleware systems and n remote systems

Figure 6.1: Architecture Pattern Overview.

6.1.1 Implementation Pattern

The most obvious solution of our initial integration scenario in Figure 1.1 is shown in Figure 6.2. We just develop a file server J2EE connector and a CRM DBS J2EE connector to integrate the file server and the CRM DBS into the Java EE application server. The disadvantages of this approach are quite obvious and show that this architecture pattern does not obey a systematic integration approach. Developing new adapters is a complex, lengthy and error-prone task and thus expensive. Moreover, from a global viewpoint there are a lot of adapter technologies and much more remote systems and thus the number of possible combinations of adapter technologies and remote systems is quite high. The overall number of adapters that must be potentially provided in an IT infrastructure using the *implementation pattern* is $m * n$ for m middleware systems and n remote systems, where usually $n \gg m$. The more remote systems have to be integrated and the more suitable adapters are missing, the less efficient and applicable the *implementation pattern* becomes. This becomes even worse, if we have existing adapters that already perform the desired integration tasks, just for a different integration technology. These adapters already are in productive use and work properly. In that case, reuse becomes very appealing. Another point is that changes in integrated remote systems can also affect the adapters that integrate the remote systems. For example, a remote system is upgraded to a new version or the APIs or other characteristics of the remote system are modified. This of course requires to modify the adapters too. If we apply the *implementation pattern* and develop a second or even more adapters for the same remote system, but for different adapter technologies, we have to modify all adapters if the remote system is modified. If we apply one of the other architecture patterns so that we rely on only one adapter, e.g. the one we also reuse in the VT, we have to modify only this adapter. Last, but not least, we need highly skilled software developers for developing new adapters and the more middleware systems and adapter technologies we want to support and the more new adapters we want to develop, the more highly skilled personnel we need, which becomes quite expensive. Moreover, highly skilled personnel is not easy to find and hire.

The disadvantages so far are very strong. Nevertheless, there also are some advantages. If we use only one middleware system in an IT infrastructure, then there clearly is only one middleware system incorporated in integration tasks within this IT infrastructure. Consequently, we have only a small middleware infrastructure footprint, at least when compared to the other architecture patterns. We can stick to one specific middleware system and adapter technology and we don't need to cope with other middleware systems and adapter technologies. This kind of specialization requires less personnel, software, hardware, administration and maintenance. The three layer architecture, i.e. client system, middleware system, remote system, of this monolithic environment also potentially provides for a better performance compared to the environments of the other architecture patterns, which require a four or even five layer architecture. Moreover, a monolithic environment also contributes to a more stable and reliable environment, which is more difficult to achieve with the

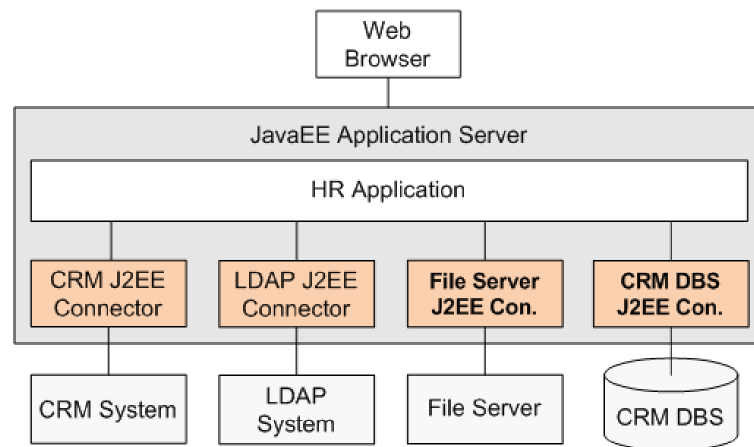


Figure 6.2: Developing new J2EE Connectors.

other architecture patterns, which comprise more layers and more software systems and components. However, the assumption of such a monolithic IT infrastructure is not very realistic, especially not for larger IT infrastructures and therefore the *implementation pattern* will be the best choice only for few integration scenarios.

In general, the *implementation pattern* can be a good choice if integration issues only rarely occur so that the initial overhead of the VT-based architecture patterns weighs more heavily than the disadvantages of the *implementation pattern*. Additionally, there may be organizational or administrative reasons why the *implementation pattern* is applied. For example, strictly separated departmental competences, maybe due to different security policies or due to different legal requirements, may lead to disjunct IT infrastructures that solve integration issues independently of each other. All these reasons may not occur very often and seem not to be very likely, but there is another reason for applying the *implementation pattern*, which is the dominating reason: the fact that the *implementation pattern* at first sight seems to be the easiest and directest way of handling integration issues so far, i.e. "just integrate this system". Actually, the VT-based architecture patterns are more complicated from an architecture viewpoint, but their benefits definitely outweigh this aspect and the disadvantages of the *implementation pattern* usually have a very undesirable and long-lasting impact on the overall IT infrastructure. Unfortunately, IM technology is unknown so far so that the *implementation pattern* has been the common integration solution and still is. This shows that IM technology has a very high potential to significantly alleviate integration issues in today's IT infrastructures.

6.1.2 Connection Pattern

A possible solution that allows the Java EE application server to use the file server SQL wrapper and the CRM DBS SQL wrapper is to interconnect the application server with the FDBS (see Figure 6.3). In this case, we avoid the development of

the file server J2EE connector and the CRM DBS J2EE connector, but we have to develop the FDBS J2EE connector that realizes the bridge between the Java EE application server and the FDBS.

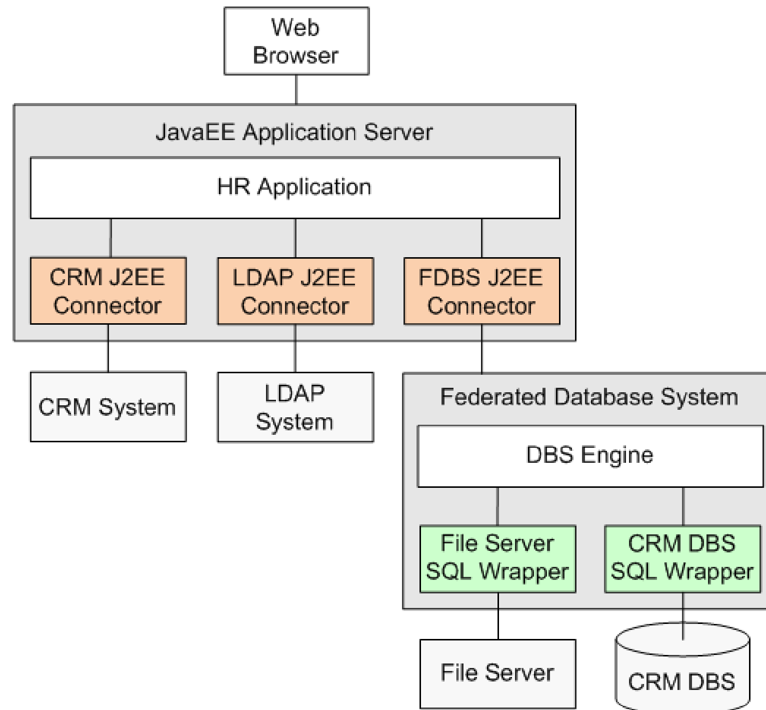


Figure 6.3: FDBS Integration.

The disadvantage of the *connection pattern* is that it is not systematic. If the Java EE application server wants to access another CRM system, e.g. the CRM application (see Figure 2.3), and only the CRM application MB adapter exists, we have to implement an MB J2EE connector so that the application server can access the message broker, which in turn uses the CRM application MB adapter (see Figure 6.4). This means that the pattern places an $m * m$ complexity on interconnecting each middleware system with each other one for m middleware systems since each middleware system requires an adapter to connect to another middleware system, e.g. the FDBS J2EE connector and the MB J2EE connector in Figure 6.4 (dashed lines) or the Java EE application server SQL wrapper for interconnecting the FDBS with the Java EE application server (see Figure 6.5, dashed line). This means that we still need highly skilled software developers for developing and maintaining the different middleware adapters. An additional problem is that the *connection pattern* adds a fourth layer to the integration architecture since we have to deal with two middleware systems instead of only one in the architecture of the *implementation pattern*. This places higher requirements on personnel, administration and maintenance and potentially reduces performance and the stability and reliability of the whole environment.

The advantage is that the Java EE application server can use any SQL wrapper

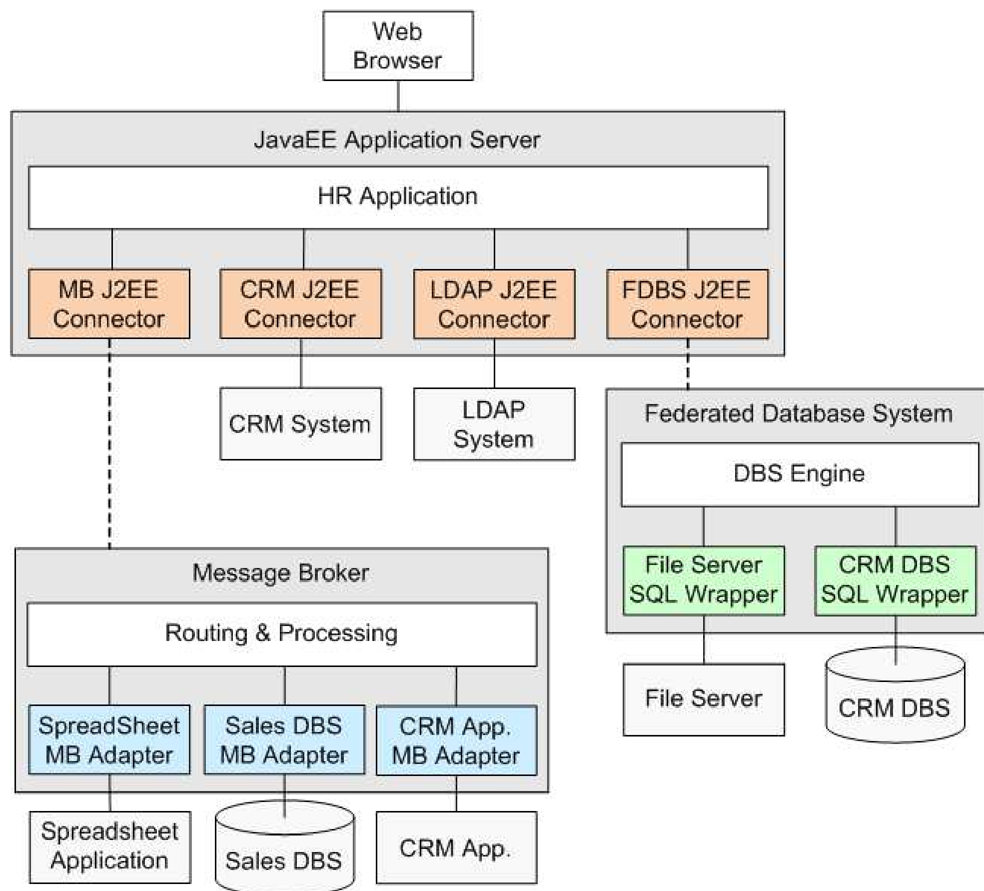


Figure 6.4: Message Broker Integration.

in the FDBS via the FDBS J2EE connector. We do not have to develop a new J2EE connector for each existing SQL wrapper, but we can use the existing adapters that already are in productive use and that work properly. Another advantage is that changes in integrated remote systems affect only one adapter since we do not develop adapters for other middleware systems to integrate the same remote system as it is done with the *implementation pattern*.

The *connection pattern* can be a good choice if frequent interconnections between two dedicated middleware systems are needed. For example, if only the FDBS and the Java EE application server have to be interconnected so that the FDBS can use some J2EE connectors, the Java EE application server could be directly interconnected with the FDBS by means of a Java EE application server SQL wrapper. However, if other middleware systems and additional interconnections are needed, the VT-based architecture patterns offer significant benefits.

6.1.3 Adapter Reuse Pattern

Systematically reusing adapters and even whole middleware systems requires more abstract considerations, which lead us to IM technology. In our example scenario,

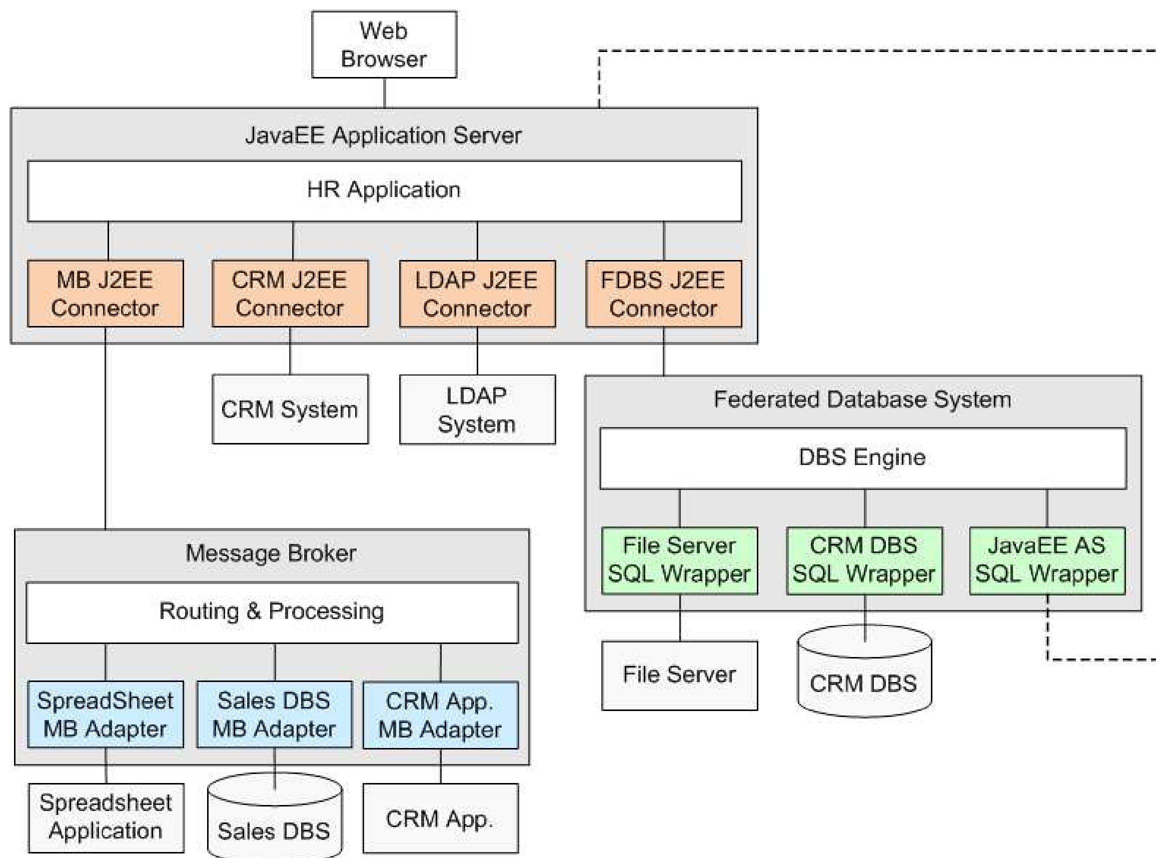


Figure 6.5: Java EE Application Server Integration.

we reuse the file server SQL wrapper and the CRM DBS SQL wrapper by deploying them into the VT (see Figure 6.6). The Java EE application server then uses the VT and the deployed SQL wrappers to properly access the file server and the CRM DBS. The Java EE application server uses the VT J2EE connector to access the VT, which is similar to the *connection pattern* where the application server needed the FDBS J2EE connector to access the FDBS server. The benefit of the *adapter reuse pattern* is that the $m * m$ complexity of the *connection pattern* and the $m * n$ complexity of the *implementation pattern* is reduced to m, i.e. m VT middleware adapters, so that every middleware system can access the VT. Each middleware system now solely connects to the VT as some kind of multiplexer, but does not need to directly access remote systems or other middleware systems.

Each of the VT-based architecture patterns allows to reuse existing adapters that already are in productive use. This avoids the expensive and error-prone development of new adapters. The big advantage of the VT-based architecture patterns is that the VT allows to systematically reuse adapters which simply means that there are no restrictions regarding the reuse of adapters. Any adapter, any adapter technology and any number of adapters can be potentially reused in the VT and any middleware system or other software system can employ the VT to access remote

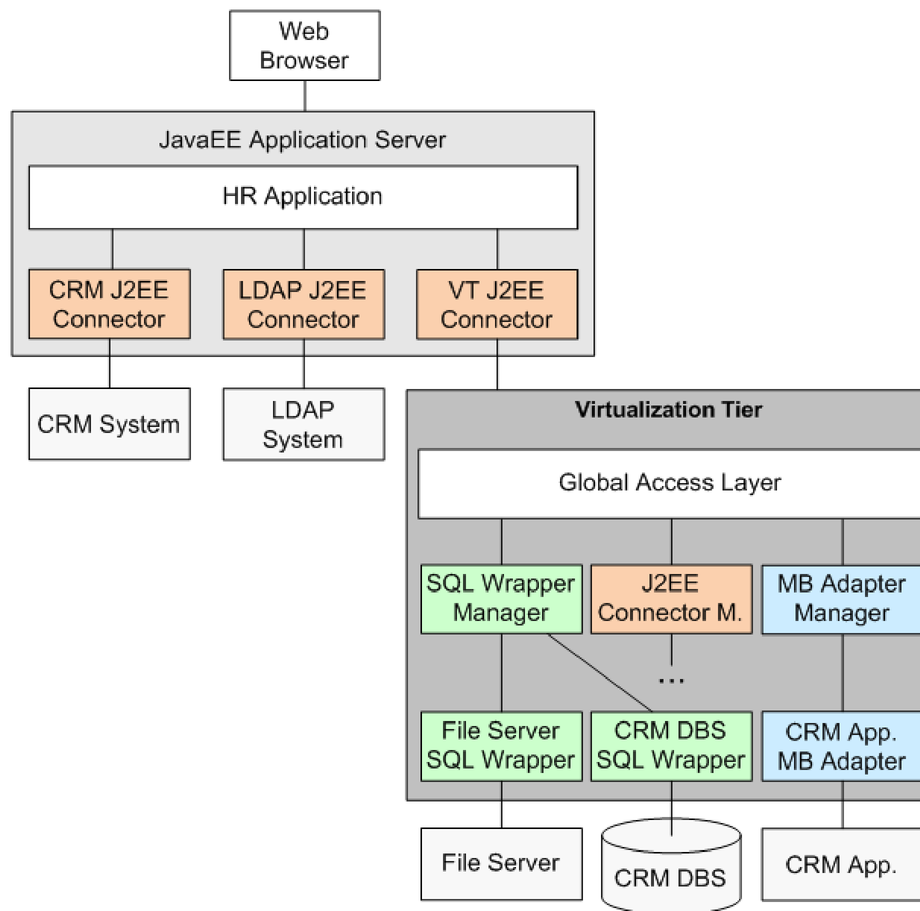


Figure 6.6: Reusing Adapters.

systems that have been integrated into the VT. We need only one adapter for a remote system so that any software system can access this remote system because every software system just needs to access the VT and thereby the adapter and the remote system. We do not any longer need software developers for developing new adapters in every integration scenario. We just reuse the existing adapters by deploying them into the VT.

The VT-based architecture patterns employ the VT as an IM system and therefore introduce a fourth layer to the integration architecture. The VT is a core system that represents a central component in an IT infrastructure. From that point, the VT can contribute to a stable and reliable environment and it can also simplify management and administration of adapters when they are deployed into the VT instead of deployments in different middleware systems. The only disadvantage of the fourth layer compared to the three architecture layers of the *implementation pattern* are performance issues. However, we addressed this issue in our experiments, which we discuss in Chapter 7 and which show that performance actually is not as critical as considered at first sight.

6.1.4 Deployment Reuse Pattern

So far, we employ the file server SQL wrapper and the CRM DBS SQL wrapper in the FDBS as well as in the VT. This means that both SQL wrappers and their deployments have to be maintained twice in different places, maybe even by different administration authorities. The *deployment reuse pattern* avoids this problem and additionally reuses adapter deployments as described in chapter 5. We use the FDBS deployment transformation wizard to automatically extract the deployment information of both SQL wrappers, transform them into suitable deployment information for the VT and finally automatically deploy both SQL wrappers into the VT (see Figure 6.7). The dashed boxes indicate the adapter deployments of the two SQL wrappers including information about the remote systems and the remote data and remote operations integrated by the SQL wrappers. Thus, only the adapter deployments in the respective middleware systems have to be maintained, e.g. the SQL wrapper deployments in the FDBS in this example. The difference between the *adapter reuse pattern* and the *deployment reuse pattern* is that the *deployment reuse pattern* is based on deployment transformation wizards, which must be additionally provided. However, the advantage is that a VT administrator can automatically and systematically deploy adapters into the VT on the basis of adapter deployments in existing middleware systems. Additionally, if these adapter deployments are changed, they can be again automatically transferred to corresponding adapter deployments in the VT by means of the deployment transformation wizards. Note that adapter deployments do not only contain proper configuration of the adapters itself, but also comprise information about the integrated remote system and the integrated data and operations, which can result in a considerable amount of specification information. For example, a remote system may export a number of table definitions to an FDBS consisting of a number of column definitions for each table. Another remote system may export a number of operations to a Java EE server consisting of a number of parameters and object definitions. Therefore, the *adapter reuse pattern* is most suitable for an IT infrastructure with a fixed set or with an almost fixed set of existing adapters and adapter deployments. However, this is not a very realistic assumption since IT infrastructures typically change over time. Therefore, deployment transformation wizards provide an excellent means for managing adapter deployments, which otherwise would have to be manually and unsystematically handled by different middleware administrators. Of course, if adapters are only deployed into the VT, the *adapter reuse pattern* is the best choice since there are no existing adapter deployments that we could reuse.

6.1.5 Middleware Reuse Pattern

We also discussed in the last chapter that reuse of adapters in the VT requires suitable middleware functionality for handling and executing adapters. For example, the two SQL wrappers in the VT in Figure 6.7 are executed by the SQL wrapper manager. This middleware functionality is the same as the functionality for execut-

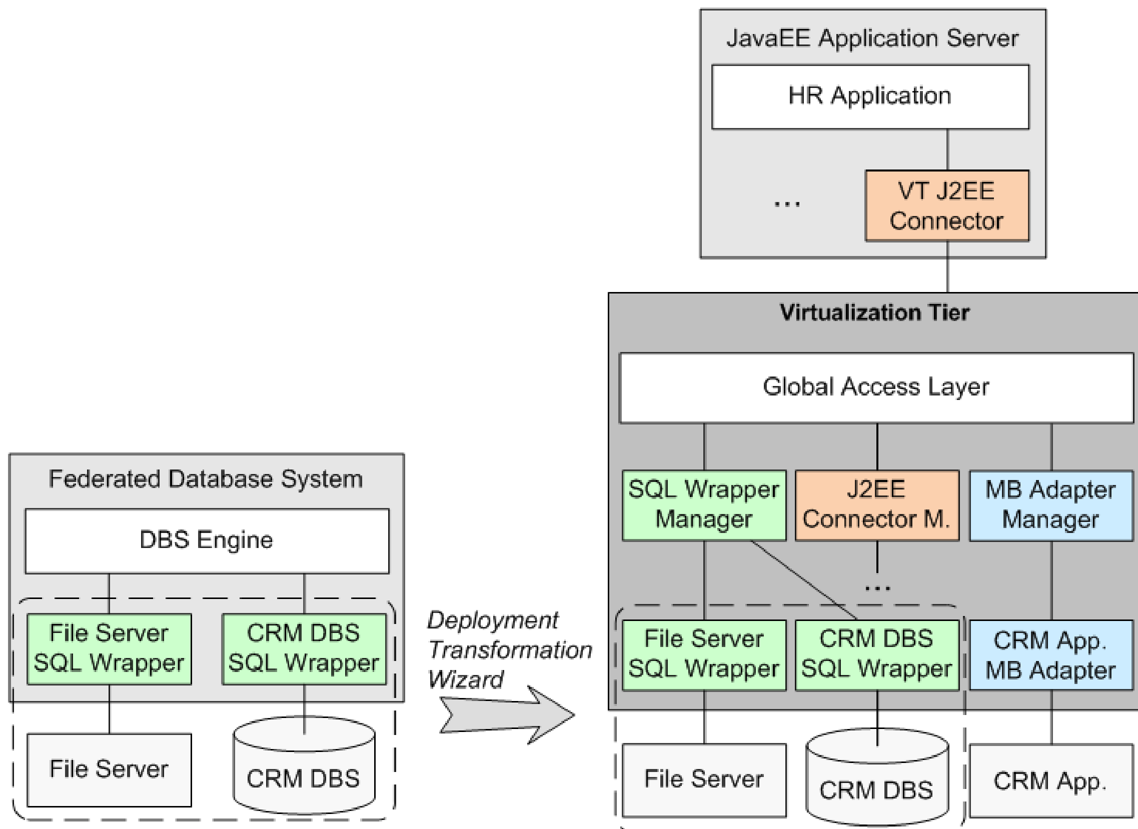


Figure 6.7: Reusing Adapter Deployments.

ing SQL wrappers in the FDBS. Therefore, we apply the *middleware reuse pattern* and reuse the FDBS instead of employing an SQL wrapper manager. All we need is the VT adapter manager that uses the FDBS VT adapter to access the FDBS (see Figure 6.8). In that way, we no longer deploy the SQL wrappers into the VT, but we indirectly reuse the SQL wrappers by means of the FDBS. This is similar to the *connection pattern*. However, the *connection pattern* is unsystematic whereas the *middleware reuse pattern* employs the VT in a systematic reuse architecture and with all the advantages a VT-based architecture pattern comes with. Additionally, we reduce the complexity of $m * m$ adapters of the *connection pattern* to $m + m$ for the *middleware reuse pattern*: m VT middleware adapters, i.e. each middleware system can access the VT, and m middleware VT adapters in the VT, i.e. the VT can access each middleware system. For example, we no longer need to deploy SQL wrappers into the VT if they are already deployed in the FDBS: the VT indirectly reuses any SQL wrapper that is deployed in the FDBS. Another advantage is that we do not have to buy, develop, employ and maintain parts of our middleware infrastructure twice.

The disadvantage of the *middleware reuse pattern* is the same as for the *connection pattern*, i.e. we introduce an additional layer to the integration architecture. Therefore, the *middleware reuse pattern* should be applied if an IT infrastructure fre-

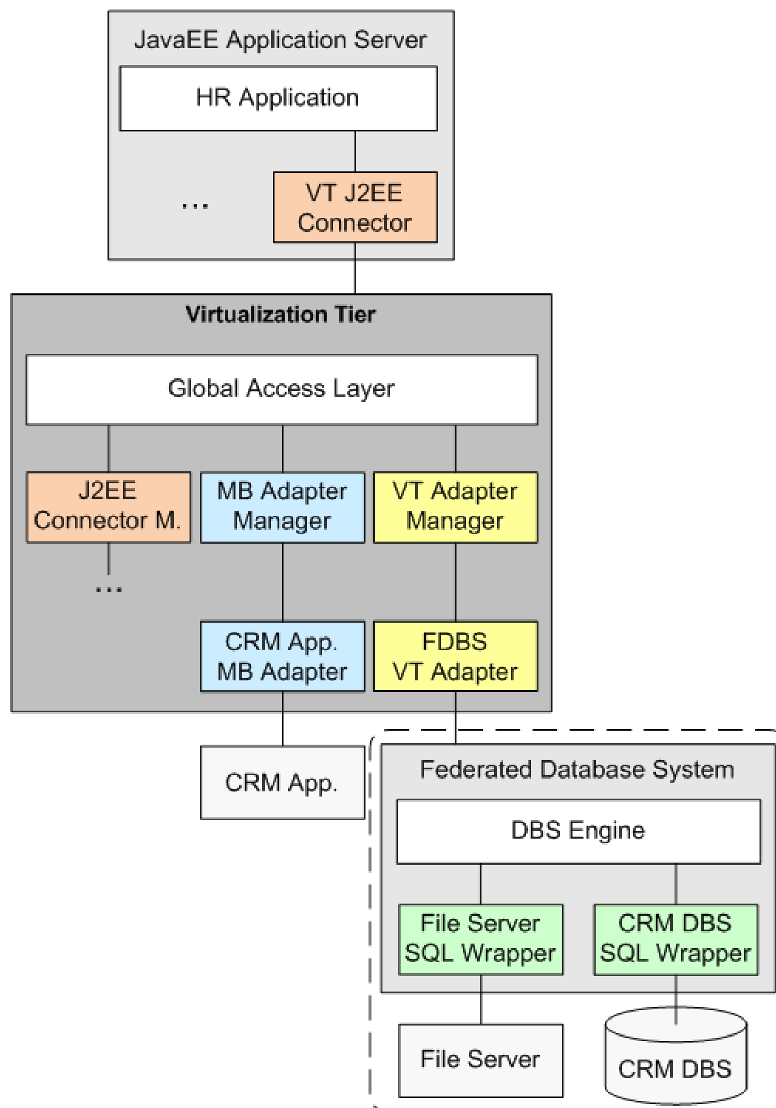


Figure 6.8: Reusing Middleware Infrastructure.

quently needs specific interconnections between two dedicated middleware systems, i.e. situations as the one sketched for the *connection pattern*.

Another reason for applying the *middleware reuse pattern* are heterogeneities within an adapter technology. For example, the J2EE connector architecture defines contracts between Java EE application server and J2EE connector, but the client interface is open to other operations and data structures and different Java EE application server products differently handle J2EE connector deployments, especially via vendor-specific extensions. This leads to insidious interoperabilities between J2EE connectors of different vendors so that the standard-conform J2EE connector manager might not be able to correctly handle all J2EE connectors. In such a case, the *middleware reuse pattern* can be a good alternative to reuse standard-degenerated adapters indirectly via the respective middleware system and not directly in the VT.

6.2 Web Services

Web services are becoming more and more ubiquitous and more and more middleware systems offer Web service access. Therefore, the question is how Web services are related to IM technology and whether the Web service approach can make IM technology superfluous?

6.2.1 Web Services & IM Technology

First of all, Web services are an integration technology, but not an IM technology. Moreover, Web services are the currently predominant integration technology, but they still are yet another integration technology with the same heterogeneity problems as any other integration technology. The most important issue is to cope with the heterogeneities of different remote systems. A Web service architecture typically relies on an ESB and corresponding adapters where we again have to develop new adapters or handle different adapter technologies when we want to integrate other remote systems via Web services. Another point is the supposition that the whole world is using Web services and only Web services. Web services undoubtedly are a considerable improvement over existing integration approaches, but they still do not make the world homogeneous. The last decade introduced, developed and employed Web services and there surely is substantial progress in making IT infrastructures more interoperable. But there still is much heterogeneity left, which clearly shows that Web services are not the final solution. Actually, it is even not realistic to strive for a homogeneous IT landscape because of the overall complexity and heterogeneity that comes from the real world and the real world requirements that finally determine IT infrastructures and the resulting heterogeneities. Put in other words, the world is complex and heterogeneous by virtue. We cannot homogenize it and therefore we cannot completely homogenize IT infrastructures. We simply have to deal with heterogeneities, but we can deal with them in a way that makes it feasible to handle heterogeneities as easy as possible. IM technology is an excellent means to do so since it shields from integration issues and substantially alleviates access to heterogeneous software systems. An IM system employs integration technologies and abstracts from them so that integration issues are hidden. An IM system acts on a higher abstraction layer than an integration technology does, which simply means that an IM system such as the VT neither is in competition with Web services nor does it make sense to compare them with each other (also refer to Section 2.6 for a discussion of IM technology). For example, the VT can employ Web service technology as another integration technology. We employ a Web service manager in the VT that can deal with Web services and that makes them available for other systems and technologies (see ① in Figure 6.9). The other way round, the VT Web service facility allows to access the VT via Web services so that the Web service architecture can benefit from the integration independence that the VT provides (see ② in Figure 6.9). An important characteristic of the VT is that it does not restrict remote system access to Web services solely. Web services are only one means

of accessing them. The VT provides integration independence to any system that accesses the VT since VT access means to support different access paradigms and technologies, not only Web services. Figure 6.9 exemplifies that an FDBS can use a VT SQL wrapper and set-oriented, declarative SQL queries to access the VT and any remote system that is integrated into the VT. A Java EE application server can use a VT J2EE connector and operation-oriented requests to access the VT and so on. We can use Web services to access the VT, but we can also use other means. We stay heterogeneous if it is desired or necessary and we also provide a much easier way to deal with integration issues that deal with heterogeneous software systems.

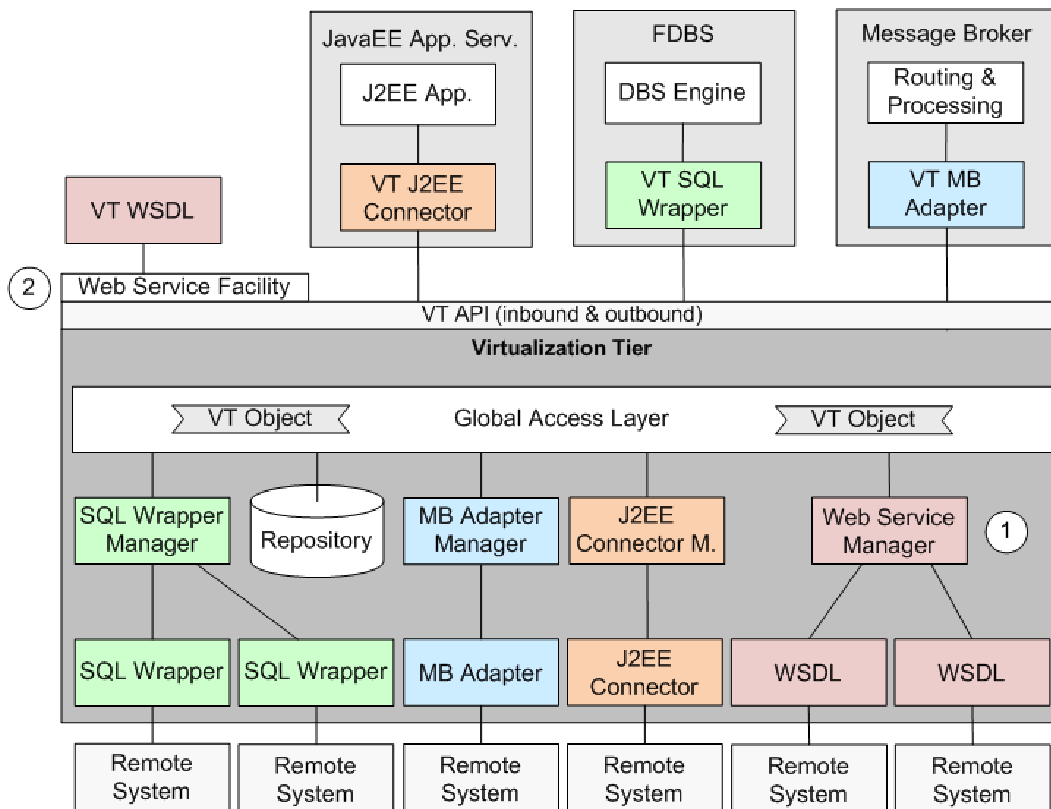


Figure 6.9: The VT, Web Services and more than Web Services.

6.2.2 Web Service Infrastructures

SOA-based applications consist of service compositions, e.g. Business Process Execution Language (BPEL) processes, that realize certain business logic [SvdAB+03]. These processes call Web services to access remote data or to execute remote operations. A WSDL specification determines how to bind to a service provider and how to drive the Web service [CMRW07]. This part of a Web service is visible to the process modeler. It is standardized and it provides uniform access to the different Web services. What is not seen by the process modeler are the Web service implementations itself, which are performing the actual work: accessing the desired data

and executing the desired operations in remote systems. The infrastructure part of Web services that is concerned with accessing remote data and remote operations lies underneath a BPEL process and underneath a Web service call. This Web service infrastructure part is a heterogeneous world that still suffers from systematic and viable solutions in terms of how to uniformly access heterogeneous remote systems. The way a remote system is called and accessed is not standardized, but completely depends on the characteristics of the accessed remote system. Web service access to remote systems is often done by means of an Enterprise Service Bus (ESB), i.e. the middleware system in the Web service infrastructure. An ESB tackles the heterogeneities in the very same way as other middleware systems and adapter technologies do. From this viewpoint, the Web service architecture represents yet another middleware solution and Web services are adapters that integrate remote systems into Web service-based applications. Previous discussions on Web service development and Web service infrastructures clarify that the coding part of Web service implementations is based on contemporary middleware systems or that Web service implementations at least require the implementation of integration components, i.e. adapters, in the same way as heterogeneous remote systems usually are integrated into middleware systems [PH07, LCL06, Lee05, Hai07, APvS03]. Next, we take a closer look at how the VT can improve a Web service infrastructure and how the VT can make it more flexible.

6.2.3 The VT Acting as an ESB

Figure 6.10 shows a simplified, graphical representation of an example BPEL process that realizes an HR application case. The core of the four activities are Web service calls that access data and operations in the remote systems of our integration scenario, i.e. the file server, the CRM DBS, the CRM application and the CRM system. The crucial point about this process is that it is not clear how access to the different remote systems is actually realized. This is of no concern to the process modeler at the BPEL and WSDL level, but it is the task of the infrastructure provider to offer Web service access to remote systems. The Web service architecture represents interfaces, but does not tell how to actually access heterogeneous remote systems, i.e. how to integrate them. Access to Web services is provided in a homogeneous manner whereas the underlying access to the remote systems still is heterogeneous and needs to be solved in an individual, remote system-specific manner, i.e. we have diverse Web service implementations. An ESB typically provides the necessary integration technology that allows to develop suitable Web service implementations, which therefore rely on adapters as other middleware systems do. In our example, the ESB would need four adapters to access the remote systems of the BPEL process as shown in Figure 6.11. However, we do not have these adapters, but we have to develop them from scratch. On the other hand, the given remote systems are already integrated into the other middleware systems of our integration scenario. From that viewpoint it would be very appealing to reuse the existing adapters instead of developing four new ESB adapters.

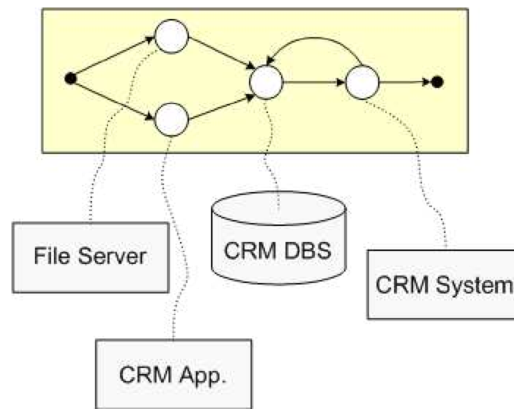


Figure 6.10: HR BPEL Process (abstract representation).

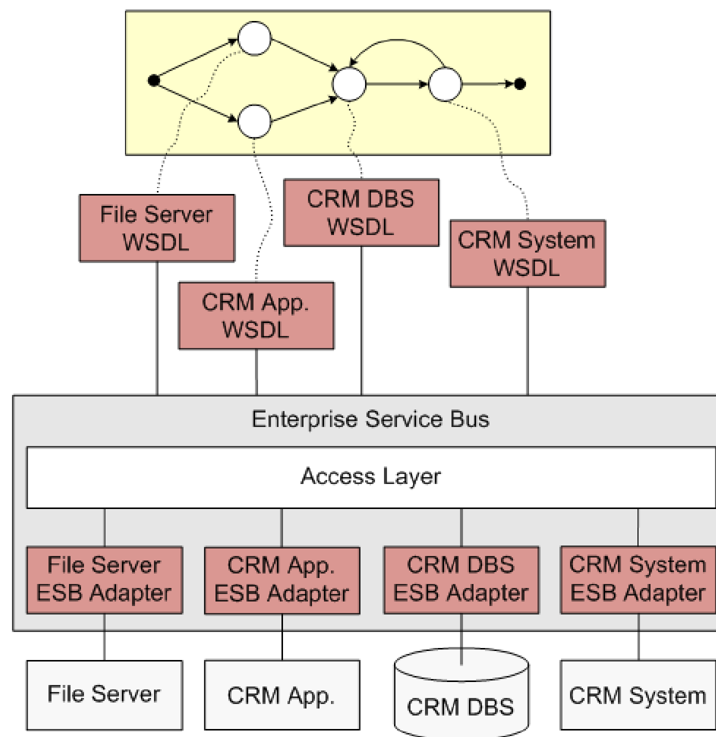


Figure 6.11: HR BPEL Process Solved with ESB.

If we assume that the existing IT infrastructure completely supports Web services, we can reuse the existing adapters via their respective middleware systems that have to be able to act as **local ESBs** as shown in Figure 6.12. The disadvantage is that we have to deal with different middleware systems and their corresponding Web service facilities, i.e. how to access the middleware system via appropriate Web services. The FDBS then provides SQL table-based Web services, the Java EE application server provides Java-based Web services, and so on. Consequently, the SOA-based application still has to deal with these remaining heterogeneities. The

other question is whether all of the middleware systems can actually act as local ESBs and provide suitable Web services. Maybe some of them even don't support Web service technology at all. Apart from that, we already discussed a more suitable and more beneficial approach, i.e. the use of IM technology. One facet of the VT is that it can act as a **global ESB** that is able to provide Web service access to any remote system that is integrated into the VT (see Figure 6.13). This means that we do not need any longer different local ESBs, i.e. middleware systems, that are restricted considering their employed integration technologies and remote systems, but we can rely on the integration independence of the VT, i.e. to uniformly reuse diverse adapters.

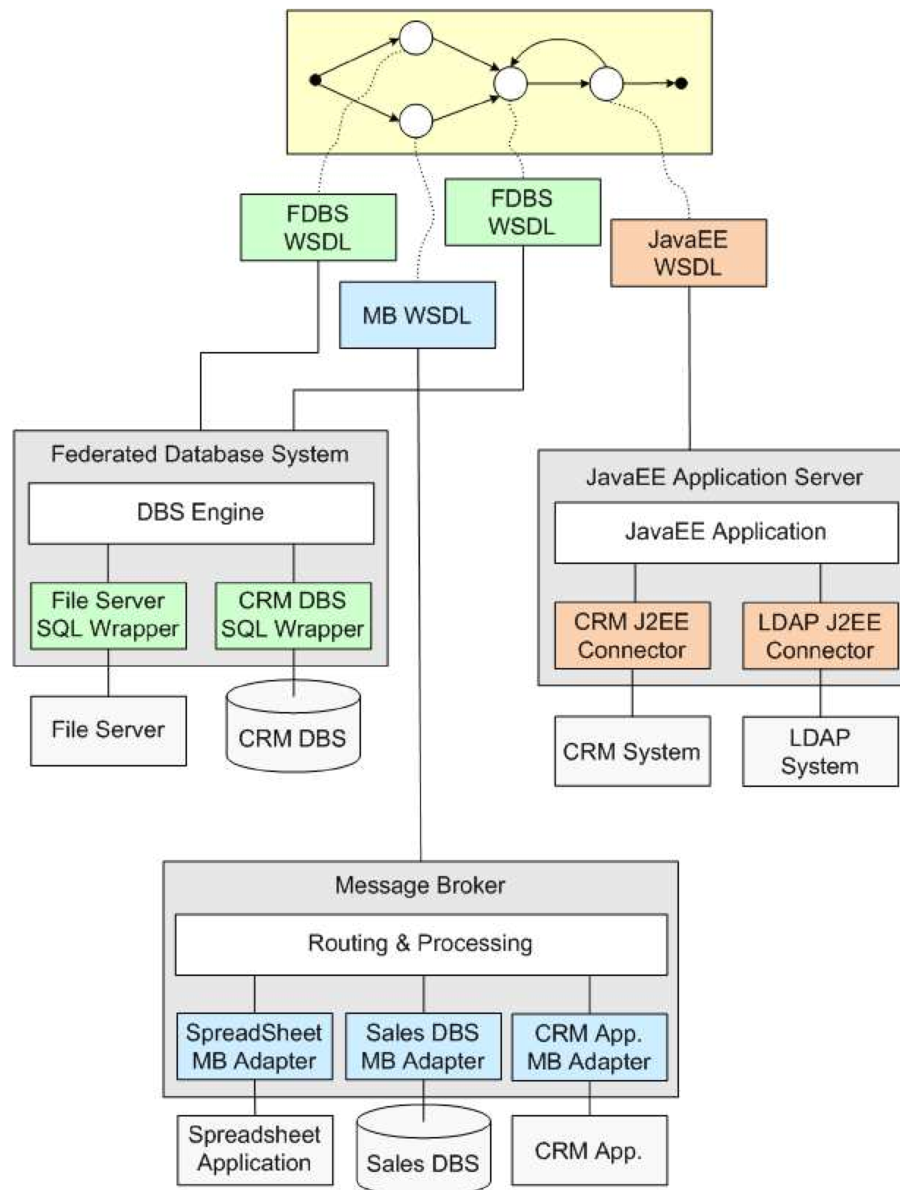


Figure 6.12: HR BPEL Process Solved with Middleware Systems.

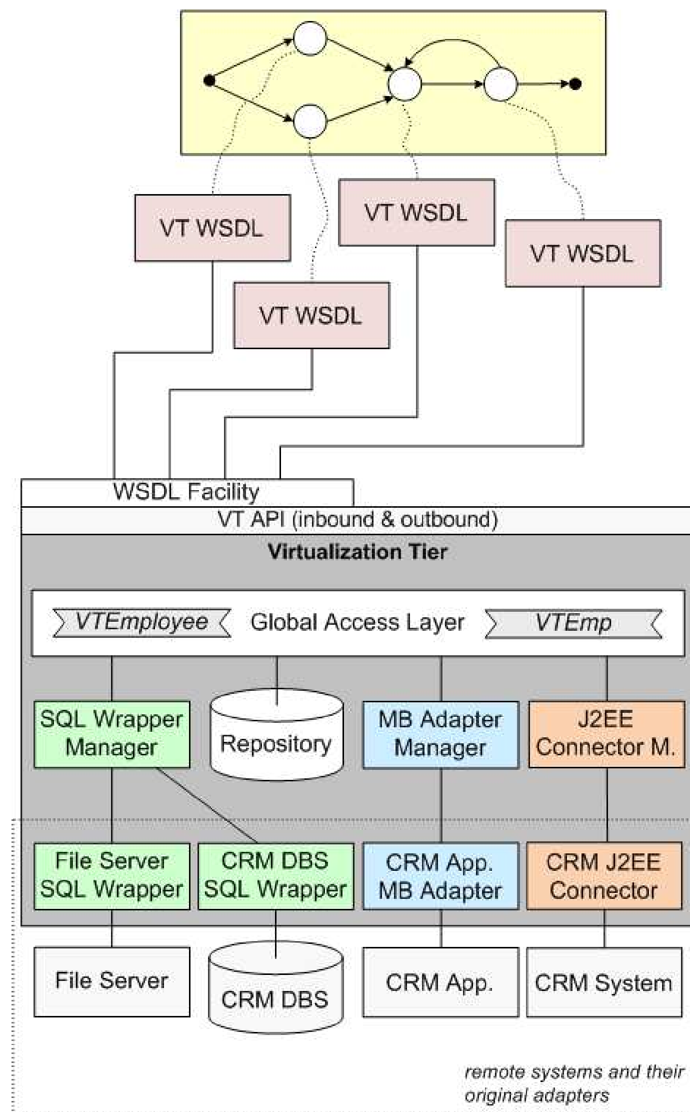


Figure 6.13: HR BPEL Process Solved with VT.

6.3 Summary

There are several architecture patterns using IM technology to resolve heterogeneities in middleware infrastructures, i.e. adapters and middleware systems. We discussed two conventional architecture patterns that solely employ adapters and middleware systems and three architecture patterns that are based on the VT. The most important point of the architecture patterns is the number of adapters that potentially have to be developed. Here, the VT-based architecture patterns clearly are superior to the conventional architecture patterns. None of the patterns is the best in every integration scenario, but the *adapter reuse pattern* and the *deployment reuse pattern* most probably suitably apply to a large part of integration scenarios especially if the VT is employed in an IM-oriented IT infrastructure.

The VT supports Web services as another integration technology. The VT completely hides integration issues from the upper part of the Web service infrastructure and can act as a *global ESB*. SOA-based applications can seamlessly access any remote system via the VT and we avoid the overhead of developing Web service implementations as ESB adapters. The result is that the heterogeneous world beneath the homogeneous Web service architecture becomes integration-independent and the VT thereby considerably alleviates the constitution of a uniform Web service infrastructure.

Chapter 7

Performance

So far, we discussed the benefits of the VT such as software reuse, integration independence, higher stability and flexibility of IT infrastructures, reduced costs for software, hardware and maintenance. These benefits do not come for free. We already mentioned that a potential disadvantage of the VT approach could be the additional VT layer in the overall processing stack. The question is whether this additional layer will significantly decrease performance or not. Therefore, we prototypically implemented the VT and systematically evaluated our IM approach in extensive experiments. The experiment results show that the VT decreases performance at an acceptable rate and that the VT surprisingly even improves performance in some settings. But even the performance decrease is negligible when we consider the benefits of the VT.

7.1 Experiment Environment

We selected representative middleware platforms, adapter technologies and remote systems for our experiments to show that the VT approach is applicable in a wide range of integration scenarios. We chose representatives for the most frequently occurring cases: data-oriented adapter technologies, i.e. SQL wrapper technology, and operation-oriented adapter technologies, i.e. J2EE connector technology (also refer to the discussion of representative adapter technologies in Section 2.2). We used IBM DB2 Information Integrator 9.1 (short: DB2) to execute the SQL wrappers and IBM WebSphere Application Server 6.1 (short: WebSphere or WS) to execute the J2EE connectors. Furthermore, we used typical kinds of remote systems for our experiments: Derby, a relational DBS that is accessed via SQL requests (case *Derby*), a legacy object-oriented application system implemented in Java, which offers an API to retrieve and manipulate personnel data (case *OOApp*), and comma-separated files that are read and written by means of simple file operations (case *File*). Figure 7.1 shows the resulting experiment scenario with the VT prototype.

The *Derby* case represents data-oriented, data-intensive processing in remote systems and it represents scalable application systems. The *OOApp* case represents operation-oriented processing and it represents non-scalable application systems.

The non-scalability simply comes from the object graph that set-oriented requests return since the object graph is locally handled and therefore it has to be passed in one chunk. Hence, the larger the result set, i.e. the object graph, the larger are the required system resources, especially main memory, which clearly does not scale for very large object graphs. The *File* case represents small systems or dependent system components such as local access to files, e.g. plain text or XML, input from sensors, e.g. temperature or humidity, or access to controls, e.g. status of a door or window (open, closed) or controls in a factory. The *Derby* case and the *File* case

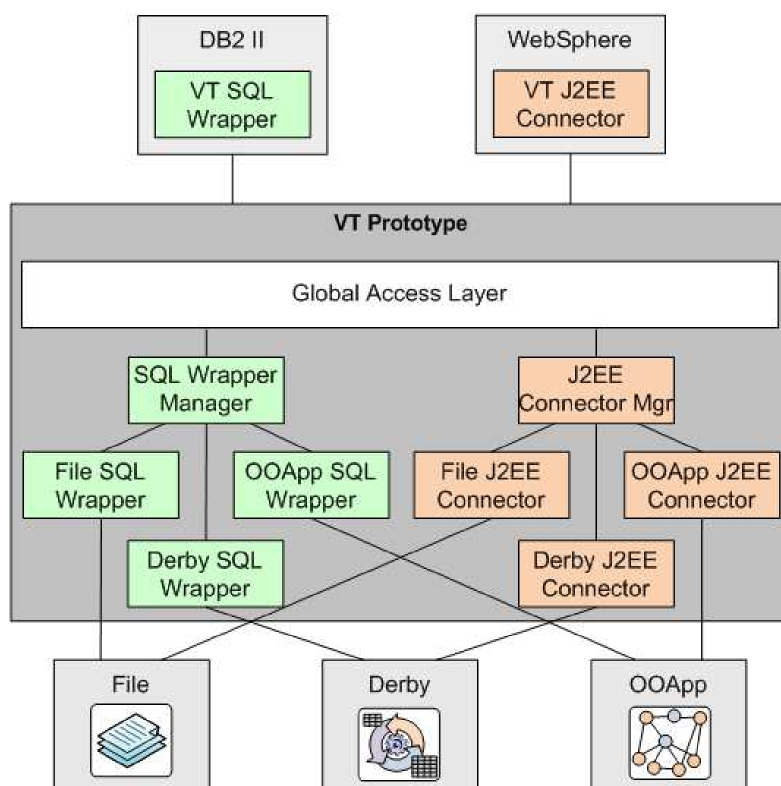


Figure 7.1: VT Prototype Experiment Scenario.

rely on data of the TPC-H benchmark that is intended for benchmarking decision support systems in complex application scenarios and with large data volumes. We use the *lineitem* table and the *orders* table for our experiments (see Figure 7.2). The *OOApp* case relies on a different data set since it requires to span a densely linked object graph. We therefore created a typical employee-department dependency (see Figure 7.3) where an employee is associated with a department and a manager and where a department is associated with a manager. A manager also is an employee so that we even have a recursive relationship.

Figure 7.4 shows the overall VT prototype experiment architecture and the different layers of the integration architecture that are executed on separate hosts, respectively. The *File* case is a special case since the file adapters directly access the files in the adapter operating system process and not as a stand-alone remote sys-

```
CREATE TABLE Orders (
  o_orderkey            Integer NOT NULL,
  o_custkey             Integer,
  o_orderstatus         Char(1),
  o_totalprice          Decimal(10, 2),
  o_orderdate           Date,
  o_orderpriority       Char(15),
  o_clerk               Char(15),
  o_shippriority        Integer,
  o_comment             Varchar(79)
);

CREATE TABLE Lineitem (
  l_orderkey            Integer NOT NULL,
  l_partkey             Integer,
  l_suppkey             Integer,
  l_linenumber          Integer NOT NULL,
  l_quantity            Decimal(10, 2),
  l_extendedprice       Decimal(10, 2),
  l_discount            Decimal(10, 2),
  l_tax                 Decimal(10, 2),
  l_returnflag          Char(1),
  l_linestatus          Char(1),
  l_shipdate            Date,
  l_commitdate          Date,
  l_receiptdate         Date,
  l_shipinstruct        Char(25),
  l_shipmode            Char(10),
  l_comment             Varchar(44)
);
```

Figure 7.2: TPC-H Tables.

```
public class Department
{
   public int deptId;
   public String name;
   public Employee manager;
   public int noOfCourses;
   public int budget;
}


public class Employee
{
   public int empId;
   public String lastname;
   public String firstname;
   public java.util.Date hdate;
   public int salary;
   public Department dept;
   public Employee manager;
}
```

Figure 7.3: OOApp Classes.

tem. The architectures of the native adapter execution environments comprise three layers according to the *implementation pattern* of Section 6.1.1, i.e. the adapters are directly executed in their native middleware systems: the file SQL wrapper, the Derby SQL wrapper, the OOApp SQL wrapper are executed in DB2 and the file J2EE connector, the Derby J2EE connector and the OOApp J2EE connector are executed in WebSphere. The VT-based architectures comprise four layers accord-

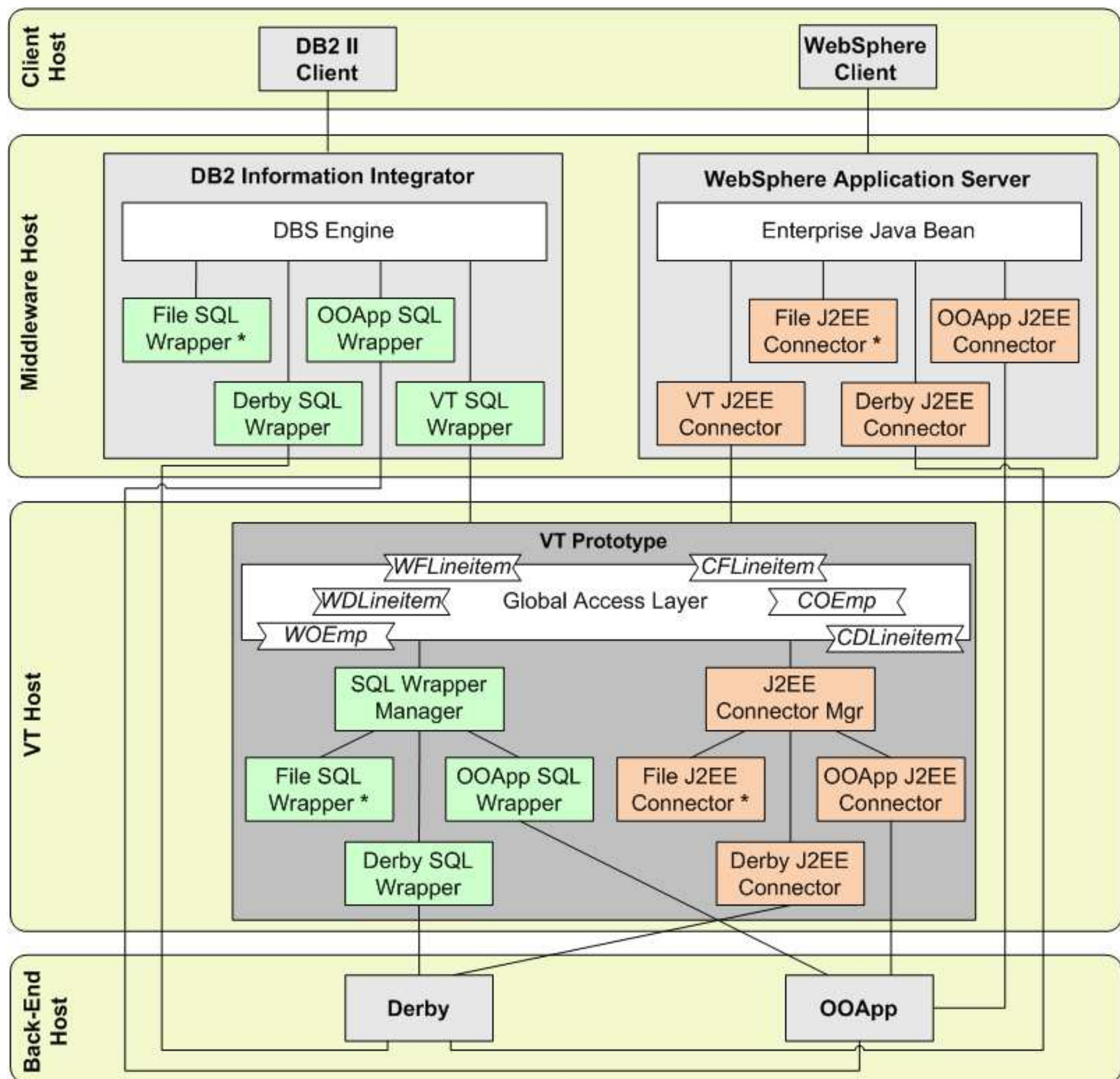
ing to the *adapter reuse pattern* of Section 6.1.3. These architectures additionally include the VT: the SQL wrappers and the J2EE connectors are executed in the VT, and DB2 and WebSphere access the VT and thereby the other adapters only indirectly. The client applications, the middleware systems and the remote systems run on separate hosts. The VT also runs on a separate host as a generally available IM system. The client host, the middleware host and the VT host have the same hardware and software configuration, i.e. a dual core processor with 1.53 GHz, 1 GB RAM and Windows XP Professional SP 3. The DB2 experiments employ a DB2 client on the client host, DB2 on the middleware host and the VT prototype on the VT host. The WebSphere experiments correspondingly employ a WebSphere client on the client host, the WebSphere application server on the middleware host and the VT prototype on the VT host. The experiments use different remote systems according to the *Derby*, *OOApp* and *File* cases, and they access different remote system instances according to the different scale levels. Therefore, we run the Derby server instance and five OOApp server instances (each one with a different scale level, i.e. a different object graph base) on a bigger machine. This gave us a greater flexibility for managing the single server instances and for handling single experiments in sequence. The remote system host has four dual core processors with 1 GHz, 32 GB main memory and CentOS 4. Important is that the VT and the middleware systems run on hosts with identical hardware and software configuration since we compare the execution of DB2 or WebSphere plus the VT versus the execution of DB2 or WebSphere only, and we compare file access on the VT versus file access on DB2 and on WebSphere for the *File* case.

7.2 Experiment Execution

In this section, we discuss the experiment categories and the different experiment architectures that we use for our prototype evaluation. We evaluate the VT approach according to the execution time that the VT additionally introduces to the overall execution time of the experiment requests. We measure the additionally introduced execution time relative to the overall execution time in percent and call it **VT overhead** (or *overhead* only if it is clear from the context that *VT overhead* is meant).

7.2.1 Experiment Categories

The experiments comprise four categories, which represent basic access types: executing operations (category *Exec*), reading data (category *Read*), writing data (category *Write*) and executing set-oriented queries (category *Query*) (also see Figure 7.5).



* *File* is not a backend system in the sense that it runs on a separate host, but the file adapters access local data files that reside on the same host as DB2 II, WebSphere or the VT.

Figure 7.4: Overall VT Prototype Experiment Architecture.

Exec Category

The *Exec* category refers to the execution of remote operations which do not incorporate data transfer, neither reading nor writing. They just use execution parameters and they are intended for side-effects or computational issues, e.g. a time service operation that delivers the current time or a reservation operation that allows a person to book cinema tickets. The representative remote operation of the *Exec* category is a simple wait operation in a remote system with a parameter for the number of ms that an operation execution lasts. The wait operation thereby simulates an operation in a remote system that performs some work. The *Derby* case uses a stored procedure that waits for x ms and then returns, the *OOApp* case offers

an RMI method that waits for x ms before it returns and the *File* case is directly executed in the file adapter, either in the middleware or in the VT, just by calling a wait routine to wait for x ms. The simulated remote operation execution duration is *1*, *10*, *100*, *1,000* and *10,000* ms to show how the VT overhead decreases with increasing operation execution duration.

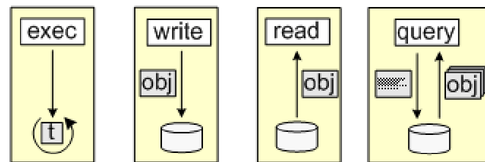


Figure 7.5: Experiment Categories *Exec*, *Read*, *Write*, and *Query*.

Read Category

The *Read* category refers to reading single data items, e.g. tuples or objects, which involve a few input parameters for identifying the data item to be read. For example, a CAD system requests single objects or a movie web application allows to look up information about movies, e.g. a user chooses a movie so that a web page is displayed with further information about this movie. The input parameters are required to identify the data item to be read from the remote system. The *Derby* case reads one tuple per request from the *lineitem* table identified by the *linenumber* and the *orderkey* of the *lineitem* tuple. The *File* case reads one comma-separated line from the *lineitem* file given two parameters, the *linenumber* and the *orderkey*. The *OOApp* case reads a plain *Employee* object, i.e. no object graph, given the *Employee*'s *id*. The experiments read an increasing number of data items (succeeding, single data item requests), i.e. *1*, *10*, *100*, *1,000* and *10,000* data items, to show that the VT overhead remains constant.

Write Category

The *Write* category refers to writing single data items, e.g. tuples or objects, which involve transfer of the data item in the request. For example, a CAD system writes single objects to the data store or a movie web application allows to store information about a movie. There are no parameters required since the complete data item is transferred to the remote system. The *Derby* case inserts a new *lineitem* tuple into the *lineitem* table, the *File* case appends a comma-separated *lineitem* line to the *lineitem* file, and the *OOApp* case inserts a new, plain *Employee* object into the object graph. The experiments write an increasing number of data items to the remote systems (succeeding, single data item requests), i.e. *1*, *10*, *100*, *1,000* and *10,000* data items, to show that the VT overhead remains constant.

Query Category

The *Query* category refers to set-oriented queries that require complex processing and that retrieve data sets. Typical examples are SQL queries or XQuery queries. The experiment queries have no parameters, but comprise a query expression that simulates a more or less complex retrieval request. This experiment category is the most complex and extensive one since there are different queries to be evaluated. The queries are differently evaluated by the different middleware systems and remote systems, which can drastically influence execution time. We chose representative queries that are typically and often used in application systems:

- Complete retrieval of all data, i.e. like

  ```
  SELECT *
  FROM   table
  ```

- Retrieving only a small subset of the data, i.e. like

  ```
  SELECT col1, col2
  FROM   table
  WHERE  col1 = x
  ```

  We choose a selectivity factor of one percent compared to the retrieval of the whole data set and we project only few columns so that the resulting data is less than one percent of the whole data in the table.

- Joining two data sets and retrieving the complete result set, i.e. like

  ```
  SELECT *
  FROM   table1, table2
  WHERE  table1.pk = table2.fk
  ```

- Joining two data sets and retrieving only a small subset of the join result, i.e. like

  ```
  SELECT col1, col2, colX, colY
  FROM   table1, table2
  WHERE  table1.pk = table2.fk AND col1 = x
  ```

  We again chose a selectivity factor of one percent compared to the retrieval of all data in the join result and we further project only few columns so that the resulting data is less than one percent of all data in the join result.

The *Derby* case actually performs SQL queries like the ones sketched above. The queries use the *lineitem* and the *orders* tables. The *File* case needs analogous processing on the *lineitem* and the *orders* files, which of course cannot be only performed by means of simple file operations. The file data has to be explicitly postprocessed in a higher layer, e.g. in the middleware or in the client. The *OOApp* case needs similar processing on *Employee* objects and on *Department* objects, which cannot be performed by the OOApp remote system itself, either. The objects have to be explicitly postprocessed in a higher layer, too, e.g. in the middleware or in the client.

7.2.2 Query Considerations

Thus, the big question is where the processing of the different queries is actually performed. The answer is simple for the WebSphere scenario since WebSphere and the J2EE connector architecture cannot handle set-oriented queries that contain selections, projections or joins. Therefore, the WebSphere client has to retrieve the required data sets and then performs the necessary processing on its own. The only exception is the *Derby* case since the Derby J2EE connector can push down an SQL query to the Derby server thereby avoiding any query processing in the WebSphere client. The DB2 client can submit the different queries as SQL queries to DB2 since SQL is powerful enough to express such queries. DB2 in turn performs the necessary processing on the data retrieved from the remote systems, but, again, the *Derby* case is an exception since DB2 can push down all queries to the Derby server.

We can already see the advantages that DB2 comes along with for the Query category since DB2 handles all necessary processing whereas the WebSphere client has to handle query processing on its own. Similar considerations hold for the VT because if we add the VT to the processing stack without any query capabilities, it breaks the push-down mechanism that facilitates more efficient processing in the *Derby* case. If the VT however offers query capabilities that comprise at least the power of the queries in the *Query* category, we are able to push down queries to the VT or even further. This does not only mean that we can push down queries to the Derby server as in the DB2 and WebSphere scenarios, but we can also offer the VT query capabilities to the WebSphere case so that the WebSphere client pushes down all its queries to the VT. This provides for a more efficient execution of set-oriented queries in the VT-based experiments. We will discuss this in more detail later.

The experiments in the *Query* category are performed on different databases of exponentially increasing size (factor 10 for each scale level) ranging from very small data sets (scale level 1) up to large data sets (scale level 6). The TPC-H benchmark especially targets very large data sets. We extend the defined scale levels to very small data sets so that the smallest scale level (scale level 1) contains six tuples in the *lineitem* table (*Derby* case) and in the comma-separated *lineitem* file (*File* case) and one tuple in the *orders* table (*Derby* case) and in the comma-separated *orders* file (File Case). The highest scale level (scale level 6) contains 600,000 tuples and 150,000 tuples, respectively. Requests on this data set can last quite long, even up to a few hours so that we did not include higher scale levels in our experiments. Higher scale levels are even not necessary since the experiments yield clear results for the given scale levels. The *OOApp* case is based on object graphs that have to be completely managed in main memory and that additionally have a higher memory consumption due to the inherent object management overhead. Therefore, our experiments have only five scale levels. Scale level 1 contains 15 objects for the *Employee* class and 1 object for the *Department* class. Scale level 5 contains 150,000 *Employee* objects and 10,000 *Department* objects. We could not apply scale level 6 with 1,500,000 objects. The reason was not a too long execution time, but simply the main memory of the hosts, which was not sufficient enough to deal with such

a large number of objects in one chunk. This is not surprising, but even desired since the *OOApp* case represents non-scalable application systems. Moreover, the experiments yielded clear results for the given scale levels so that we did not need higher scale levels.

7.2.3 Experiment Architectures

Figure 7.6 shows the three and four layer architectures of the SQL wrapper integration scenarios that we use for the experiments. Each layer is executed on a separate host, which requires up to four hosts for each experiment scenario. We compare the execution time of the native SQL wrapper execution in DB2 (shown on the left side) with the SQL wrapper execution in the VT (shown on the right side) to determine the overhead that the VT places for SQL wrappers. We additionally compare the execution time of the native SQL wrapper execution with the J2EE connector execution in the VT (shown in the lower part). We do so since we are interested in the behavior and in the execution times of different adapter technologies in the VT. For example, if we assume that we already have the file J2EE connector, but no file SQL wrapper, we can compare how the execution time of the file J2EE connector in the VT, which obeys the *adapter reuse pattern*, relates to the execution time of a new, recently developed file SQL wrapper in DB2, which obeys the *implementation pattern*. Note that the VT is able to handle SQL wrappers as well as J2EE connectors, but that the DB2 server can only handle SQL wrappers.

Figure 7.7 correspondingly shows the three and four layer architectures of the J2EE connector integration scenarios that we use for the experiments. Again, each layer is executed on a separate host, which requires up to four hosts for each experiment scenario. We compare the execution time of the native J2EE connector execution in WebSphere (shown on the left side) with the J2EE connector execution in the VT (shown on the right side) to determine the overhead that the VT places for J2EE connectors. We also additionally compare the execution time of the native J2EE connector execution with the SQL wrapper execution in the VT (shown in the lower part). The reason why we do this is analogous to the SQL wrapper scenario. For example, if we assume that we already have the file SQL wrapper, but no file J2EE connector, we can compare how the execution time of the existing file SQL wrapper in the VT, which obeys the *adapter reuse pattern*, relates to the execution time of a new, recently developed file J2EE connector in WebSphere, which obeys the *implementation pattern*. Note that the VT is able to handle SQL wrappers as well as J2EE connectors, but that the WebSphere server can only handle J2EE connectors.

7.2.4 Experiment Notation

Now we can establish a notation for the different experiments we perform. An *experiment name* constitutes in the following way:
<operation>-<mode>-<compensation>-<mw>-<vt_adapter>-<rs>-<scalelevel>

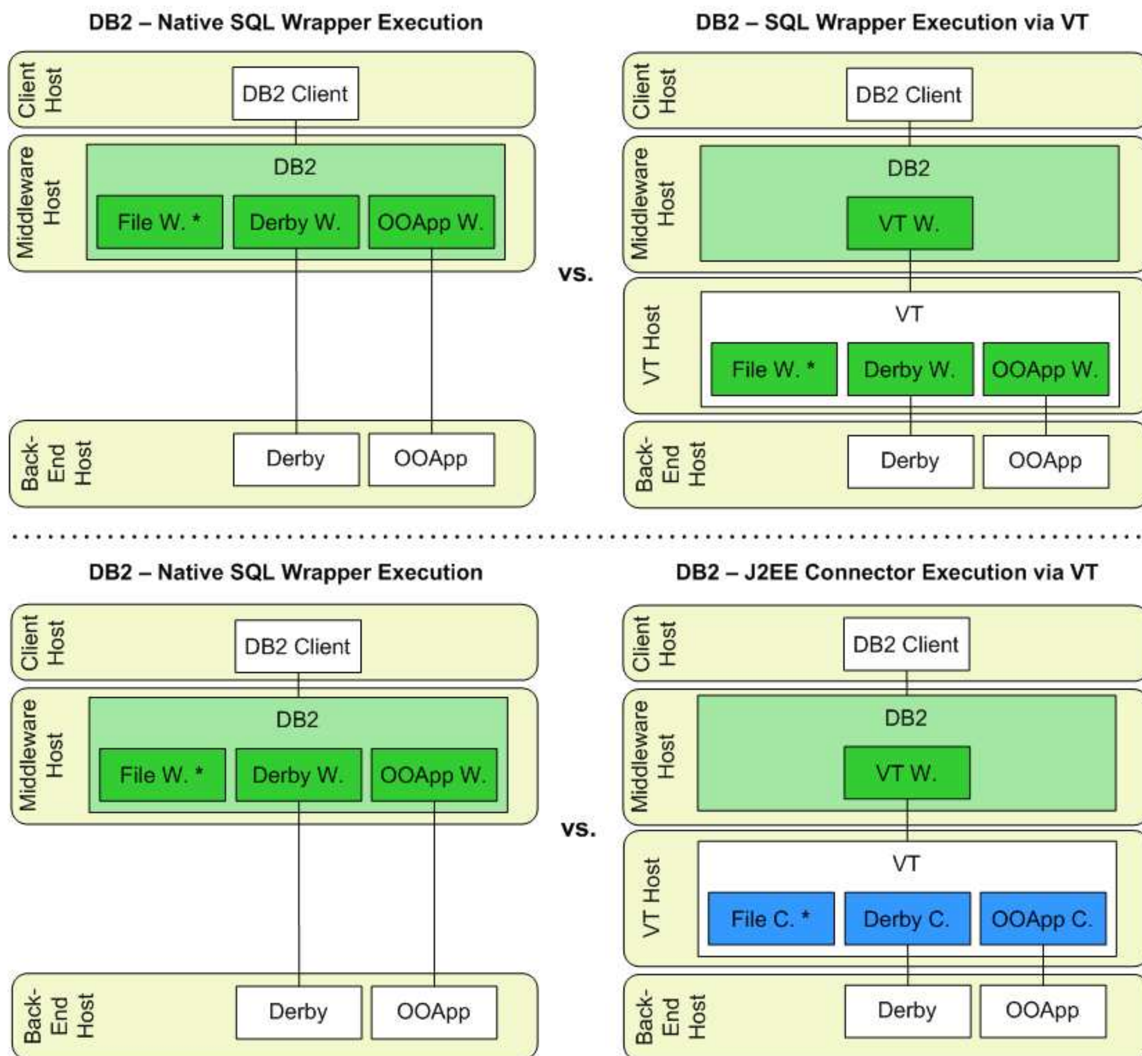


Figure 7.6: SQL Wrapper Experiments Architectures.

The single elements stand for:

- <*operation*>: *Exec*, *Read*, *Write*, *Query*; the operation category.
- <*mode*>: *R*, *SP*, *J*, *SPJ*; the query mode with *R* as retrieval, *S* as selection, *P* as projection, *J* as join; this only applies to the *Query* case.
- <*compensation*>: *Comp*; if missing processing capabilities in the VT have to be compensated by another system, otherwise omitted; this only applies to the *Query* case.
- <*mw*>: either *WS* or *DB2*; the middleware system, i.e. WebSphere or DB2 Information Integrator.
- <*vt_adapter*>: *VT-W* or *VT-C*; the adapter employed in the VT, i.e. the VT

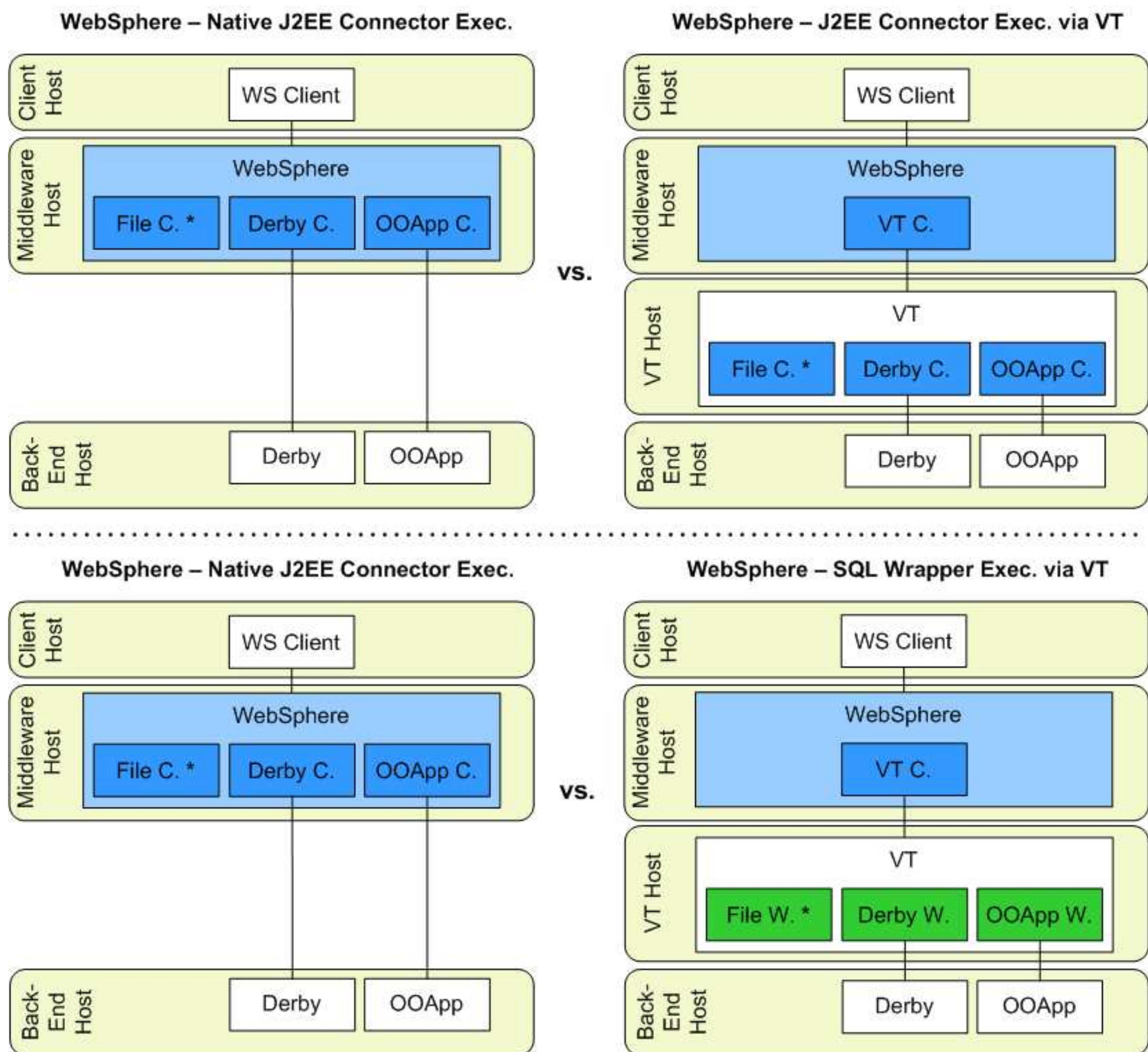


Figure 7.7: J2EE Connector Experiments Architectures.

executing an SQL wrapper or the VT executing a J2EE connector if it is a VT-based scenario, otherwise omitted.

- *<rs>*: *File*, *Derby*, *OOApp*; the remote system, i.e. files or Derby or the object-oriented application system.

- *<scalelevel>*: the scale level; *1*, *2*, *3*, *4*, *5*, *6* for the *Query* category; *1*, *10*, *100*, *1000*, *10000* for the other categories.

Examples of experiment names are *Exec-WS-File-100*, *Read-WS-VT-W-Derby-10*, *Query-R-DB2-OOApp-3* or *Query-SPJ-Comp-WS-VT-C-OOApp-2*. An experiment name that omits the scale level represents the whole experiment series including all scale levels, e.g. *Exec-WS-File* implicitly comprises *Exec-WS-File-1*, *Exec-WS-File-10*, *Exec-WS-File-100*, *Exec-WS-File-1000*, *Exec-WS-File-10000*; and

Query-R-DB2-OOApp comprises *Query-R-DB2-OOApp-1*, *Query-R-DB2-OOApp-2*, *Query-R-DB2-OOApp-3*, *Query-R-DB2-OOApp-4*, *Query-R-DB2-OOApp-5*. An experiment name that omits the remote system and the scale level represents the whole experiment series of all remote systems including all scale levels, e.g. *Exec-WS* implicitly comprises *Exec-WS-File*, *Exec-WS-Derby*, *Exec-WS-OOApp* and *Query-R-DB2* comprises *Query-R-DB2-File*, *Query-R-DB2-Derby*, *Query-R-DB2-OOApp*. An experiment consists of a corresponding request that is successively executed a number of times to yield a reliable and stable result. A request execution starts and ends in the client application, i.e. from request submission until the response is completely processed, which is especially important when reading data.

Each experiment without the VT is compared to a corresponding experiment that additionally employs the VT, i.e. one experiment that only uses DB2 or WebSphere, e.g. *Exec-WS-File-1*, and one that additionally employs the VT, e.g. *Exec-WS-VT-C-File-1*. The comparison result is the overhead in percent that the VT causes, e.g.

$$\frac{t_{Exec\text{-}WS\text{-}VT\text{-}C\text{-}File\text{-}1} - t_{Exec\text{-}WS\text{-}File\text{-}1}}{t_{Exec\text{-}WS\text{-}File\text{-}1}} * 100.$$

7.3 Challenges

The execution of experiment requests is subject to different disruptive influences which come from the software systems, operating systems, network communication and other system processes and software running on the hosts. We spent considerable time and efforts to reduce these factors to a minimum where possible, but there still are some relevant system characteristics and influences that cannot be completely eliminated since they are crucial for proper execution or since they even are an inevitable part of the system environment. Examples are security software, network load caused by other users and systems, operating system and hardware components, e.g. reading and writing files, sockets for network communication, essential DBS reorganization procedures or JVM management functionality such as garbage collection.

Important to note is that every host runs a JVM since we implemented the VT prototype and the SQL wrappers and J2EE connectors in Java. Therefore, the JVMs played a major role in the infrastructure of our experiment scenario. A JVM is a complex software that dynamically allocates memory in cooperation with the operating system. The allocated memory is managed by a garbage collection facility that automatically reserves and frees object memory. Therefore, some experiments typically degenerated due to memory consumption and corresponding garbage collection actions, especially in the *OOApp* case, which deals with large object graphs. This behavior especially influences experiments with higher scale levels, but also holds for lower scale levels since we executed experiment requests consecutively and with high repetition rates to even out execution time deviations so that the JVM earlier or later had to start over with inevitable garbage collection procedures. This restricted the number of consecutive request executions. Typically, the faster a se-

ries of succeeding request executions was affected by degeneration effects, the less precise the final execution time result got so that we had to re-execute such experiments more than once. Reasons for fast degenerating series of succeeding request executions are long running requests, i.e. experiments with high scale levels.

The waiting time between request executions has also shown to be significant since some experiments tended to yield oscillating execution times for succeeding request executions. Such behavior established patterns like $x, y, x, y, ...$ or $x, x, y, x, x, y, ...$ or $x, x+c, x+2c, x+3c, ..., x+8c, x, x+c, x+2c, x+3c, ..., x+8c, ...$ (with x, y and c standing for some ms of execution time). This behavior partially improved with longer waiting times between two succeeding request executions. The reasons partially were garbage collector behavior, but it also had other reasons like operating system caching. Other experiments continually decreased execution time for succeeding request executions so that we partially had to execute some hundred requests until the execution time results became stable. Here, the JVM just-in-time compiler (JIT) played a major role, but also prefetching and caching mechanisms of operating systems or database systems contributed to the decreasing execution time of succeeding request executions.

The effects of these influences became even worse since we used up to four hosts for each experiment with corresponding systems and communication between them. Additionally, the final execution time results of our experiments are the comparison between two experiments, i.e. one experiment with the VT and one without, to determine the overhead of the VT so that we even have up to seven hosts that contribute to the final execution time results and thus to the final deviation too. Therefore, we repeatedly executed the same request consecutively up to a few thousand times in the worst cases to yield precise execution time results, especially for series of request executions that took quite long to improve execution time due to JVM JIT compilers, caching effects, etc. Higher scale levels of course did not allow to repeat a request execution very often. Therefore, we had to start over with request executions several times for long-running requests, i.e. restart systems and clients. Deviations here were relatively less significant due to the longer execution times. JIT phenomena also applied faster to longer running request executions. In summary, the execution time results slightly deviate to some extent, typically up to only one percent overhead in the worst case. This contributes to a sound picture of the yielded experiment results.

7.4 Experiment Results

The main result of the experiments is that the longer a request execution lasts and the larger the data sets are that are read or written, the better performs the VT, i.e. the lower is the overhead. If request executions have a very short execution time or if very small data sets are read or written, the VT creates some overhead. However, this can be usually neglected in integration scenarios since the overhead is only up to a few hundred ms in the worst cases. The overall result of the experiments is

that the VT actually is practically applicable. Another important result is that the execution times of SQL wrappers in the VT basically do not significantly differ from the execution times of J2EE connectors in the VT. This resumes from two facts: the already stated fact that adapters for the same remote system basically perform the same integration tasks, e.g. reading or writing data from or to the remote system, and the fact that the whole integration scenario, its complexity and partially suboptimal request executions lead to significant influences on execution times so that the time differences in the execution of adapters of different adapter technologies only contribute a small part to the overall execution time. This means that the VT can efficiently handle diverse adapter technologies independent of the context where it is used.

In the next four subsections, we discuss the experiment results of the four experiment categories, respectively. At the beginning of each subsection we first give a rough estimation of what we expect from the results. In the second part of each subsection we discuss the actual results of the experiments and the deviations from the estimations if there are significant ones.

7.4.1 Operations

An experiment of the *Exec* category consists of a request that executes a remote operation. The remote operation is a simple wait operation that waits for a given number of ms and thereby simulates a remote operation with an execution time of 1, 10, 100, 1,000 or 10,000 ms (cf. Section 7.2).

Expectations

An operation execution time of 1 ms is quite short compared to the overall request execution time, which includes the execution of the request in the client, in the middleware and in the VT, and the execution of the remote operation in the remote system, which either is the Derby stored procedure, the OOApp RMI method or the local wait routine in the file adapter (also see Figure 7.4). The processing in the different systems and the communication between them takes at least a few ms. Therefore, the VT introduces some overhead for the low scale levels, i.e. operation execution times of 1 ms and 10 ms, and a significantly decreasing overhead for higher scale levels, i.e. operation execution times of 100 ms and higher. The VT overhead is higher for the *File* case compared to the *Derby* case and the *OOApp* case since the *File* case has one layer less than the other cases: the *File* case has two layers with DB2 or WebSphere only and three layers with the VT, the other cases have three layers with DB2 or WebSphere only and four layers with the VT. The *File* case has one network connection with DB2 or WebSphere only and two with the VT, the other cases correspondingly have two network connections with DB2 or WebSphere only and three with the VT. Therefore, a rough estimation of how the requests will behave expects about 50 to 100 percent VT overhead for the *File* case (about 100 percent overhead for communication and about 50 percent overhead for processing),

and about 33 to 50 percent VT overhead for the other two cases (about 50 percent overhead for communication and about 33 percent overhead for processing) for low scale levels. The higher the scale levels get, the more the VT overhead will decrease.

Experiments

And indeed, there is some overhead for requests with operation execution times of 1 and 10 ms (see Figures 7.8 and 7.9). Especially, the *File* case shows a higher overhead due to the worse relation of 2:3 in contrast to 3:4 in the *Derby* case and in the *OOApp* case. Nevertheless, the time for processing the request in the systems except the remote operation execution time, i.e. the waiting time, always is only a few ms, i.e. between 2 and 5 ms. This means that processing and routing the request through the different systems of the integration scenario takes a constant time of only a few ms and therefore is independent of the overall request execution time, which in turn is proportionally increasing corresponding to the remote operation execution time of 1 to 10,000 ms. This in turn means that the remote operation execution time is the dominating factor of the *Exec* category experiments and therefore the VT overhead of the 1 ms and 10 ms execution requests is absolutely seen only a few ms, which in most cases will not seriously affect performance. The experiments with a request execution time of 100 ms or higher yield a low VT overhead of less than one percent and therefore such requests are applicable to any environment.

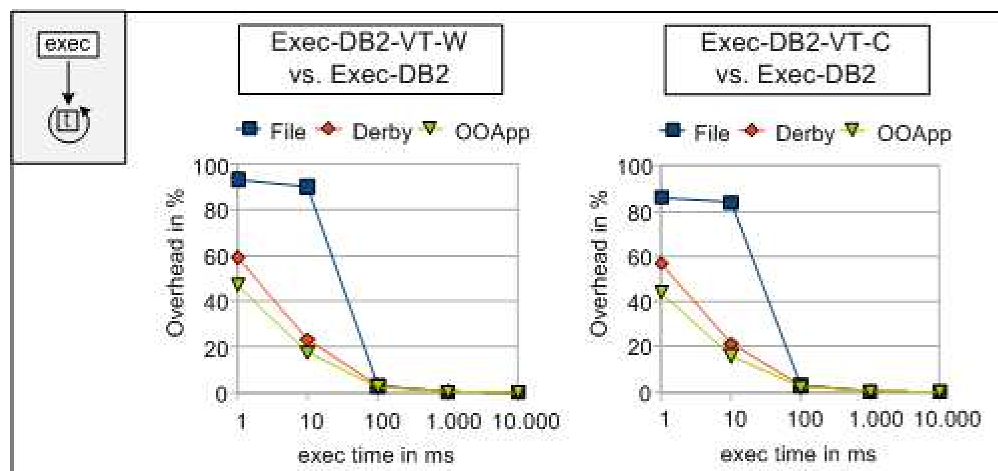


Figure 7.8: Exec-DB2.

7.4.2 Reading Data

The *Read* category reads single data items from a remote system, i.e. 1, 10, 100, 1.000 or 10.000 tuples from the Derby *lineitem* table, lines from the *lineitem* file or *Employee* objects from the OOApp (cf. Section 7.2).

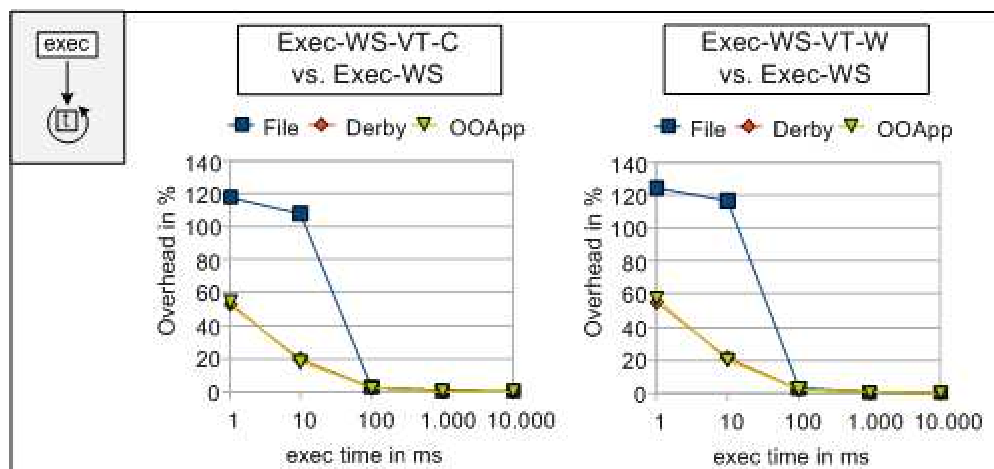


Figure 7.9: Exec-WS.

Expectations

Reading only few data items will result in a very short execution time so that the preparations for executing the request in the different systems will considerably influence the execution time. Reading more than few data items will reduce the preparation part. The task of retrieving, processing and transferring the data items will consume most of the request execution time. Therefore, the VT overhead will be higher for reading only few data items, but it will be lower and also constant for more than few data items. We roughly expect about 50 to 100 percent VT overhead for the *File* case and about 33 to 50 percent VT overhead for the other two cases, analogous to the *Exec* category, but for all scale levels, i.e. we expect constant VT overhead.

Experiments

The execution time results partially behave as expected or slightly deviate from the expectations, but they also show some surprising, unexpected effects, that even decrease the expected VT overhead (see Figures 7.10 and 7.11). The *File* case completely differs from our expectations since the VT does not introduce a runtime overhead, but even places a "negative" runtime overhead, i.e. a runtime improvement by using the VT! The reason for this behavior is that integration scenarios are inherently complex and therefore tend to run suboptimally considering their efficiency. Therefore, adding another component to such a complex environment may not further increase execution time of an already suboptimal request execution, but could make a suboptimal execution less suboptimal, i.e. decrease execution time. The crucial point is that the different systems do not run and work isolated, but they are interconnected and correlated so that the overall infrastructure becomes more than the sum of its parts. The infrastructure is quite complex and the dependencies and mutual influences of its parts are neither easily comprehensible nor

easily to predict so that some experiments lead to execution times that are different to what we originally expected. This requires to take into account further aspects and to regard the individual parts of the infrastructure in a more global context. Therefore, we discuss this and other experiment phenomena in further detail in Section 7.5. The *Derby* case and the *OOApp* case show similar effects and reveal even further characteristics, i.e. leaps. Reading 1, 10 and 100 data items shows some VT overhead, as expected or a little bit more than expected in a few cases, which still is acceptable. However, the higher scales, i.e. reading 1.000 and 10.000 data items, show significantly lower VT overhead although they should yield the same VT overhead as for reading 1, 10 and 100 data items.

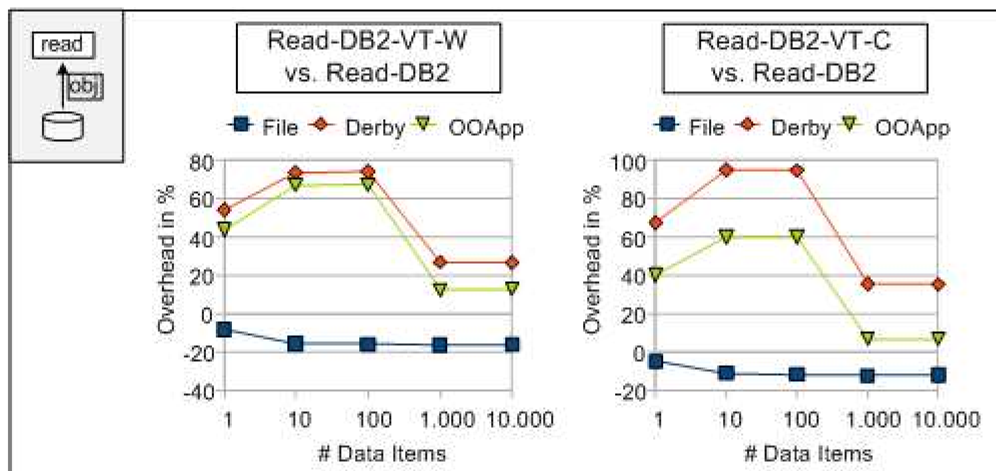


Figure 7.10: Read-DB2.

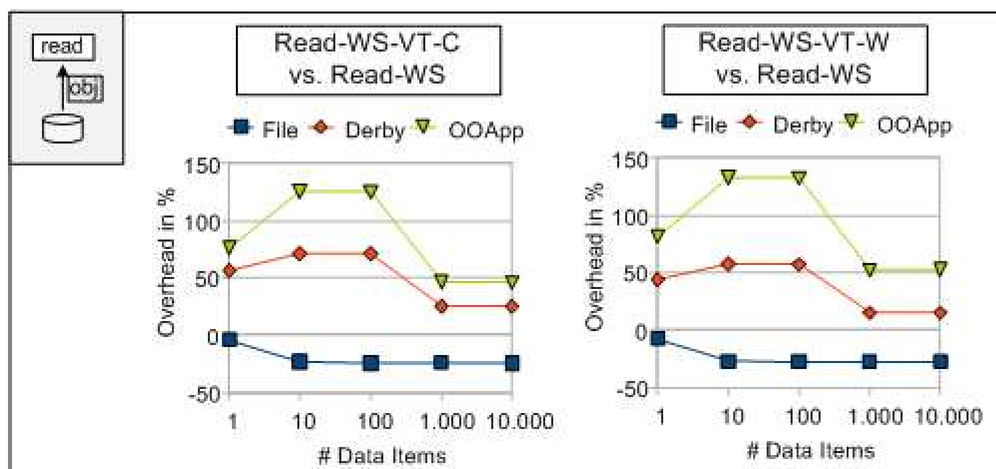


Figure 7.11: Read-WS.

We further investigated this phenomenon and detected a leap between reading 104 data items and 105 data items (see Figures 7.12 and 7.13). The absolute execution time for reading 104 data items yields some 100 ms, whereas reading 105

data items yields an absolute execution time that is about two to three times higher than the execution time for reading 104 data items (see Figures 7.14 and 7.15). This phenomenon is another unexpected behavior that we also discuss in more detail in Section 7.5. However, the important message of these execution time results is that the experiments behave as expected or even perform better than expected, which is a very satisfying result.

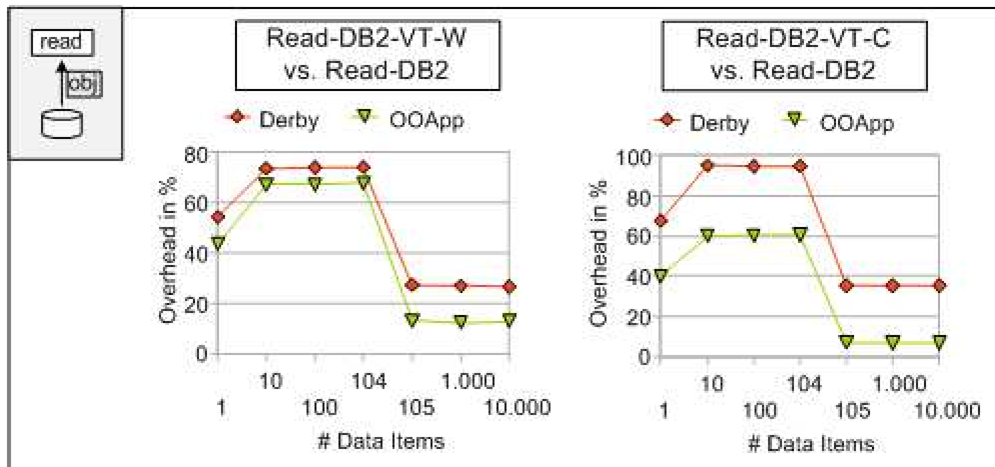


Figure 7.12: Read-DB2: Leaps.

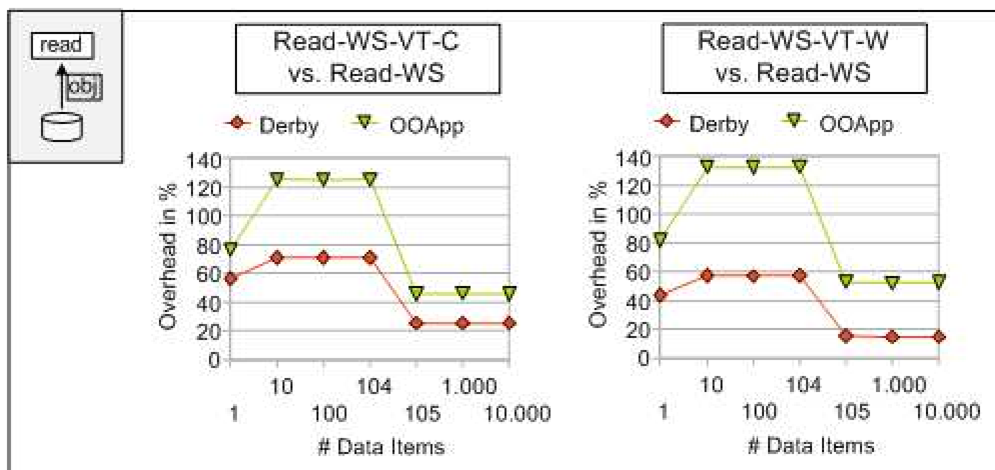


Figure 7.13: Read-WS: Leaps.

7.4.3 Writing Data

The *Write* category writes single data items to a remote system, i.e. 1, 10, 100, 1.000 or 10.000 tuples to the Derby *lineitem* table, lines to the *lineitem* file or *Employee* objects to the OOApp (cf. Section 7.2).

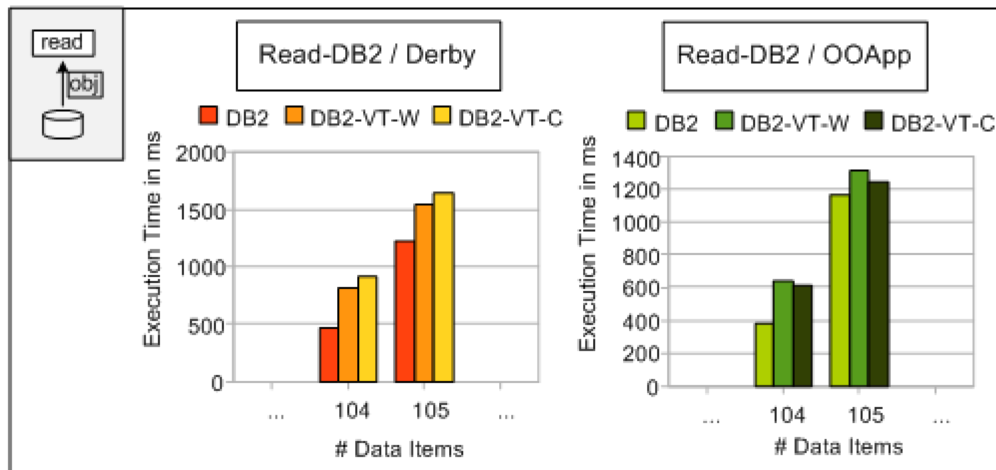


Figure 7.14: Read-DB2: Leaps (absolute execution time).

Figure 7.15: Read-WS: Leaps (absolute execution time).

Expectations

The *Write* category will behave analogously to the *Read* category, i.e. about 50 to 100 percent VT overhead for the *File* case and about 33 to 50 percent VT overhead for the other two cases. The VT overhead for writing few data items will be higher than the VT overhead for writing more than few data items due to the very short request execution time for writing few data items and the constant time for preparing the request execution in the different systems.

Experiments

The execution time results behave similarly to the *Read* category. Some of the execution time results behave as expected and few slightly deviate from the expectations, but there also are some surprising, unexpected execution time results decreasing the

expected VT overhead (see Figures 7.16 and 7.17). The *File* case behaves as expected in the WebSphere case and as expected for writing 1, 10 and 100 data items in the DB2 case, but then it shows a leap so that the DB2 *File* case with 1.000 and 10.000 data items even improves runtime by using the VT. There are two leaps in the DB2 *File* case (see Figure 7.18), one between writing 108 and 109 data items, and another leap between writing 134 and 135 data items. The first leap increases execution time of the VT cases to almost four times and the second leap increases execution time of the non-VT case to more than five times (see Figure 7.19). The other cases behave in the same way (see Figures 7.16 and 7.17): the *Derby* case and the *OOApp* case perform as expected or deviate slightly for writing 1 and 10 data items, but thereafter leaps occur so that writing 100 and more data items yield better execution time results than expected. The leaps occur at different positions: the DB2 *Derby* case has two leaps (see Figure 7.20), one between writing 62 and 63 data items and one between writing 72 and 73 data items. The leap between 62 and 63 doubles execution time for the VT cases, which results in a temporary high VT overhead (see Figure 7.21). The leap between 72 and 73 more than doubles the execution time for the non-VT case, which reduces the VT overhead permanently to a very low value, which is lower than expected. The DB2 *OOApp* case behaves similarly (see Figure 7.22). The leap between writing 90 and 91 data items almost doubles execution time for the VT cases (see Figure 7.23), which temporarily increases VT overhead, and the leap between writing 99 and 100 data items more than doubles the execution time for the non-VT case thereby improving the overall runtime by using the VT. The WS *Derby* case and the WS *OOApp* case show only one leap (see Figures 7.24 and 7.25). The leap in the WS *Derby* case is between writing 52 and 53 data items where the execution time is doubled for the VT cases and more than doubled for the non-VT case (see Figure 7.26), which results in a lower VT overhead than expected. The leap in the WS *OOApp* case is between writing 34 and 35 data items where the VT cases almost double execution time and where the non-VT case more than doubles execution time (see Figure 7.27), which again results in a lower VT overhead than expected. Refer to Section 7.5 for a discussion of the leap phenomenon. Again, the important message is that the execution time results perform as expected or even better than expected and thereby provide very good results.

7.4.4 Queries

The *Query* category performs complex, declarative requests, i.e. set-oriented queries, that retrieve data sets from remote systems of different scale levels, i.e. Derby databases, files and the OOApp (cf. Section 7.2).

Expectations

Basically we expect a similar behavior as for the *Read* and *Write* cases, although there are some differences. We again expect about 50 to 100 percent VT overhead

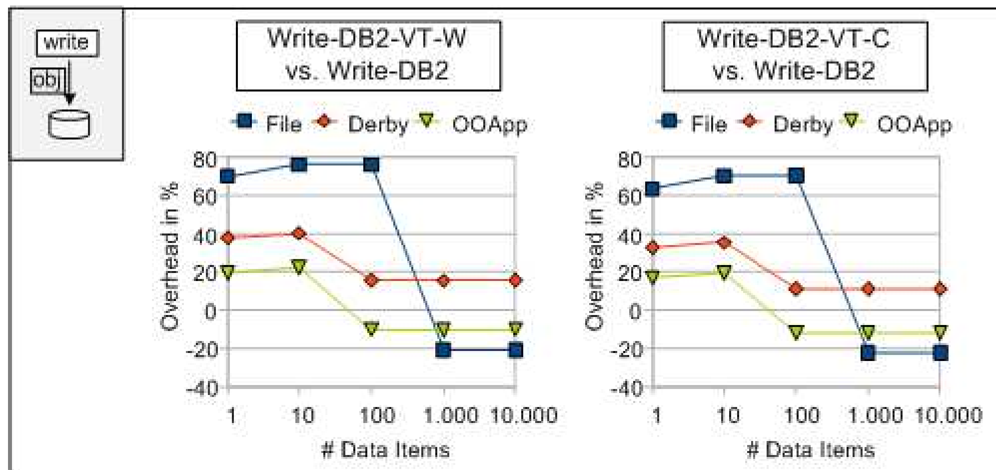


Figure 7.16: Write-DB2.

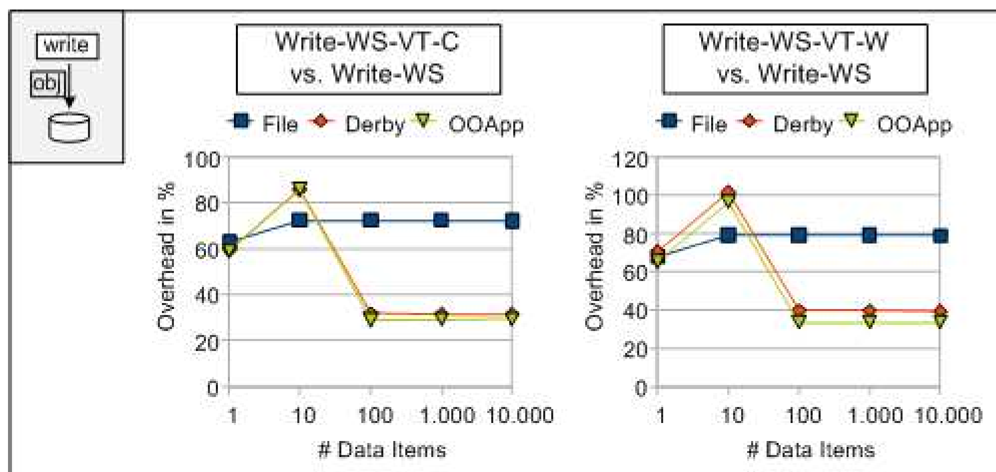


Figure 7.17: Write-WS.

for the *File* case and about 33 to 50 percent VT overhead for the *Derby* case and for the *OOApp* case. However, this only holds for very small data sets since we employ a pipeline processing model as it is typically done in a DBS, for example. The VT does not perform each processing step strictly separated from the others, but pipelines them so that the result sets can be processed in parallel and execution time is sped up. For example, a straight forward approach would be the following: the VT retrieves the whole result set from a remote system at once. Then it uses the result set to apply some processing on the contained data items. Finally it submits the processed result set to the calling middleware system, e.g. DB2 or WebSphere, and so on. This processing strategy is blocking and time-waisting and therefore we apply a pipelining model so that the VT, the adapters and the clients work in parallel where possible (see Figure 7.28). The gray boxes represent the operating system processes that execute middleware client, middleware system, VT and remote system. The small boxes represent threads that run within the operating system

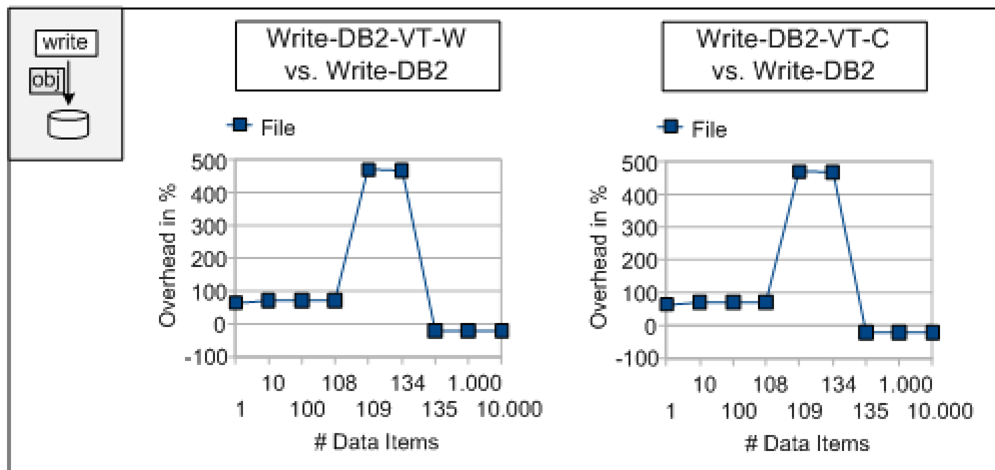


Figure 7.18: Write-DB2-File.

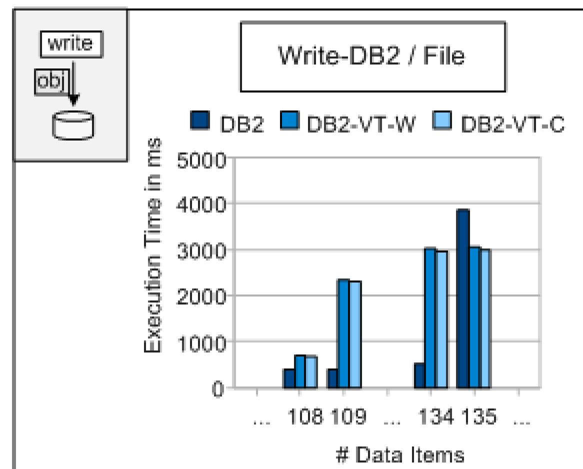


Figure 7.19: Write-DB2-File (absolute execution time).

processes and that execute in parallel to each other where possible. For example, the VT retrieves the result set (*access & preprocessing*), processes it (*request processing*) and submits it to the caller (*access & dispatching*) in parallel. In that way, we can reduce the VT overhead since the VT is just another filter in the "data stream" that flows from a remote system to a middleware client. Of course, this does not work very well for very small data sets, i.e. low scale levels, but we expect that it works better the larger the result sets become, i.e. especially for scale level 3 and higher.

Experiments

The first query is a basic read operation, i.e. `SELECT * FROM x` where x is *lineitem* for the *File* and the *Derby* cases and *Employee* for the *OOApp* case. The query simply reads all *lineitem* data or all *Employee* objects from the remote systems. The request execution time behaves as expected or better for the low scale levels

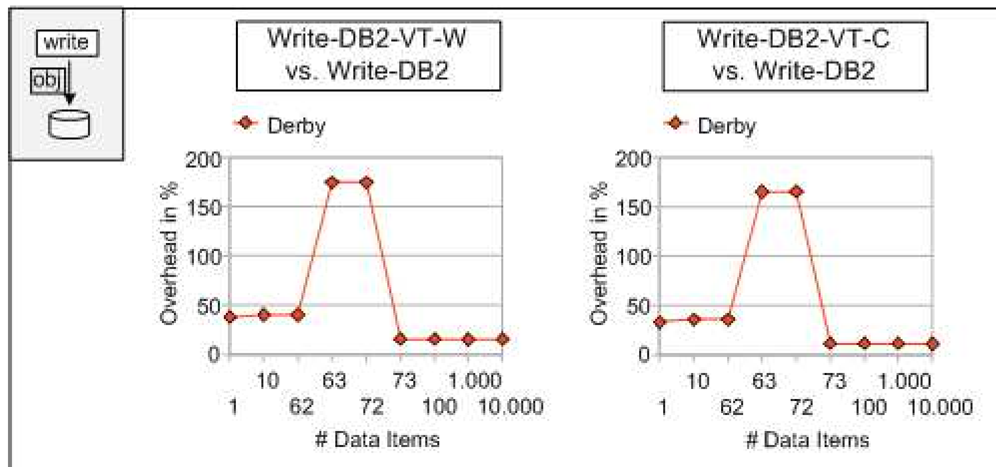


Figure 7.20: Write-DB2-Derby.

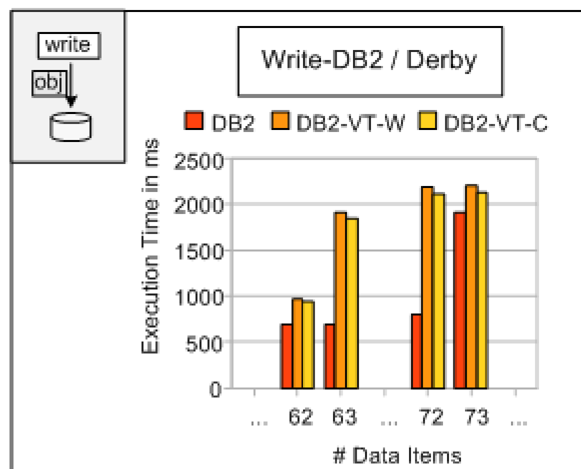


Figure 7.21: Write-DB2-Derby (absolute execution time).

(see Figures 7.29 and 7.30). The queries perform even better for the higher scale levels since the pipeline effect positively influences the execution time of the VT cases as expected and thereby reduces the VT overhead.

The second query is a read operation that returns only a small part of the whole data set, i.e. we perform a projection and a selection in VTQL like `SELECT c1, c2, c3 FROM x WHERE c4 = v1`. We select one percent of the whole data set, i.e. *lineitem* data or *Employee* data, and additionally return only a few attributes or columns of them thereby significantly reducing the returned data amount to less than one percent compared to the first query. However, we face several problems since processing complex requests such as VTQL queries requires corresponding capabilities as they are provided by a DBS, for example. The DB2 case supports query processing by means of the DB2 query engine so that the second query can be processed more efficiently than in the WS case since WebSphere does not have such processing capabilities. This means, that the WebSphere client can only forward

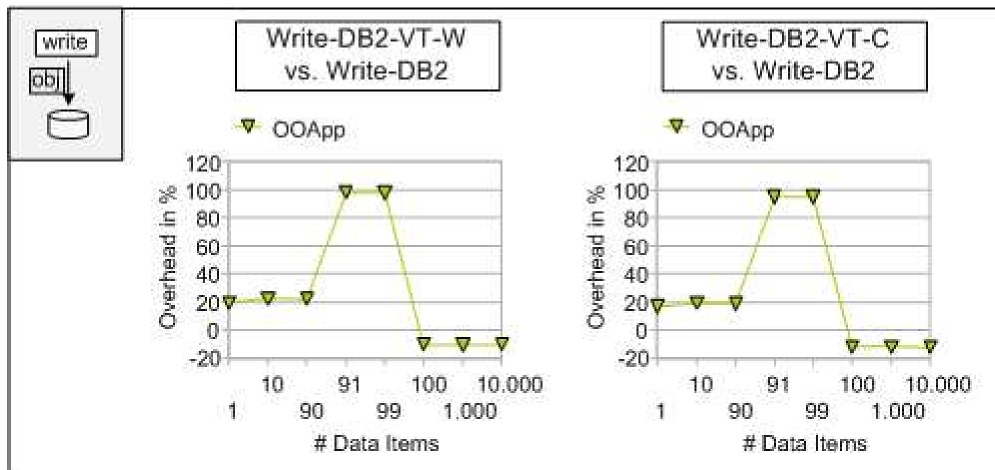


Figure 7.22: Write-DB2-OOApp.

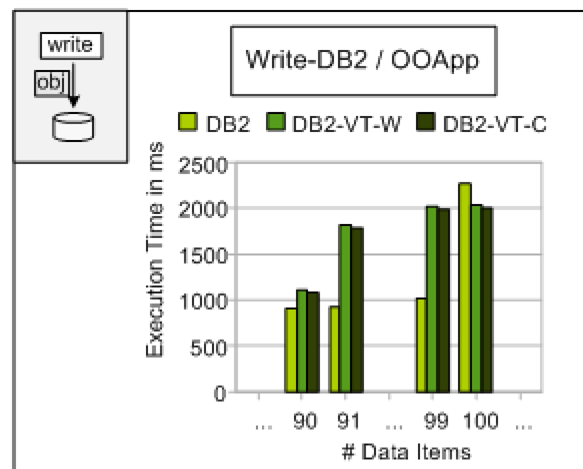


Figure 7.23: Write-DB2-OOApp (absolute execution time).

a corresponding SQL query string via WebSphere and the Derby J2EE connector to the Derby server, but the *File* case and the *OOApp* case require to retrieve the whole data set to the WebSphere client where the necessary processing has to be compensated for (see Figure 7.31). The DB2 case works differently since DB2 can process the SQL query of the DB2 client very well. It hands over a corresponding SQL string to the Derby server similar to the WS *Derby* case (called *push-down*) and it compensates for the missing query capabilities in the *File* case and in the *OOApp* case. The advantage of the DB2 case over the WS case is that the DB2 client does not need to retrieve whole data sets in the *File* case and in the *OOApp* case. This is done by DB2 since DB2 can process the data sets and return the significantly smaller result data sets to the DB2 client (see Figure 7.31). The VT case works even worse if it does not have any query processing capabilities, i.e. if the only supported operation is retrieval of the whole data set (see Figure 7.32). In that case, the VT introduces an incredible overhead for the *Derby* cases for both the DB2 case as well

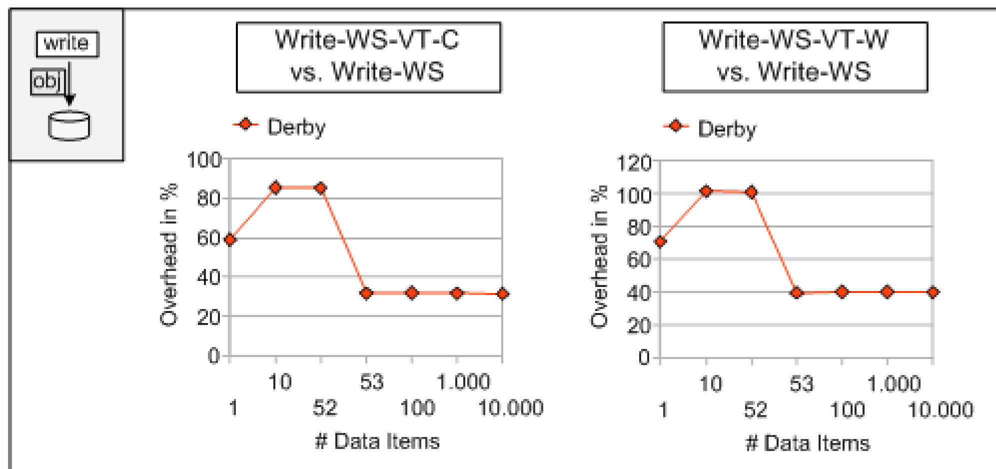


Figure 7.24: Write-WS-Derby.

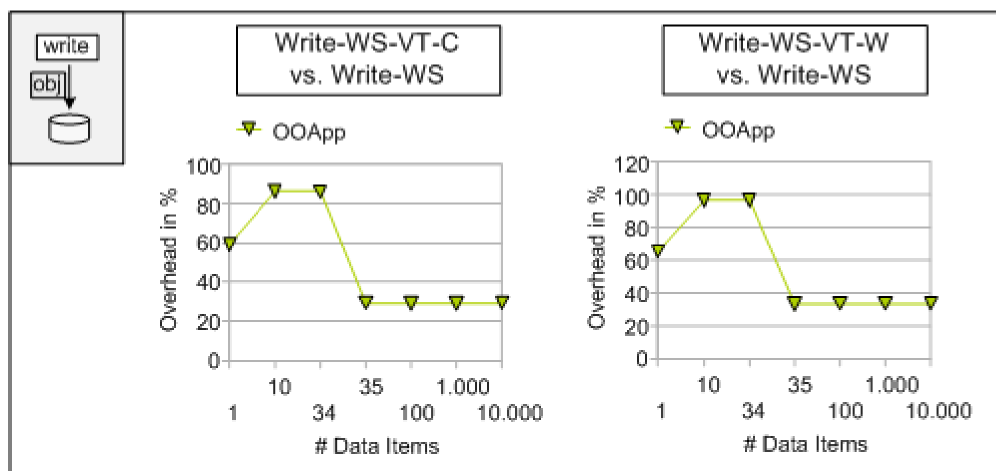


Figure 7.25: Write-WS-OOApp.

as for the WS case if the VT is not able to hand over the SQL query to Derby, but just retrieves the whole data set (see Figures 7.33 and 7.34). The dent at scale level five stems from unexpected optimization effects. The other WS cases, i.e. the *File* case and the *OOApp* case, work as expected, but the DB2 *File* case and the DB2 *OOApp* case suffer from the same drawbacks as the *Derby* case although not in such a serious manner (see Figures 7.35 and 7.36). Nevertheless, even these two cases introduce an overhead of some hundred percent due to the limited processing capabilities of the VT. The conclusion from this behavior is clear: the VT requires sophisticated query processing capabilities to suitably and efficiently process complex requests. In that case, we get a completely different behavior, which is shown in Figure 7.37. DB2 can now push-down all queries to the VT and only the VT has to retrieve the whole data sets in the *File* case and in the *OOApp* case since the VT now is able to compensate for the missing query capabilities of the *File* case and the *OOApp* case. Moreover, the VT can push-down the SQL query to the Derby server, which completely avoids

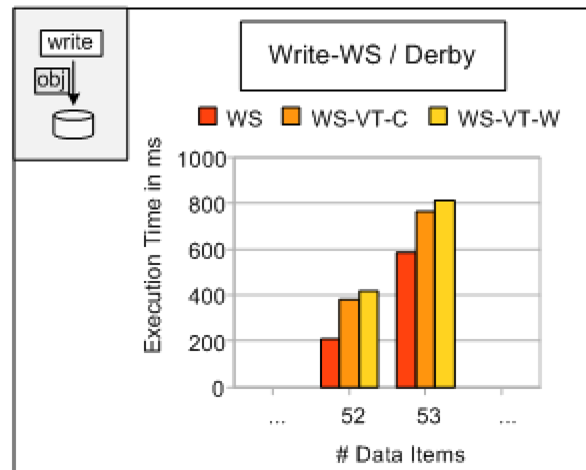


Figure 7.26: Write-WS-Derby (absolute execution time).

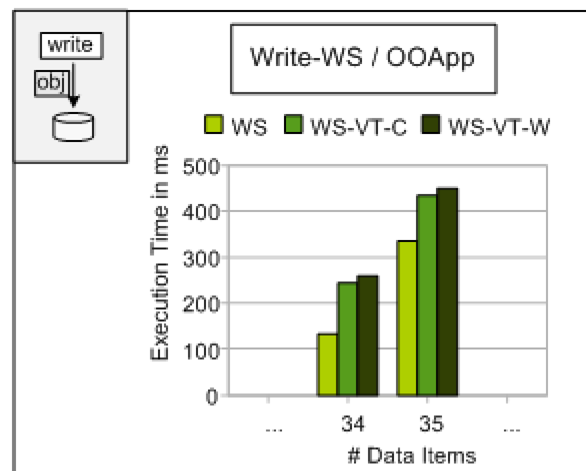


Figure 7.27: Write-WS-OOApp (absolute execution time).

transfer of the whole data set. The WS case behaves even much better since the VT can actually compensate for the missing query capabilities of WebSphere in the *File* case and in the *OOApp* case, i.e. the VT processes the query, so that WebSphere already receives the final result, which in turn avoids query compensation in the WebSphere client (compare Figures 7.32 and 7.37). This is the main reason beside the unexpected optimization effects why the WS *File* case and the WS *OOApp* case significantly speed up query processing when the VT is employed (see Figures 7.38 and 7.39).

The third query is a classic equi-join query like `SELECT * FROM x,y WHERE x.c1 = y.c3`, which returns a data set that is larger than the sum of its constituents, i.e. *lineitem* and *orders* data for the *File* case and for the *Derby* case and *Employee* and *Department* data for the *OOApp* case. If we use the push-down capability of the VT, the execution time results of the DB2 case behave as expected, the execution

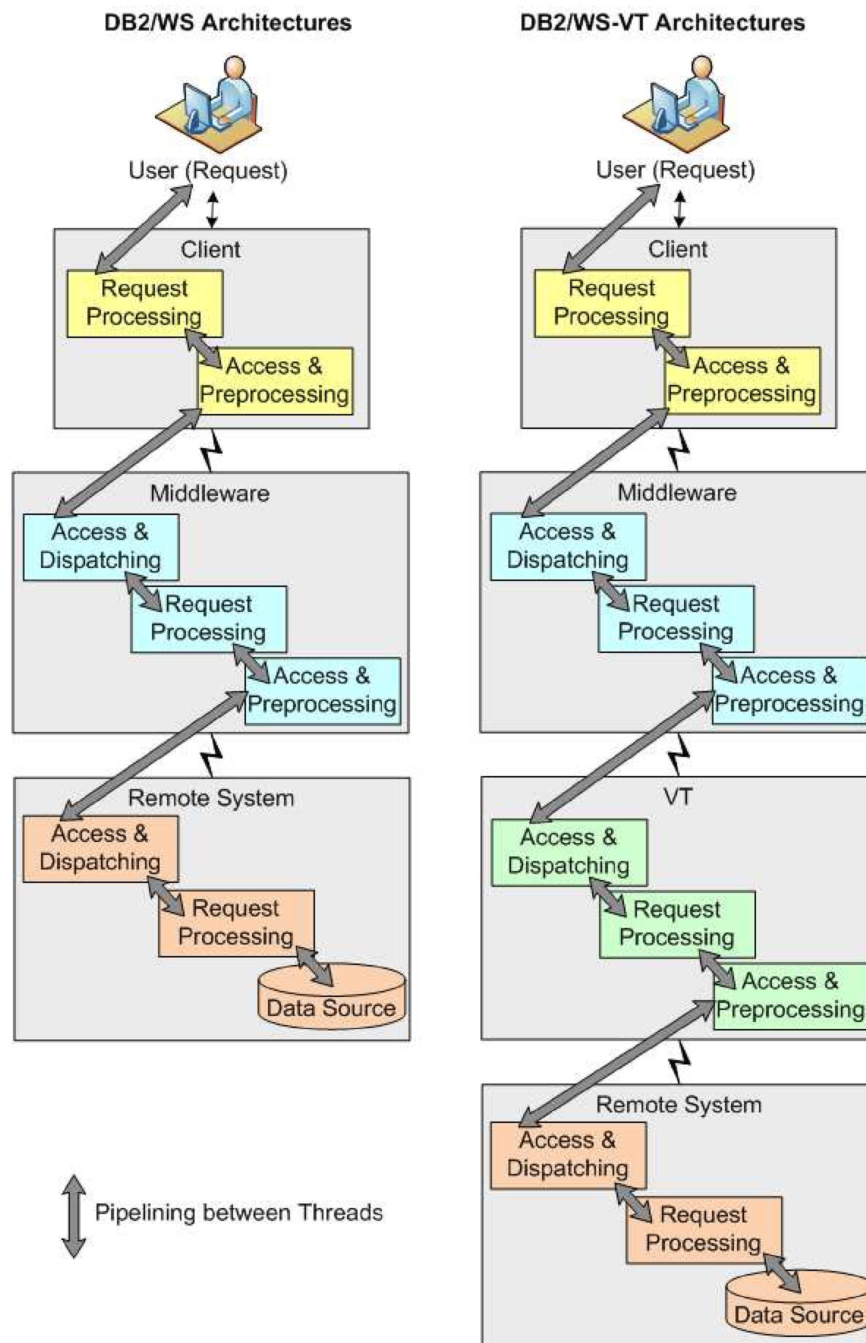


Figure 7.28: Request and Data Pipelining Potential.

time results of the WS case show almost constant VT overhead for the *File* case and an even increasing VT overhead for the *OOApp* case (see Figures 7.40 and 7.41). The main reason for this behavior is that the result data set is transferred from the VT via WebSphere to the WebSphere client although it is larger than both originating data sets together. The WS case without the VT actually transfers both originating data sets so that the WebSphere client has to compensate for the

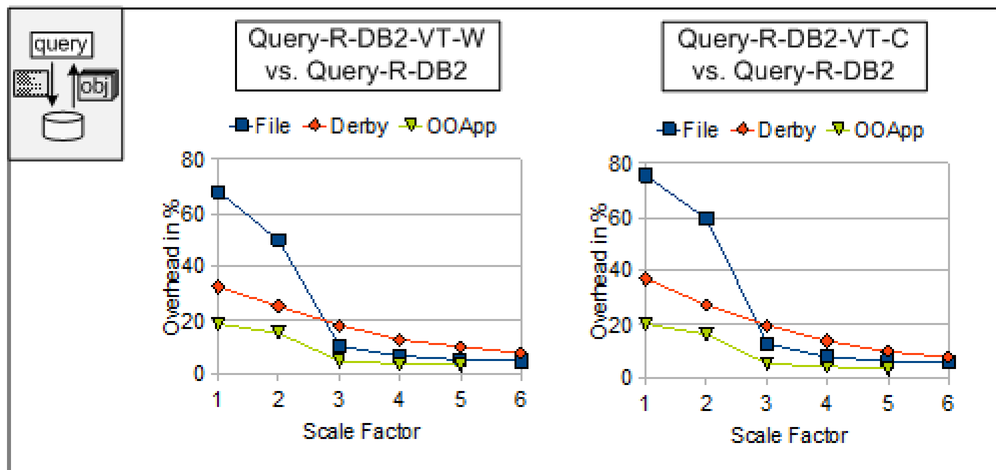


Figure 7.29: Query-R-DB2.

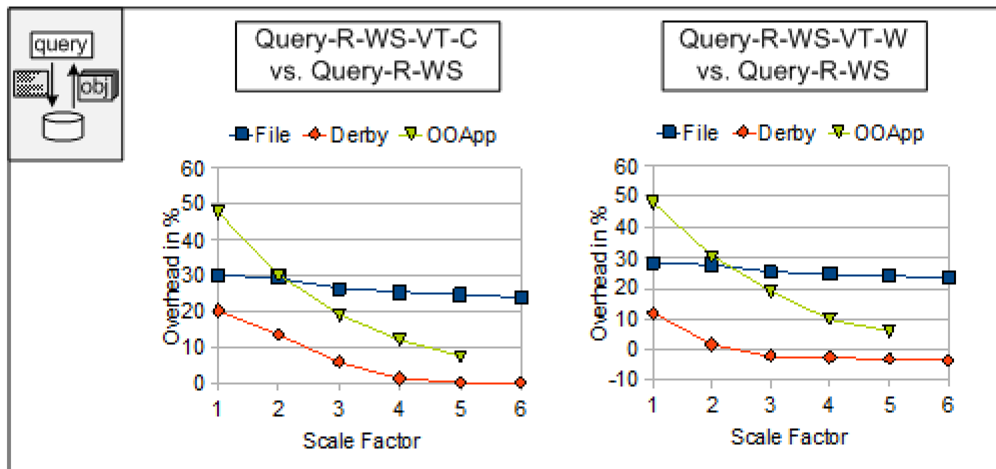


Figure 7.30: Query-R-WS.

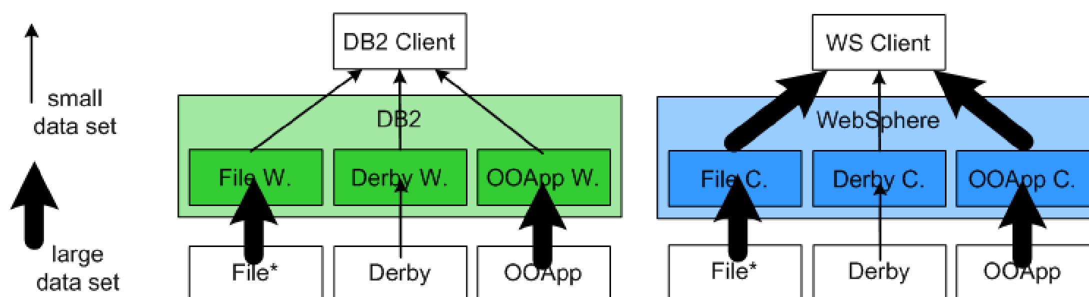


Figure 7.31: Query-SP-DB2/WS (query processing in DB2 and in the WS client).

join. But this time it is beneficial to perform the join as late as possible to reduce the amount of data that has to be transferred (see Figures 7.42 and 7.43). The execution time for the VT-compensated case partially is higher or even much higher

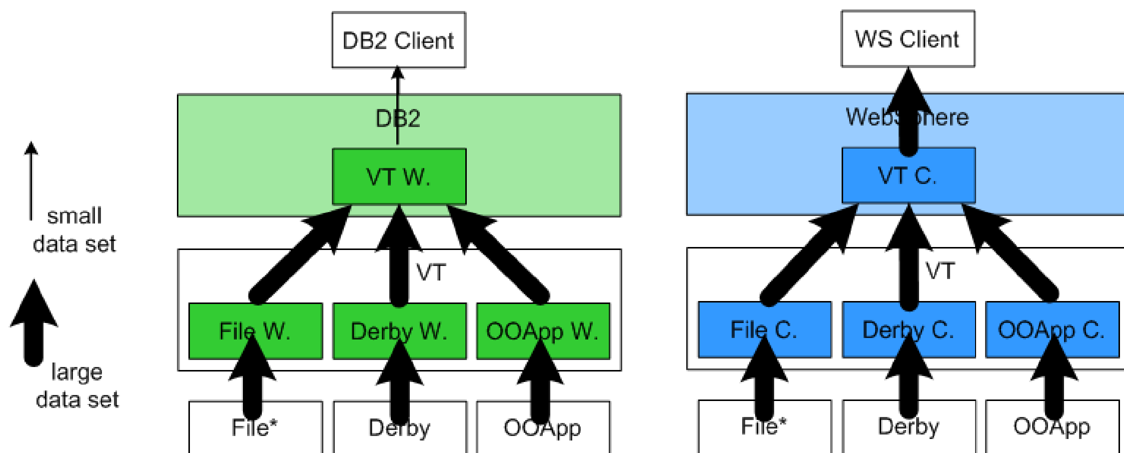


Figure 7.32: Query-SP-DB2/WS-VT (query processing in DB2 and in the WS client).

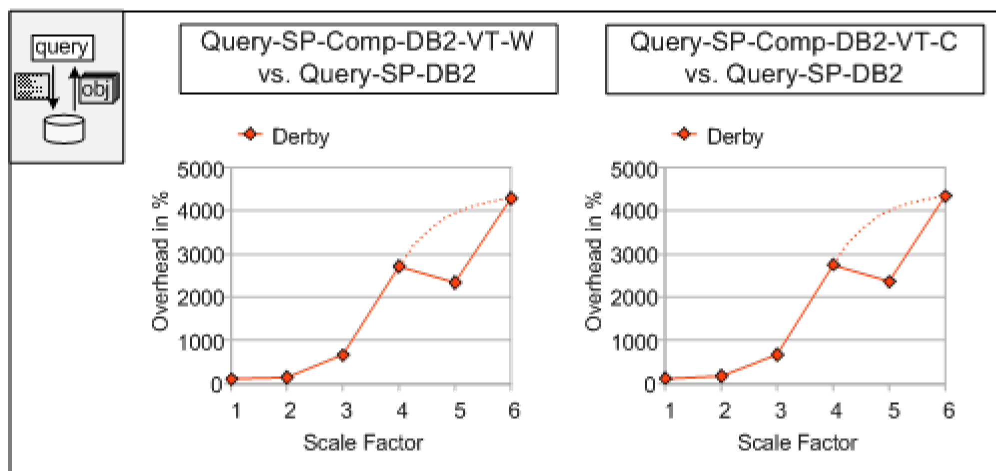


Figure 7.33: Query-SP-DB2-Derby (query processing in DB2).

than for the other cases. However, some of the experiments in the lower scale levels still result in lower execution times when the queries are processed in the VT. This ambivalent behavior clearly indicates that straight-forward push-down can yield suboptimal query execution and therefore the VT does not only need complex query processing capabilities but also sophisticated query evaluation and query optimization capabilities analogous to FDBSs that can optimize cross-query execution and that can negotiate query execution with remote systems.

In summary, queries that return small data sets introduce an overhead of up to 100 percent in the worst case. But, again, this is not a problem since response times of requests of these scale levels range from 10 ms to 200 ms which usually is of no concern. Queries that access larger data sets reduce the overhead to 10 to 20 percent or even lower which is a really acceptable rate. The most important aspect is that the VT needs capabilities to properly optimize and execute complex queries and it also needs capabilities to negotiate and decide about push-down operations and cross-

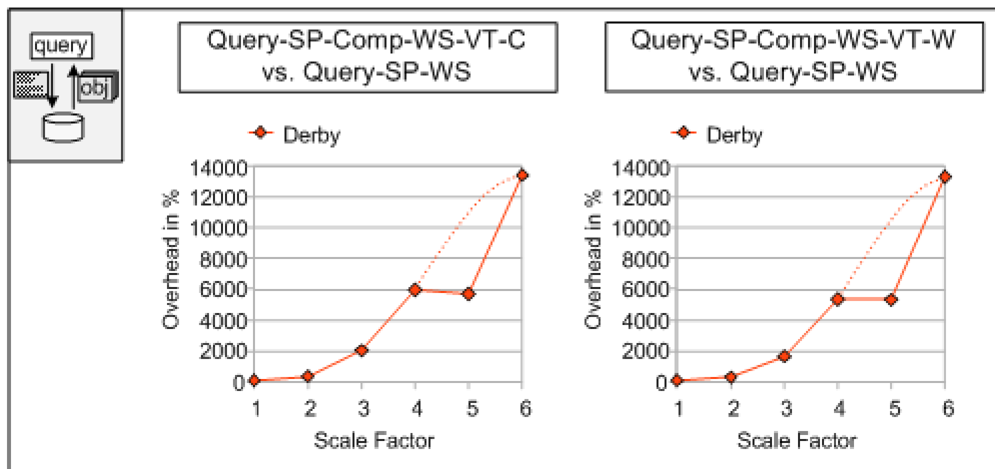


Figure 7.34: Query-SP-WS-Derby (query processing in the WS client).

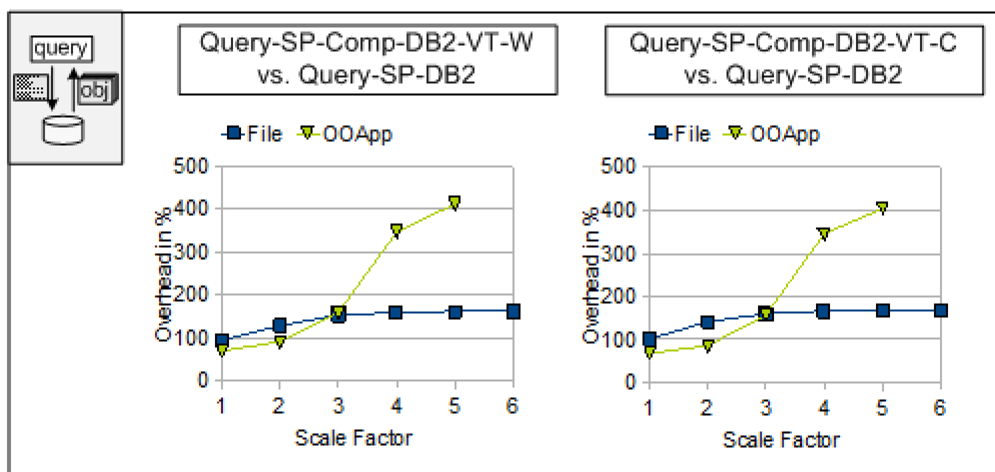


Figure 7.35: Query-SP-DB2-File/OOApp (query processing in DB2).

query optimization [HKWY97, LPL96]. If the VT incorporates such capabilities it is even able to improve execution time of existing integration scenarios since middleware approaches such as EAI or Web services usually do not offer complex query execution capabilities.

7.5 Optimization Effects

The unexpected behavior of the experiments prevailed on us to check the VT prototype and the other systems of the integration scenario and to thoroughly perform and evaluate the experiments a second time. The integration scenario worked correctly and the experiment execution time results indeed remained the same. We further investigated the unexpected behavior, which comprised local monitoring of single, isolated systems of the infrastructure, e.g. local execution of parts of the

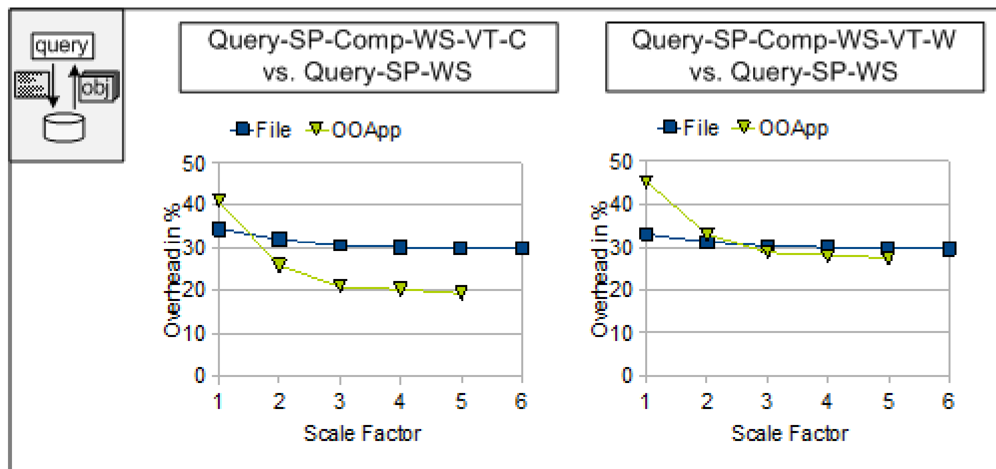


Figure 7.36: Query-SP-WS-File/OOApp (query processing in the WS client).

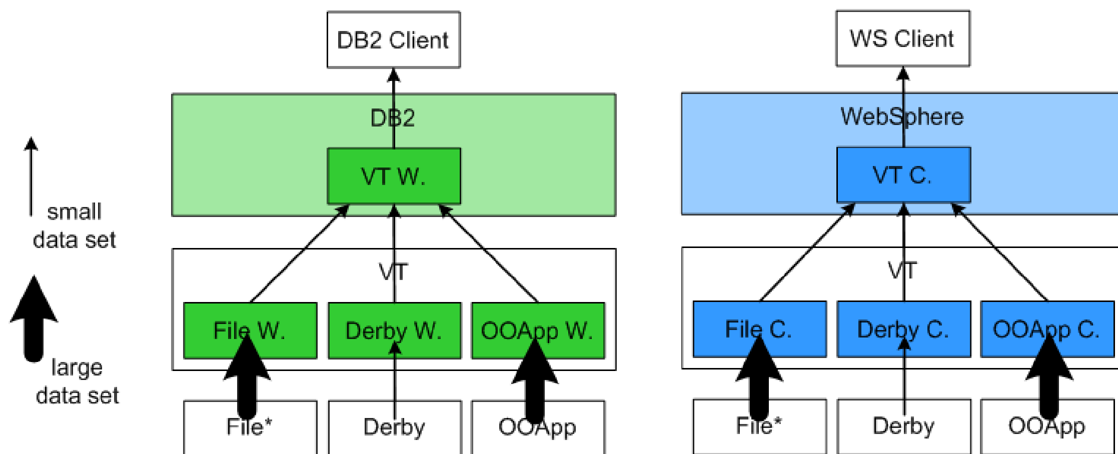


Figure 7.37: Query-SP-DB2/WS-VT (query processing in the VT).

experiments, JVM garbage collection behavior or JVM profiling. However, these investigations detected deviations from the expected behavior only in some cases. The majority of the cases with unexpected behavior cannot be solely explained by means of a single, isolated system of the infrastructure, but originates from the complexity of the whole infrastructure. The unexpected behavior comprises unexpected leaps in the *Read* and *Write* categories as well as unexpected, constant behavior in the *Read* and *Write* categories (see Sections 7.4.2 and 7.4.3). There also is an unexpected behavior in the *Query* category as shown in Figures 7.33 and 7.34. The reason here is the Derby server that significantly decreases performance for queries with scale level 5 and with a high selectivity factor (see Figure 7.44). We detected the same behavior for a selection-projection-join query of the form `SELECT c1, c2, c3, c4 FROM x,y WHERE x.c1 = y.c3 AND c4 = v1`, which selects one percent of the join result, i.e. *lineitem* and *orders* data for the *File* case and for the *Derby* case and *Employee* and *Department* data for the *OOApp* case, and which additionally

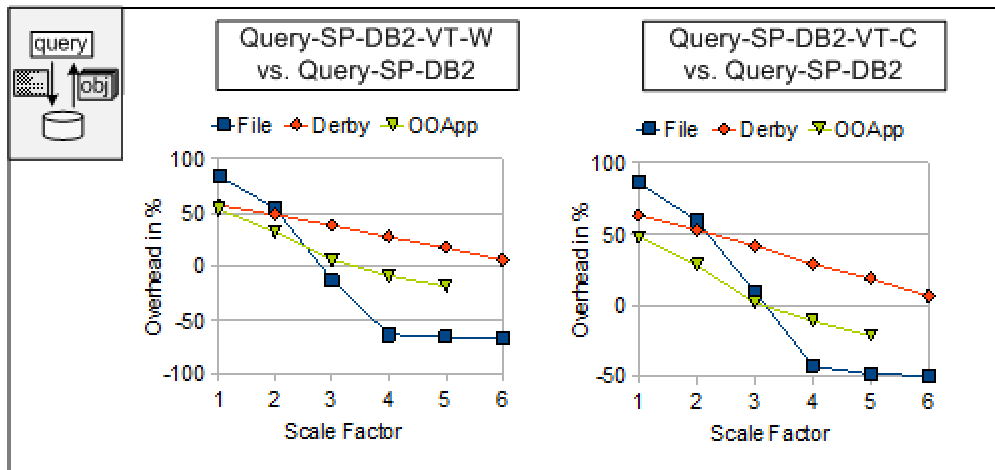


Figure 7.38: Query-SP-DB2 (query processing in the VT).

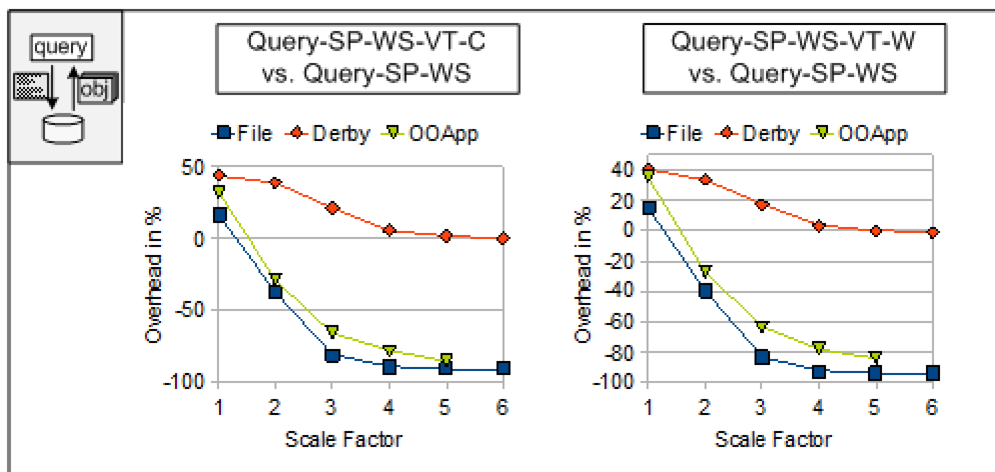


Figure 7.39: Query-SP-WS (query processing in the VT).

reduces the returned data amount to less than one percent due to the projection to a few columns (see Figures 7.45 and 7.46). We completely recreated the database for this scale level, but the performance decrease still remained. The query plans also showed no differences to the query plans of the other scale levels. The executing JVMs showed no conspicuous garbage collection activities and the JVM profiling tool did not give any useful hints, either. In the end, the deviation of the query execution time results from the expected behavior is locally caused by the Derby server and not by the VT. Therefore, this behavior is not relevant for an assessment of the VT. The rest of the unexpected behavior, however, cannot be reduced to local deviations only, but has more complex reasons.

Generally, the unexpected behavior shows better performance values than expected. The unexpected behavior in the *Read* and *Write* categories shows up to 28 percent faster executions when a VT-based scenario is used (e.g. see Figure 7.11), whereas the expected performance decrease in the *Query* category is up to 13,000

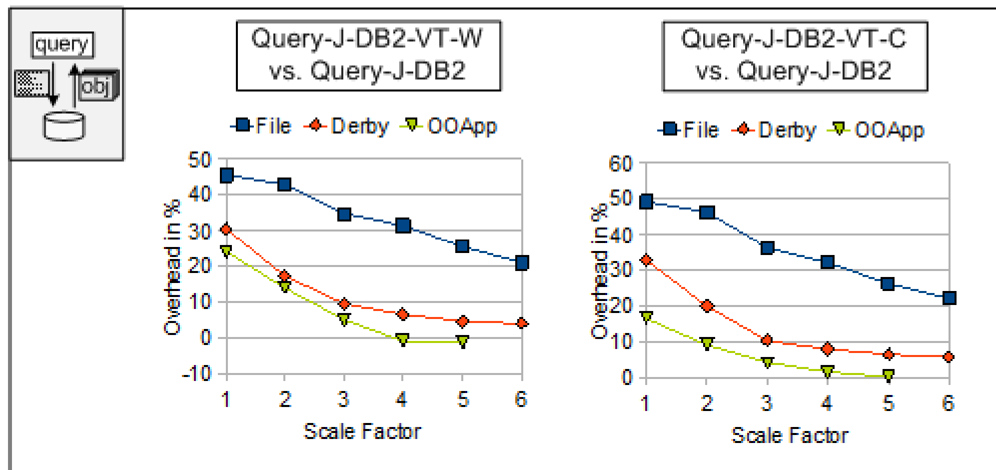


Figure 7.40: Query-J-DB2 (query push-down).

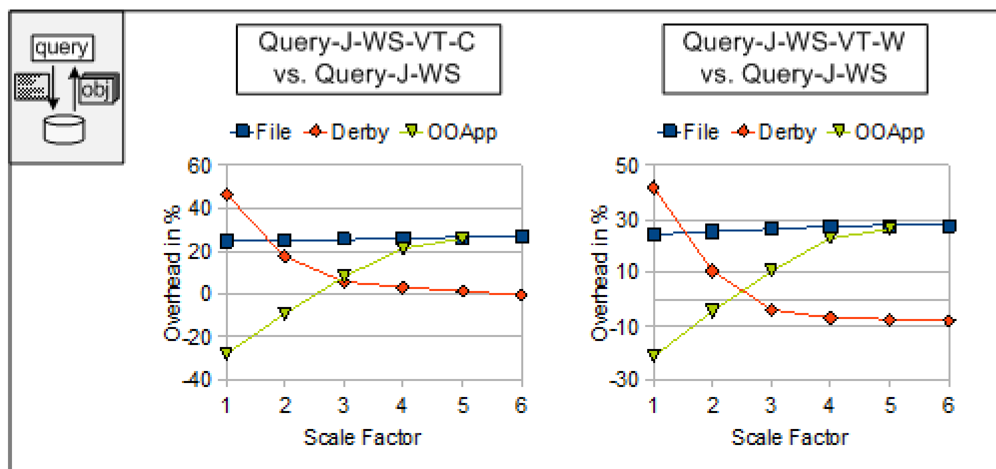


Figure 7.41: Query-J-WS (query push-down).

percent if the VT does not support sophisticated query processing (e.g. see Figure 7.34). The VT-based scenarios in the *Query* category basically change the way a query is executed in the infrastructure and therefore they partially yield significant performance differences to the conventional, non-VT scenarios. This is not the case for the unexpected behavior where the performance differences between the expected behavior and the actual behavior are much less significant as for the query execution in the *Query* category. The reason is that the effects that cause the unexpected behavior do not come from basic changes in the way the infrastructure executes requests, but they originate from the correlations and mutual dependencies of the components of the infrastructure so that a sub-optimal execution of the infrastructure becomes a little bit less sup-optimal. In that way, the unexpected behavior works as an optimization effect, even if it is an unexpected and unintentional one. The leaps are a very good example for that phenomenon as they show a non-linear increase of the execution time either for both compared scenarios or

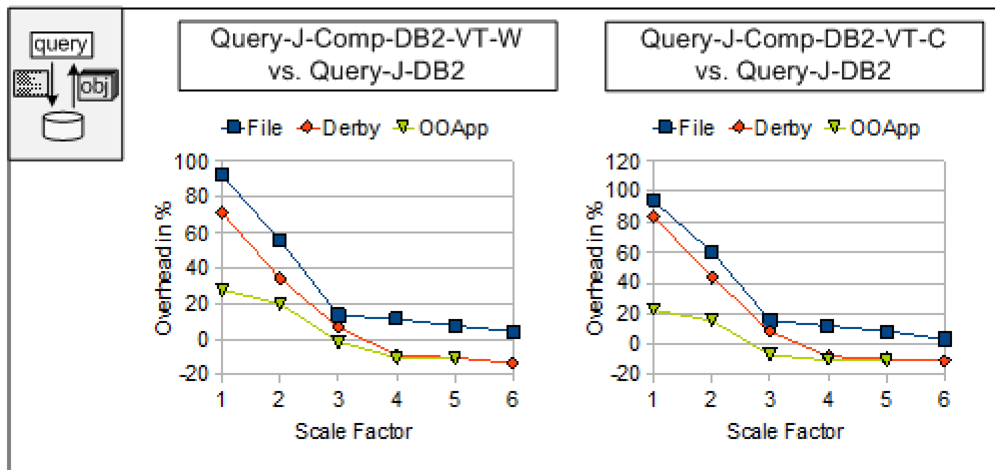


Figure 7.42: Query-J-DB2 (query compensation in DB2).

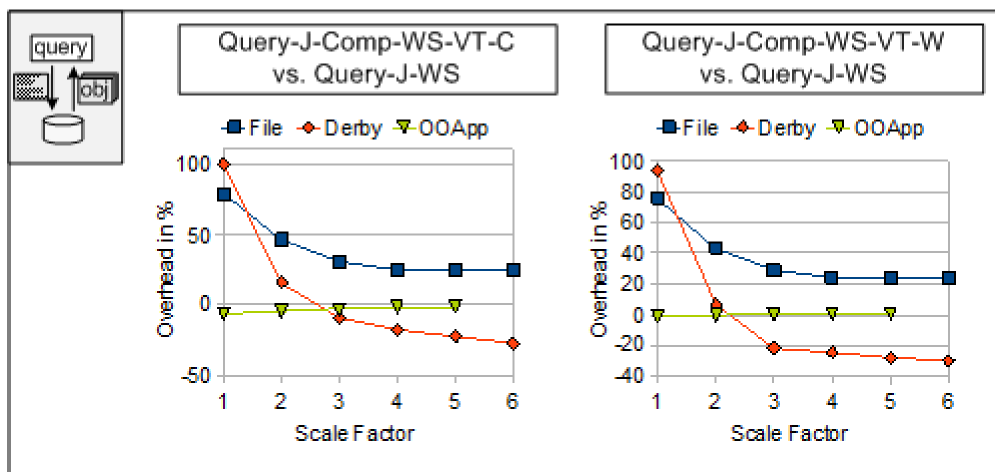


Figure 7.43: Query-J-WS (query compensation in the WS client).

for one of both, i.e. VT-based scenario or conventional, non-VT scenario. These leaps, i.e. the non-linear increase of execution time, indicate that the infrastructure suddenly starts stuttering, i.e. the execution becomes less optimal for some reason so that performance finally decreases. An isolated execution and evaluation of the single, participating systems showed similar leap behavior only in some cases and did not show any conspicuous characteristics in the other cases at all because the leaps and the other effects in the experiments are mainly caused by the interactions and correlations within the infrastructure, which means that they originate from the complexity of the whole infrastructure: four different layers with different software systems on each layer, different hardware, operating systems, programming languages, multiple network connections and different ways of communicating via the network, i.e. request, response, transfer of data. Performance issues in such a complex scenario also depend on the whole infrastructure and cannot always be divided up into clearly separated parts and statements such as "component x con-

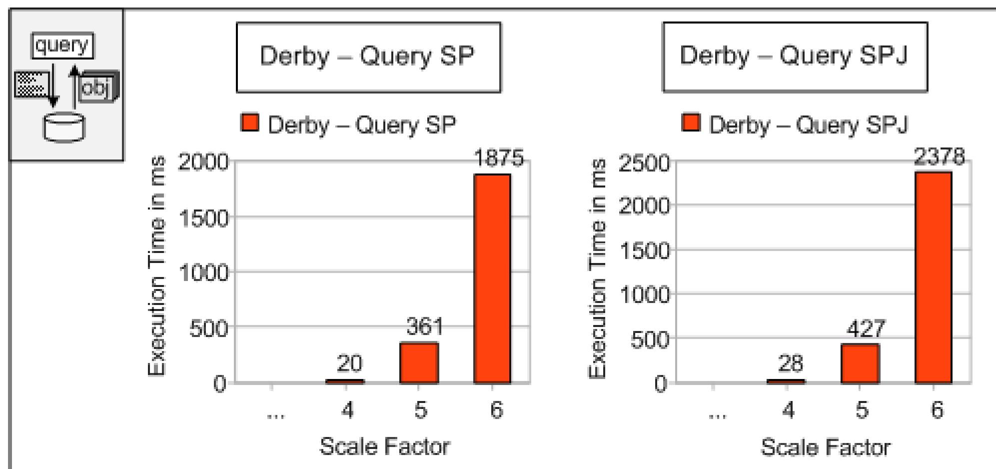


Figure 7.44: Degeneration Effect: Querying Derby directly.

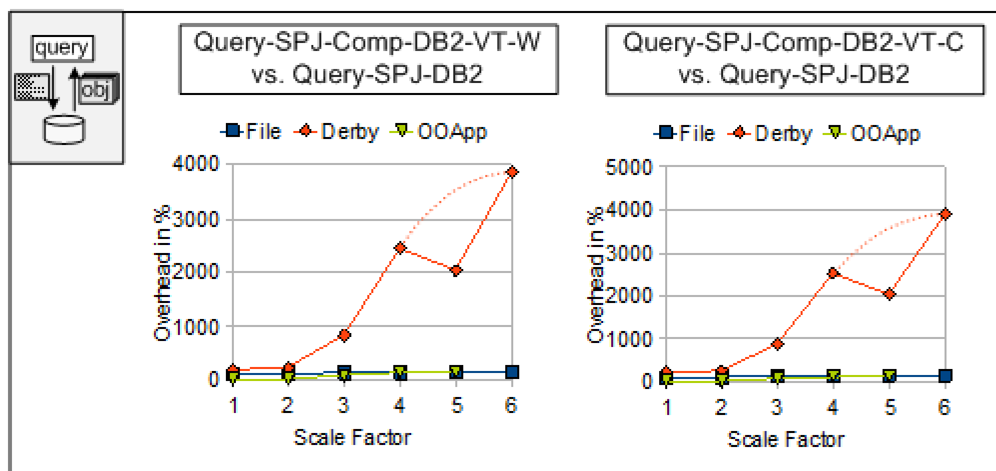


Figure 7.45: Query-SPJ-DB2: Derby Degeneration Effect.

sumes exactly y ms of the overall request execution time of z ms". It is impossible to completely assess a single, isolated part of the infrastructure separately without the context of the surrounding infrastructure. Adding an additional component such as the VT to an infrastructure does not necessarily decrease the performance of the infrastructure according to the performance characteristics of the added component. Put in other words, the infrastructure of integration scenarios typically is more complex and more meaningful than the sum of its parts.

Logically, the next step is to monitor the participating systems not isolated, but in the context of the whole infrastructure. The problem is that we have to instrument and monitor all participating systems, i.e. operating system, JVM, middleware system, network communication, thread time, etc., to reliably detect the interactions and dependencies during the processing of a request. However, instrumenting and monitoring all participating systems inherently influences the actual execution of the infrastructure and thereby distort the "original" request execution

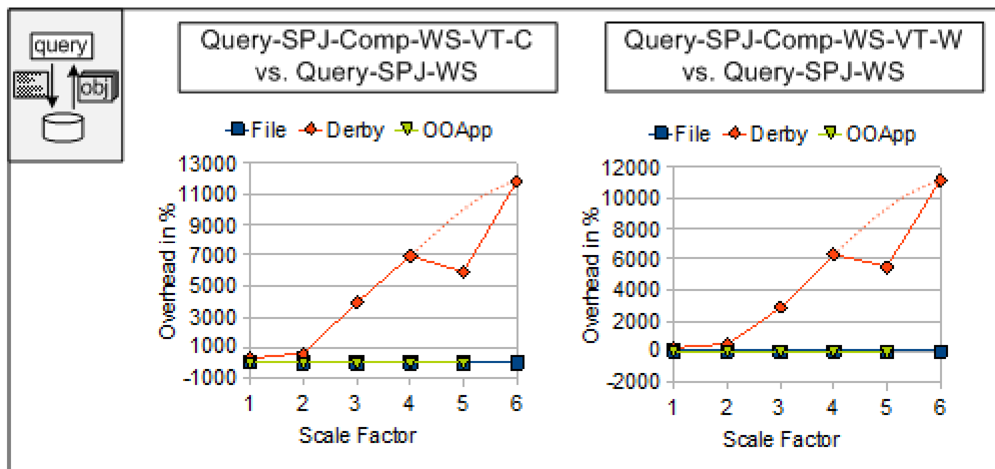


Figure 7.46: Query-SPJ-WS: Derby Degeneration Effect.

(think of quanta in quantum theory that are changed when they are observed by a third party). Consequently, the "altered" execution differs in most cases from the "original" execution in a way that effects that formerly occurred no longer occur or other effects occur that did not occur before. This in turn makes it difficult, if not impossible to determine the exact reasons and causes of the effects to be investigated. For example, the JVM profiling facility that we used to monitor the execution of our systems and adapters significantly slowed down performance due to the costly task of instrumenting the code to be monitored and executing the instrumented code. This did not only significantly increase the overall request execution time, but especially significantly influenced the interactions and dependencies of the participating systems so that the effects of the previously detected unexpected behavior did not come into play and thereby the unexpected behavior did not occur.

It would be very interesting to apply a much more extensive and sophisticated diagnostics and monitoring approach in the next step to determine the reasons and causes of the unexpected behavior in more detail. However, such an in-depth investigation is not relevant for the results of this work and it would also go far beyond the scope of this work. An investigation and a systematic analysis and evaluation of this class of phenomena, e.g. how optimization in integration scenarios can be achieved, what does it cost to optimize the processing in integration scenarios, etc. is dedicated to future work. Nevertheless, we give a brief overview why the VT can actually decrease execution time in the experiment scenarios and how the unexpected optimization effects look like. A major issue are concurring, blocking or overstrained resources such as main memory, hard disk, processor or communication that are better managed or exploited when the VT is additionally used. For example, the footprint of the VT prototype is smaller than the footprint of DB2 or WebSphere. The VT host has the same hardware and configuration than the middleware host and thus the JVM on the VT host has more main memory to hold the data of the remote systems, to marshal and unmarshal data, etc. than the JVMs of DB2

or WebSphere. Therefore, the VT host can more efficiently handle high request rates or queries with larger result sets. The *Read* and *Write* categories show this behavior for the *Derby* case and for the *OOApp* case, e.g. reading or writing 1,000 or more data items per request. The *File* case indicates another cause. DB2 and WebSphere have to read and write data from and to files in the *File* case. If they also concurrently read or write other data from or to files, access to the hard disk can be more often or longer required than if the VT is reading or writing files on the VT host and if DB2 or WebSphere read configurations, log information, reorganize disk space, etc. on the middleware host. In that case, concurring access to the hard disk is distributed to access to hard disks on two hosts, i.e. the VT host and the middleware host, and thereby the whole request is sped up. Another point are waiting situations. If DB2 and WebSphere read data from files in the *File* case, they request the next data item when it is needed, i.e. each data item has to be retrieved from hard disk on demand as there is no sophisticated prefetching mechanism. If however DB2 and WebSphere access the VT and the VT reads the files, the VT is reading data from files concurrently to DB2 or WebSphere that are processing the data items requested from the VT. Thereby, the VT can speed up request execution and may even be faster than if the same request is executed without the VT as given by the *File* case in the *Read* category. The bottom-line of the unexpected behavior is that we partially perform unintended optimizations in integration scenarios when we use the VT. The infrastructure at least partially behaves sub-optimally so that optimization considerations might be an issue and worth further investigations.

7.6 Summary

The benefits of the VT such as software reuse, integration independence, higher stability and flexibility of IT infrastructures, reduced costs for software, hardware and personnel come at some costs, i.e. partial performance decrease. However, the performance decrease is moderate and even negligible when we consider the benefits of the VT. The main result of the experiments is that the longer a request execution lasts and the larger the data sets are that are read or written, the better performs the VT, i.e. the lower is the overhead. If request executions have a very short execution time or if very small data sets are read or written, the VT creates some overhead. However, this can be usually neglected in integration scenarios since the overhead is only up to a few hundred ms in the worst cases. The overall result of the experiments is that the VT actually is practically applicable and efficient. Another important result is that the VT can efficiently handle diverse adapter technologies independent of the context where it is used.

The experiments revealed unexpected behavior that yielded optimization effects and thereby better performance values than expected. The effects that caused the unexpected behavior in the *Read* and *Write* categories did not come from basic changes in the way the infrastructure executed requests, but they originated from the correlations and mutual dependencies of the infrastructure components so that a

sub-optimal execution of the infrastructure became less sup-optimal. The VT-based scenarios in the *Query* category basically changed the way a query was executed in the infrastructure by means of a different query optimization strategy and therefore they partially yielded significant performance improvements to the conventional, non-VT scenarios.

Chapter 8

Conclusion

There is an abundance of integration technologies and there still is no means to systematically deal with them. This leads to increasing complexity in IT environments and also to increasing integration efforts and IT costs. *Integration management* (IM) provides a means of systematically dealing with integration technologies. It abstracts from integration technologies so that software development is shielded from integration tasks. The achieved integration independence significantly alleviates maintenance and evolution of IT environments and reduces the overall complexity and costs of IT landscapes.

We designed and realized an IM system, the *virtualization tier* (VT), that allows to reuse adapters of different integration technologies so that costly development of new adapters from scratch can be avoided. The VT achieves integration independence by means of a global access layer that supports transparent processing independent of the access style chosen by a client system and independent of the integration technology used in the VT to access a remote system. A client system can access any remote system by means of the VT if there is a suitable adapter deployed in the VT. The integration tasks that are related with accessing remote systems can be completely encapsulated by means of the VT so that a software developer can concentrate on the development of the core application logic.

Deployment and administration tasks in the VT are handled by different IT roles, i.e. *VT object deployer* and *adapter deployer*, so that the complex use of integration technologies becomes practically manageable. The VT also enables different architecture patterns to realize systematic integration solutions based on IM technology. The VT-based architecture patterns reuse parts of integration technologies and provide for significantly less complex integration solutions than conventional integration solutions do. Even SOA-based applications can benefit from the VT by using the VT as a global ESB. IM technology can thereby become a central part of Web service infrastructures. Finally, our performance evaluation of the VT shows that IM technology can work efficiently and that optimization effects can even improve performance of IT scenarios. The moderate performance decrease that is partially introduced by the VT layer becomes negligible when we consider the benefits of the VT such as integration independence, higher stability and flexibility of IT

infrastructures, reduced costs for software, hardware and personnel. Especially the reuse of adapters in the VT does not only avoid costly and lengthy development of new adapters, but also employs existing adapters in the sense of IT infrastructure assets, i.e. investments, that pay off much better. Moreover, the reuse of adapters in the VT allows IT infrastructures to rely on existing, stable adapters instead of new ones that have been recently developed and that first have to become mature and stable. Existing adapters may be the only means that is left to adequately and correctly access legacy remote systems that would be out of control and no longer usable otherwise. This means that adapter reuse may be the only way to economically and effectively deal with remote systems no matter how efficient this works. And the VT prototype actually shows that such cases can still work efficiently.

In summary, IM technology is a key technology to cope with integration issues and IT infrastructure heterogeneities in a systematic and uniform manner.

Bibliography

[ABCB+01] Danny Ayers, John Bell, Carl Calvert-Bettis, Thomas Bishop, Bjarki Holm, Glenn E. Mitchell, Kelly Lin Poon, and Sean Rhody. *Professional Java Data*. Wrox Press, 2001.

[APvS03] J.P.A. Almeida, L.F. Pires, and M.J. van Sinderen. Web services and seamless interoperability. *First European Workshop on Object Orientation and Web Services*, 2003.

[AQM+97] Serge Abiteboul, Dallan Quass, Jason Mchugh, Jennifer Widom, and Janet Wiener. The lorel query language for semistructured data. *International Journal on Digital Libraries*, 1:68–88, 1997.

[BB97] Ph. Bonnet and S. Bressan. Extraction and integration of data from semi-structured documents into business applications. In *Intl. Conf. on Industrial Applications of Prolog..1997*, 1997.

[BBH+08] M. Böhm, J. Bittner, D. Habich, W. Lehner, and U. Wloka. Model-Driven Generation of Dynamic Adapters for Integration Platforms. In J.P. Bourey, X. Franch, and E. Hunt, editors, *Proceedings of the 1st International Workshop on Model Driven Interoperability for Sustainable Information Systems (MDISIS'08)*, pages 105–119, June 2008.

[BCG+05] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In *Advanced Information Systems Engineering*, pages 415–429, 2005.

[BCMP13] Mirko Bronzi, Valter Crescenzi, Paolo Merialdo, and Paolo Papotti. Extraction and integration of partially overlapping web sources. *Proc. VLDB Endow.*, 6(10):805–816, August 2013.

[BHM+04] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard, editors. *Web Services Architecture*. World Wide Web Consortium, February 2004. W3C Working Group Note.

[CBB+00] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando

Velez, editors. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[CGMH+94] Sudarshan Chawathe, Hector Garcia-Molina, Joachim Hammer, Kelly Ireland, Yannis Papakonstantinou, Jeffrey D. Ullman, and Jennifer Widom. The TSIMMIS Project: Integration of Heterogeneous Information Sources. In *16th Meeting of the Information Processing Society of Japan*, pages 7–18, 1994.

[CHS+95] Michael J. Carey, Laura M. Haas, Peter M. Schwarz, Manish Arya, William F. Cody, Ronald Fagin, John Thomas, John H, and Edward L. Wimmers. Towards heterogeneous multimedia information systems: The garlic approach. In *In RIDE-DOM*, pages 124–131, 1995.

[CMRW07] Roberto Chinnici, Jean-Jacques Moreau, Arthur Ryman, and Sanjiva Weerawarana, editors. *Web Services Description Language (WSDL)*. World Wide Web Consortium, 2.0 edition, June 2007. W3C Recommendation.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[DHW01] Denise Draper, Alon Y. HaLevy, and Daniel S. Weld. The nimble xml data integration system. In *Proceedings of the 17th International Conference on Data Engineering*, pages 155–160. IEEE Computer Society, 2001.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[FT96] William Frakes and Carol Terry. Software reuse: Metrics and models. *ACM Comput. Surv.*, 28(2):415–435, June 1996.

[GAD+05] Ian Gorton, Justin Almquist, Kevin Dorow, Peng Gong, and Dave Thurman. An architecture for dynamic data source integration. In *Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Volume 09*, pages 276.3–, Washington, DC, USA, 2005. IEEE Computer Society.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *Proceedings of the 17th International Conference on Software Engineering*, ICSE '95, pages 179–185, New York, NY, USA, 1995. ACM.

[GAO09] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is still so hard. *IEEE Software*, 26(4):66–69, 2009.

[GGF+95] Georges Gardarin, Sofiane Gannouni, Béatrice Finance, Peter Fankhauser, Wolfgang klas, Dominique Pastre, Régis Legoff, Antonis Ramfos, In Omran Bukhres, and Ahmed K. Elmagarmid (eds. Irodb - a distributed system federating object and relational databases. In *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications, chapter 20*, pages 684–712. Prentice Hall, 1995.

[GRVB98] Jean-Robert Gruser, Louiqa Raschid, María Esther Vidal, and Laura Bright. Wrapper generation for web accessible data sources. In *In CoopIS*, pages 14–23, 1998.

[Hai07] Marc N. Haines. The impact of service-oriented application development on software development methodology. *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 172b–172b, Jan. 2007.

[HGMN+97] Joachim Hammer, Hector Garcia-Molina, Svetlozar Nestorov, Ramana Yerneni, Marcus Breunig, and Vasilis Vassalos. Template-Based Wrappers in the TSIMMIS System. In *SIGMOD '97*, pages 532–535, 1997.

[HKWY97] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In *In Proc. of VLDB*, pages 276–285, 1997.

[HMKH10] Laura M. Haas, Renée J. Miller, Donald Kossmann, and Martin Hentschel. A first step towards integration independence. In *ICDE Workshops*, pages 147–150, 2010.

[KBA+10] Gavin King, Christian Bauer, Max Rydahl Andersen, Emmanuel Bernard, Steve Ebersole, and Hardy Ferentschik. *Hibernate Reference Documentation, 3.6.0.* 2010.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput Surv.*, 24(2):131–183, 1992.

[LCL06] Siew Poh Lee, Lai Peng Chan, and Eng Wah Lee. Web services implementation methodology for soa application. *Industrial Informatics, 2006 IEEE International Conference on*, pages 335–340, Aug. 2006.

[LDA06] Lightweight Directory Access Protocol, June 2006.

[Lee05] Eng Wah Lee, editor. *Web Service Implementation Methodology.* Organization for the Advancement of Structured Information Standards (OASIS), July 2005. Public Review Draft.

[Lev98] Alon Y. Levy. The Information Manifold Approach to Data Integration. *IEEE Intelligent Systems*, 13(5):12–16, September/October 1998.

[LHB+99] Ling Liu, Wei Han, David Buttler, Calton Pu, and Wei Tang. An XML-based Wrapper Generator for Web Information Extraction. In *In Proc.of ACM-SIGMOD 99*, pages 540–543, 1999.

[LPH00] Ling Liu, Calton Pu, and Wei Han. Xwrap: An xml-enabled wrapper construction system for web information sources. In *Proceedings of the 16th International Conference on Data Engineering*, pages 611–621. IEEE Computer Society, 2000.

[LPL96] Ling Liu, Calton Pu, and Yooshin Lee. An adaptive approach to query mediation across heterogeneous information sources. *Cooperative Information Systems, IFCIS International Conference on*, 0:144, 1996.

[LR00] Frank Leymann and Dieter Roller. *Production Workflow: Concepts and Techniques*. Prentice Hall, 2000.

[McG06] David McGoveran. Embracing SOA: The Benefits of Integration Independence. *White paper, Alternative Technologies*, 2006.

[Mic10] *.NET Framework*. Microsoft, April 2010.

[Mil10] Renée J. Miller. Information integration: a vision for integration independence and linking open data. In *AMW*, 2010.

[Noe05] Jasmine Noel. BPM and SOA: Better together. *White paper, IBM*, 2005.

[Noy04] Natalya F. Noy. Semantic integration: A survey of ontology-based approaches. *SIGMOD Record*, 33:2004, 2004.

[OMG08] OMG. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1.* Object Management Group, 2008.

[PH07] Mike P. Papazoglou and Willem-Jan Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB Journal*, 16(3):389–415, 2007.

[PRÁ+02] Alberto Pan, Juan Raposo, Manuel Álvarez, Paula Montoto, Vicente Orjales, Justo Hidalgo Luca Ardao, Anastasio Molano, and Ángel Viña. The Denodo Data Integration Platform. In *Proceedings of the 28th international conference on Very Large Data Bases*, VLDB '02, pages 986–989. VLDB Endowment, 2002.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB JOURNAL*, 10:2001, 2001.

[RPÁ+02] Juan Raposo, Alberto Pan, Manuel Álvarez, Justo Hidalgo, and Ángel Viña. The Wargo System: Semi-Automatic Wrapper Generation in Presence of Complex Data Access Modes. In *DEXA '02*, pages 313–317, 2002.

[RS97] Mary Tork Roth and Peter M. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *VLDB '97*, pages 266–275, 1997.

[SQL08] ISO/IEC 9075:2008 Information technology – Database languages – SQL, 2008.

[Sto02] Michael Stonebraker. Too much middleware. *SIGMOD Rec.*, 31(1):97–106, 2002.

[Sun02] Sun. *Java Message Service.* Sun Microsystems Inc., April 2002. Final Release.

[Sun03] Sun. *J2EE Connector Architecture Specification, Version 1.5.* Sun Microsystems Inc., November 2003. Final Release.

[Sun05] Sun. *Java Business Integration(JBI) 1.0.* Sun Microsystems Inc., August 2005. Final Release.

[Sun06a] Sun. *Java EE Specification, Version 5.* Sun Microsystems Inc., April 2006. Final Release.

[Sun06b] Sun. *JDBC 4.0 Specification.* Sun Microsystems Inc., November 2006. Final Release.

[Sun10] Sun. *Java Data Objects 3.0.* Sun Microsystems Inc., 2010.

[SvdAB+03] S. Staab, W. van der Aalst, V.R. Benjamins, A. Sheth, J.A. Miller, C. Bussler, A. Maedche, D. Fensel, and D. Gannon. Web services: been there, done that? *Intelligent Systems, IEEE*, 18(1):72–85, Jan-Feb 2003.

[Vin03] Steve Vinoski. Integration with web services. *IEEE Internet Computing*, 7:75–77, 2003.

[Whi75] J.E. White. RFC 707: A High-Level Framework for Network-Based Resource Sharing, 1975.

[WM07a] Ralf Wagner and Bernhard Mitschang. Enhancing middleware functionality by virtualizing adapters. In *ICEIS (Selected Papers)*, pages 108–120, 2007.

[WM07b] Ralf Wagner and Bernhard Mitschang. Flexible reuse of middleware infrastructures in heterogeneous it environments. In *OTM Conferences (1)*, pages 522–539, 2007.

[WM07c] Ralf Wagner and Bernhard Mitschang. A methodology and guide for effective reuse in integration architectures for enterprise applications. In *IRI*, pages 323–328, 2007.

[WM07d] Ralf Wagner and Bernhard Mitschang. A virtualization approach for reusing middleware adapters. In *ICEIS (1)*, pages 78–85, 2007.

[WM09] Ralf Wagner and Bernhard Mitschang. Uniform and efficient data provisioning for soa-based information systems. In *ITNG*, pages 1012–1017, 2009.