Informatik – Forschung und Entwicklung manuscript No.
(will be inserted by the editor)

Ankopplung heterogener Anwendungssysteme an Föderierte Datenbanksysteme durch Funktionsintegration

Theo Härder[1], **Klaudia Hergula**[2]

[1] Universität Kaiserslautern, Fachbereich Informatik, AG DBIS, Postfach 3049, 67653 Kaiserslautern
[2] DaimlerChrysler AG, IT Management, Datenbanken und Data Warehouse Systeme (TOS/TDW), HPC 096/0516, 70546 Stuttgart

Received: date / Revised version: date

Zusammenfassung Mit der zunehmenden Zahl an Anwendungssystemen, welche Datenbank und zugehörige Anwendung kapseln, ist die reine Datenintegration nicht mehr ausreichend. Stattdessen können die Daten des Anwendungssystems nur noch über vordefinierte Funktionen abgefragt werden. Sollen Daten und Funktionen integriert werden, so ist eine Kombination von deklarativen Anfragen und dem Zugriff auf Funktionen erforderlich. In diesem Aufsatz stellen wir einen Middleware-Ansatz vor, der diese erweiterte und neuartige Form der Integration unterstützt. Hierzu werden zunächst die Probleme der Funktionsintegration erörtert und eine Spezifikationsssprache eingeführt, die föderierte Funktionen zu bilden und diese mehreren lokalen Funktionen der zu integrierenden Anwendungssysteme zuzuordnen erlaubt. Anschließend zeigen wir, wie die zugehörige Ausführungskomponente – in unserem Fall ein Workflow-Managementsystem – an ein Föderiertes Datenbanksystem mittels eines Wrappers gekoppelt wird, um die Integration von Daten als auch Funktionen zu ermöglichen. Dabei wird die zu unterstützende Funktionalität innerhalb des Wrappers sowie deren Auswirkungen auf die Anfrageverarbeitung als auch das zugehörige Kostenmodell untersucht. Abschließend wird ein Eindruck von der Performanz der vorgestellten Architektur vermittelt.

Schlüsselwörter Föderierte Datenbanksysteme, Anfrageverarbeitung, Funktionsintegration, Workflow-Managementsysteme, Wrapper

Abstract With the emergence of so-called application systems which encapsulate databases and related application components, pure data integration using, for example, a federated database system is not possible anymore. Instead, access via predefined functions is the only way to get data from an application system. As a result, retrieval of such heterogeneous and encapsulated data sources needs the combination of generic query as well as predefined function access. In this paper, we present a middleware approach supporting such a novel and extended kind of integration. After a brief characterization of function integration, we introduce a mapping language for the specification of federated functions combining functionality of one or more application systems (local functions) to be integrated. We discuss how the related execution component, a workflow management system, can be connected via a wrapper to a federated database system to enable the integration of data as well as functions. Furthermore, we explore the use of query processing functionality within the wrapper and its effect towards the overall query evaluation as well as related cost model. Finally, we give a first impression of the performance to be expected from our architecture.

Key words Federated database systems, query processing, function integration, workflow management systems, wrapper

CR Subject Classification C.3, D.2.12, H.2.4, H.2.5, H.5.3

1 Motivation

In den meisten Unternehmen liegt heutzutage eine heterogene Systemlandschaft vor. Es werden unterschiedliche Hardware-Komponenten, Netzwerk- und Betriebssysteme, Datenbanksysteme (DBS) sowie Anwendungsprogramme zum Einsatz gebracht, um den Lebenszyklus eines von den Unternehmen hergestellten Produktes informationstechnisch vollständig abzudecken, d. h. von dessen Entwicklung und Produktion bis hin zum Vertrieb. Zur Überwindung der dabei entstehenden Heterogenität der Systeme gibt es vor allem im Bereich der heterogenen Datenbanken bereits seit einigen Jahren zahlreiche Forschungsarbeiten. Im Mittelpunkt steht hierbei die Unterstützung von Interoperabilität zwischen heterogenen Datenbanksystemen. Dabei wurden Konzepte und Prototypen sog. Föderierter Datenbanksysteme

(FDBS) und Multidatenbanksysteme (MDBS) entworfen, um Datenbanken mit unterschiedlichen Datenmodellen und Schemastrukturierungen integrieren zu können. Mittlerweile sind auch kommerzielle Produkte verfügbar, die als Datenbank-Gateways oder Datenbank-Middleware bezeichnet werden [14]. Für die essentiellen Probleme auf dem Gebiet der Datenbankintegration liegen somit inzwischen mächtige Lösungen vor, wenn auch einige Fragen noch offen sind [1,2,3,5,12,19].

Das Bild der heterogenen Datenbanklandschaft beginnt sich jedoch zu ändern. Haben sich die Unternehmen bisher bewusst für ein DBS entschieden und auch das Datenbankschema selbst festgelegt, so kommt es heute immer häufiger vor, dass eine Datenbank innerhalb eines Standardsoftware-Pakets mitgeliefert wird. Dabei wird das DBS sowie das zugehörige Anwendungsprogramm zusammengefasst und nach außen hin lediglich eine Programmierschnittstelle, ein sog. API (*Application Programming Interface*), bereitgestellt. Eine Datenbankschnittstelle ist somit nicht mehr verfügbar. Systeme, die dieses Konzept der Kapselung realisieren, werden im Folgenden *Anwendungssysteme* genannt. Hervorzuhebende Stellvertreter für Anwendungssysteme sind z. B. SAP R/3 [15] oder PDM (Produktdatenmanagement)-Systeme wie Metaphase [17]. Im Falle von SAP können demnach die darin verwalteten Daten nicht direkt mittels SQL aus der relationalen Datenbank ausgelesen werden. Stattdessen stellt SAP sog. BAPIs (*Business APIs*) zur Verfügung, die dem Benutzer den Datenzugang über vordefinierte Funktionen erlauben. Neben diesen kommerziellen Produkten gibt es häufig proprietäre, von den Unternehmen selbst implementierte Software-Lösungen, die ausschließlich über ein API zugänglich sind. Dies ist in der Tatsache begründet, dass die Einhaltung von Integritätsbedingungen als auch die Überwachung der Sicherheit (Autorisierung) in vielen Fällen im Anwendungsprogramm realisiert und nicht durch das DBS unterstützt werden. Überdies mangelt es oft an konzeptionellen Schemata und somit an ausdrucksstarken Bezeichnern. Um zu vermeiden, dass Schemaänderungen eventuell die Konsistenz, die Integrität und den Schutz der Daten gefährden, ist der Zugang zu den Daten (und zu der von der Anwendung bereitgestellten Funktionalität) nur über ein API gestattet. Deshalb besteht die Notwendigkeit, nicht nur die Datenbank- bzw. Schema-Integration, sondern in Ergänzung dazu auch die Anwendungssystem- bzw. API-Integration zu unterstützen.

Das folgende Beispiel soll verdeutlichen, wie die Benutzer heutzutage mit Anwendungssystemen arbeiten. Das Beispielszenario ist in der Einkaufsabteilung eines Unternehmens angesiedelt, findet sich aber in ähnlicher Form auch in jedem anderen Bereich. In dieser Abteilung hat ein Mitarbeiter zu entscheiden, ob ein neues Produkt eines bereits dem Unternehmen bekannten Zulieferers bestellt werden soll. Ein Einkaufssystem soll den Mitarbeiter mit der Funktion `Kaufentscheid` bei seiner Entscheidung helfen. Diese Funktion schlägt eine Entscheidung zum Kauf vor, die auf einem berechneten Qualitäts- und Zuverlässigkeitsgrad und der Nummer der zu betrachtenden Komponente basiert. Leider kennt der Mitarbeiter nur den Komponentennamen und die Nummer des Zulieferers. Da ihm die benötigten Eingabewerte nicht zur Verfügung stehen, muss der Mitarbeiter weitere Systeme heranziehen, um diese Werte zu ermitteln. Abb. 1 illustriert die einzelnen Schritte, die er dafür ausführen muss. Ausgehend von den Werten, die ihm bekannt sind, ruft er zunächst die Funktion `GibQualität` des Lagerhaltungssystems und die Funktion `GibZuverlässigkeit` des Einkaufssystems mit der Zulieferernummer auf, um Qualität und Zuverlässigkeit des Zulieferers zu erfragen. Die resultierenden Ergebnisse werden anschließend als Eingabe für die Funktion `GibGrad` zur Berechnung des Grades eingesetzt, um den ersten Eingabewert der Funktion `Kaufentscheid` zu erhalten. Außerdem ruft der Mitarbeiter die Funktion `GibKompNr` des Produktdatenmanagementsystems zur Abfrage der zugehörigen Komponentennummer auf. Nun stehen ihm die Werte zur Verfügung, die für den Aufruf der Funktion `Kaufentscheid` benötigt werden.

Während des Entscheidungsprozesses muss der Mitarbeiter mit drei unterschiedlichen Anwendungssystemen und demzufolge mit drei verschiedenen Benutzerschnittstellen arbeiten. Aus technischer Sicht setzt er von Hand eine Art Integration um, indem er die Funktionen der Anwendungssysteme aufruft und deren Ergebnisse zwischen den einzelnen Systemen hin- und herkopiert. Die Interaktion des Benutzers stellt daher die Verknüpfung der Anwendungssysteme dar. Ein solches Verfahren kostet dem Benutzer aber zum einen sehr viel Zeit und zum anderen ist es fehleranfällig. Ein typisches Beispiel hierfür ist, dass alte Werte im Zwischenspeicher versehentlich als Eingabe in ein anderes System weitergereicht werden. Außerdem stellt man fest, dass bestimmte Schritte immer wieder in derselben Reihenfolge durchgeführt werden. Diese Beobachtung führte zu der Idee, so genannte *föderierte Funktionen* anzubieten, welche die einzelnen Aufrufe der lokalen Funktionen der Anwendungssysteme implementieren und diese Schritte vor dem Benutzer verbergen. In dem gezeigten Beispiel würde dies bedeuten, dass der Benutzer lediglich eine föderierte Funktion `KaufeKomponente` statt der fünf lokalen Funktionen aufzurufen hat.

Um diese Systemhilfe zu bieten, ist eine Integration von Funktionen bzw. Anwendungssystemen zu unterstützen. Da auch DB-Referenzen in einer Benutzeranfrage enthalten sein können, benötigt man sogar einen kombinierten Ansatz für den Daten- und Funktionszugriff. Hierbei sind mehrere Integrationsszenarien denkbar. Man kann den kombinierten Zugriff mit einem reinen Daten- oder einem reinem Funktionsintegrationsansatz angehen. Bei der reinen Datenintegration werden ausschließlich SQL-Datenquellen zusammengeführt, so dass die bekannten Konzepte der FDBS und MDBS greifen. Diese Ansätze ermöglichen die Bereitstellung eines

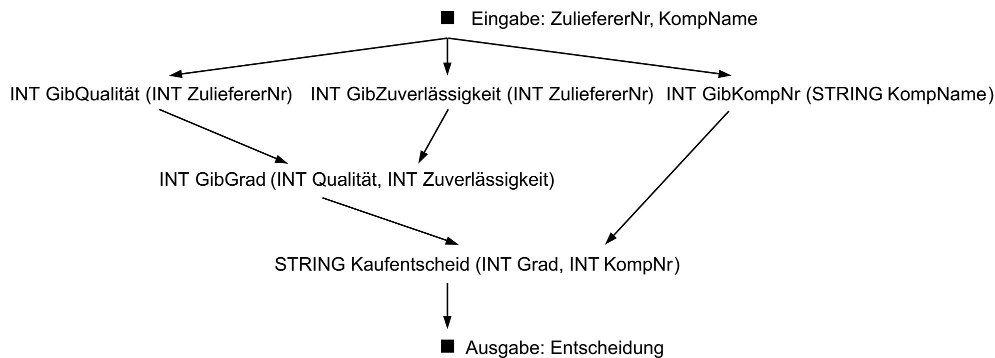

Abb. 1 Der Ablauf der einzelnen Schritte des Benutzers bei der Arbeit mit mehreren Anwendungssystemen.

globalen Schemas basierend auf den bekannten lokalen Schemata, wodurch eine generische und deshalb flexible Anfragesprache zur Verfügung gestellt werden kann. Ein allgemein anerkanntes Modell für die Integration von Daten wird in [19] als Fünf-Schichten-Architektur definiert. Es soll daher nicht weiter auf die Datenintegration eingegangen werden.

Bei der reinen Funktionsintegration sind die zu integrierenden Anwendungssysteme in der Regel auf eine festgelegte Funktionalität zugeschnitten. Dies bedeutet, dass keine lokalen Schemata verfügbar sind. Außer den nach außen offengelegten Funktionen sind keine weiteren Informationen über die Daten vorhanden. Da solche Funktionen nicht die Datenunabhängigkeit und die Flexibilität bieten, die eine deklarative DB-Anfragesprache auszeichnen, ist auch ihre Nutzung eingeschränkt, d. h., nicht vordefinierte Anfragen sind nicht möglich. Typisch hierfür ist das vorgestellte Anwendungsbeispiel.

Sollen nun Datenbank- und Anwendungssysteme in einer Architektur integriert werden, so ist die reine Datenintegration nicht mächtig genug. Stattdessen ist ein Lösungsansatz wünschenswert, der zwischen den aufgezeigten Ansätzen liegt und somit eine Mischform von Daten- und Funktionsintegration darstellt. Unseres Erachtens stellt ein FDBS eine effektive Integrationsplattform dar. Wichtige Gründe hierfür sind zum einen die deklarative Anfragesprache sowie die inzwischen ausgiebig untersuchte Integration von Datenbanksystemen. Zum anderen sind die weit verbreiteten SQL-Kenntnisse als auch die große Anzahl an Software-Produkten, welche die SQL-Schnittstelle bedienen können, wichtige Entscheidungskriterien für den Einsatz von neuen Lösungen. Eine Anfrage, die den Zugriff auf Datenbanken als auch Anwendungssysteme enthält, beinhaltet somit SQL-Prädikate sowie eine Art Zugriff auf externe Funktionen.

In diesem Aufsatz soll ein Ansatz zur Integration von Daten und Funktionen vorgestellt werden, der auf der Kopplung eines FDBS zur Integration von Daten und einer weiteren Komponente zur Funktionsintegration basiert. Hierzu führen wir in Abschnitt 2 ein Abbildungsmodell für die Funktionsintegration ein. Anschließend wird in Abschnitt 3 das Anbindungsmodell zur Kopplung der beiden Integrationskomponenten erläutert und in Abschnitt 4 Optimierungsaspekte bei der daraus resultierenden heterogenen Anfrageverarbeitung betrachtet. Abschnitt 5 führt mögliche Ausführungskomponenten für die Funktionsintegration auf, bevor in Abschnitt 6 empirische Untersuchungen mit einem realisierten Prototypen beschrieben werden. Abschließend werden die Ergebnisse zusammengefasst.

2 Abbildungsmodell

Wie in der Motivation dargestellt, soll mit der Funktionsintegration der Benutzer in seiner Arbeit entlastet werden, indem ihm für bestimmte, immer wiederkehrende Sequenzen von Funktionsaufrufen die Arbeit erleichtert wird. Für diese Sequenzen, die in gewisser Weise Prozesse darstellen, soll die Kombination der lokalen Funktionsaufrufe hinter einer föderierten Funktion verborgen werden. Auf diese Weise muss der Benutzer nicht mehr selbst die Verknüpfung der einzelnen lokalen Funktionen und der damit verbundenen Konversion von Datentypen oder auch die Fehlerbehandlung vornehmen. Eine ähnliche Motivation ist auch die Basis für die Entwicklung von Workflow-Systemen. Im Gegensatz zu unseren Bestrebungen sollen die Workflow-Systeme aber das korrekte Durchlaufen von Geschäftsprozessen unterstützen und die Interaktion des Benutzers steuern. In unserem Fall hingegen soll genau diese Interaktion nicht mehr stattfinden müssen.

Um die Umsetzung einer solchen Abbildung von föderierten Funktionen auf lokale Funktionen wartbar zu machen, soll eine entsprechende Abbildungssprache entwickelt werden, so dass sich der vollständige Ansatz aus einem Beschreibungs- und einem Ausführungsmodell zusammensetzt. Verzichtet man auf das Beschreibungsmodell, so treten folgende Nachteile auf:

– Die Abbildung auf die lokalen Funktionen ist in den meisten Fällen im Programmcode verborgen. Möchte man die implementierte Abbildung nachvollziehen, so ist man gezwungen, den Programmcode zu analysieren.

– Des Weiteren liegt häufig keine oder nur eine rudimentäre Dokumentation der umgesetzten Abbildung vor, so dass auch hier das Nachvollziehen Schwierigkeiten bereitet.
– Außerdem ist die implementierte Lösung aufwendig zu warten und unflexibel bezüglich sich ändernder Anforderungen. Soll beispielsweise eine weitere Datenquelle hinzugefügt oder eine bestehende ersetzt werden, so muss der Programmcode verstanden und entsprechend geändert werden. Hinzu kommt, dass häufig die ursprünglichen Entwickler nicht mehr da sind, so dass andere Mitarbeiter sich erst mit dem Code vertraut machen müssen, bevor die Änderungen vorgenommen werden können.

Daher sollte es die Möglichkeit geben, die Funktionsabbildung explizit zu spezifizieren. Auch hier können wieder Parallelen zu Workflow-Systemen gezogen werden, da bei ihnen ebenfalls die Abbildung eines Workflow-Prozesses explizit auf lokale Aktivitäten festgelegt wird. Die Workflow Management Coalition hat hierzu zunächst die Sprache WPDL (*workflow process definition language*) und inzwischen eine XML-Version XPDL (*XML process defintion languages*) definiert [22].

Geht man davon aus, dass eine solche Funktionsabbildung explizit spezifiziert wird, so kann in einem zweiten Schritt diese Spezifikation dazu dienen, um zur Umsetzung der Abbildung vom Programmcode so viel wie möglich zu generieren. Mit dieser Vorgehensweise können alle oben aufgeführten Nachteile entschärft werden, indem die Abbildung explizit spezifiziert und dokumentiert wird und somit einfacher zu warten ist. Überdies verkürzt die (teilweise) Generierung von Funktionen die Entwicklungszeit.

Unsere Abbildungssprache ist auf Basis von XML (*eXtended Markup Language*, [23]) definiert, da es eine standardisierte Syntax zur Verfügung stellt, mit welcher wir den Aufbau der Sprache in standardisierter Form definieren können. Außerdem kann die Funktionalität von XML durch Nutzung der wachsenden Anzahl verwandter Standards wie XLink [23] oder XSLT [23] erweitert werden. Ein zusätzlicher Vorteil ist in der großen Zahl verfügbarer Werkzeuge zu XML zu sehen.

Der vorgestellte Ansatz besteht aus zwei Teilen: der erste beschreibt die föderierten und lokalen Systeme sowie ihre APIs, während der zweite die Funktionsabbildung betrifft. In den folgenden Abschnitten gehen wir auf diese beiden Teile näher ein.

2.1 Systembeschreibung

Der erste Teil der Abbildungsbeschreibung charakterisiert sowohl die zu integrierenden lokalen Systeme als auch das föderierte Zielsystem. Die resultierenden Spezifikationen enthalten jeweils die Signaturen der Funktionen, die Programmiersprache der Schnittstelle als auch Informationen über das Kommunikationsprotokoll. Außerdem stellt die Sprache ein Typsystem zur Verfügung, mit welchem benutzerdefinierte, komplexe Datentypen wie strukturierte Datentypen und geschachtelte Aggregationen als auch einfache Datentypen definiert werden können. Danach können die Funktionen einschließlich ihres Namens sowie der Parameternamen und -datentypen beschrieben werden. Zusätzlich ist anzugeben, ob es sich um Eingabe- oder Ausgabeparameter handelt.

Jede Schnittstelle, sowohl solche der lokalen Systeme als auch des globalen Systems, wird in einem separaten XML-Dokument beschrieben. Die resultierenden Dokumente werden anschließend in einem Repository abgelegt.

Die lokale Funktion `GibQualität` des Lagerhaltungssystems in unserem Beispiel wird wie in Abb. 2 gezeigt beschrieben[1].

```
<?xml version="1.0"?>
<!DOCTYPE systems_desc SYSTEM "system_desc.dtd">
<system id="Lager" type="source">
  <sys_name> Lagerhaltungssystem </sys_name>
  <description type="system" language="de">
    Dies ist das Lagerhaltungssystem,
    das die Funktion GibQualität enthält.
  </description>
  <prog_language> C++ </prog_language>
  <communication>
    <os> AIX </os>
    <ip> 444.444.444.444 </ip>
    <port> 71000 </port>
  </communication>
  <function id="GibQuali">
    <func_name> GibQualität </func_name>
    <description type="function" language="de">
      Diese Funktion hat einen Ein- und
      einen Ausgabeparameter.
    </desription>
    <parameter id="L_ZNr" type="IN">
      <para_name> ZuliefererNr </para_name>
      <datatype> integer </datatype>
    </parameter>
    <parameter id="L_Qual" type="OUT">
      <para_name> Qualität </para_name>
      <datatype> string </datatype>
    </parameter>
  </function>
</system>
```

Abb. 2 Systembeschreibung des Lagerhaltungssystems.

2.2 Abbildungsspezifikation

Der nächste Schritt in dem deskriptiven Ansatz betrifft die Abbildungsspezifikation. Die grundlegende Idee dabei ist, die Abbildung von föderierten auf lokale Funktionen aus dem Blickwinkel ihrer Parameter zu beschreiben, indem Abhängigkeiten zwischen den Parametern

[1] Die gezeigte Syntax entspricht nicht exakt der entwickelten Abbildungssprache. Bei der dargestellten Sprache handelt es sich um eine vereinfachte Form, da der tatsächliche Sprachumfang viel größer ist und den Rahmen dieses Aufsatzes sprengen würde.

spezifiziert werden. Ausgehend davon, dass keine Abhängigkeiten zwischen den föderierten Funktionen existieren, kann die Abbildung für jede föderierte Funktion separat beschrieben werden. Bei dieser Vorgehensweise kann die Abbildung als ein gerichteter, azyklischer Graph dargestellt werden.

Der in Abb. 1 dargestellte Ablauf entspricht einem Graph, welcher die Logik zur Abbildung der föderierten Funktion `KaufeKomponente` auf die lokalen Funktionen `GibQualität`, `GibZuverlässigkeit`, `GibKompNr`, `GibGrad` sowie `Kaufentscheid` darstellt. Dabei werden die Eingabeparameter `ZuliefererNr` und `KompName` der globalen Funktion als Eingabe für die lokalen Funktionen `GibQualität`, `GibZuverlässigkeit` und `GibKompNr` genutzt. Deren Ausgabeparameter werden wiederum auf die Eingabeparameter der Funktionen `GibGrad` als auch `Kaufentscheid` abgebildet. Schließlich wird der Ausgabeparamater von `Kaufentscheid` auf die Ausgabe von `KaufeKomponente` projiziert.

Betrachtet man die Parameter und ihre Abhängigkeiten näher, so kann daraus ein gerichteter, azyklischer Graph abgeleitet werden, wobei die Parameter die Knoten und die Abhängigkeiten die Kanten des Graphen darstellen. Momentan fehlt jedoch noch die Definition der Ausführungsreihenfolge der einzelnen lokalen Funktionen. Diese muss jedoch nicht explizit angegeben werden, sondern es können mit einer topologischen Sortierung mögliche Ausführungssequenzen ermittelt werden. Da die Abhängigkeiten nur eine partielle Reihenfolge auf den Parametern definieren, sind mehrere topologische Sortierergebnisse möglich. Dies deutet darauf hin, dass manche Parameter unabhängig sind und diese Funktionen parallel ausgeführt werden können. Somit sind wir in der Lage, die Parameterabbildung als auch die Ausführungsreihenfolge der Quellfunktionen in kohärenter Art und Weise darzustellen.

Bisher können nur Systeme aufeinander abgebildet werden, die hinsichtlich ihrer Parameter und Datentypen kongruent sind. Wir benötigen aber auch Informationen, um Heterogenitäten überwinden zu können. Dazu gehören Operationen auf Parametern, die beispielsweise ihre Datentypen konvertieren oder ihre Werte kombinieren. Solche Operationen werden als weitere Funktionen, sog. Hilfsfunktionen, betrachtet, die in derselben Weise in die Abbildungsbeschreibung eingebettet werden, wie dies für die zu integrierenden Quellfunktionen der Fall ist. Auf diese Weise bleibt die Darstellung der Abbildung homogen.

In der Abbildungssprache wird der Abhängigkeitsgraph in einer Form beschrieben, bei der die betroffenen Knoten (d. h. die Funktionsparameter) sowie die Kanten (d. h. die Abhängigkeiten) aufgezählt werden. Dabei greifen wir auf die Verknüpfungssprache XLink [23] zurück. Dieser Verknüpfungsmechanismus erweitert das bekannte Konzept der *Links* in HTML (*HyperText Markup Language*, [23]), mit dem HTML-Seiten verbunden werden. Die für die Abbildung relevanten Teile von XLink sind die so genannten erweiterten Verknüpfungen (*extended links*), mit welchen eine beliebige Anzahl von XML-Dokumenten verknüpft werden können. Das Besondere dabei ist, dass die verknüpften Dokumente keine ausgehende Verknüpfung haben müssen. So können wir Verknüpfungen zwischen den Parametern definieren, die in den einzelnen Systembeschreibungen aufgeführt werden. Eine erweiterte Verknüpfung besteht aus so genannten *Locators* und *Arcs*. Dabei definieren die Locators die teilnehmenden Quellen und die Arcs spezifizieren das traversierende Verhalten.

Zur Beschreibung der Abhängigkeiten zwischen den Parametern nutzen wir das Konzept der erweiterten Verknüpfung und repräsentieren Parameter als Locators und Abhängigkeiten als Arcs. Die DTD sowie die Abbildung für unser Beispiel ist in Abb. 3 dargestellt.

```
<!ELEMENT function_map  (node, dependency+)>
<!ATTLIST function_map
  xmlns:xlink CDATA                          #FIXED "http://www..."
  xlink:type simple|extended|locator|arc #FIXED "extended">
<!ELEMENT node EMPTY>
<!ATTLIST node
  xmlns:xlink CDATA                          #FIXED "http://www..."
  xlink:type simple|extended|locator|arc #FIXED "locator"
  id          ID                             #REQUIRED
  xlink:href  CDATA                          #REQUIRED>
<!ELEMENT dependency EMPTY>
<!ATTLIST dependency
  xmlns:xlink CDATA                          #FIXED "http://www..."
  xlink:type simple|extended|locator|arc #FIXED "locator"
  from        IDREF                          #REQUIRED
  to          IDREF                          #REQUIRED>

<?xml version="1.0"?>
<!DOCTYPE function_map SYSTEM "map.dtd">
<map>
<!-- Parameter der föderierten Funktion -->
  <node id="FF_Entsch" href="c:\data\FoedFunk.xml#
                            id("FF_Entsch")/self::parameter"/>
  <node id="FF_ZNr" href="c:\data\FoedFunk.xml#
                         id("FF_ZNr")/self::parameter"/
  <node id="FF_KName" href="c:\data\FoedFunk.xml#
                           id("FF_KName")/self::parameter"/
<!-- Parameter der lokalen Funktionen -->
  <node id="L_ZNr" href="c:\data\Lager.xml#
                        id("L_ZNr")/self::parameter"/>
  <node id="L_Qual" href="c:\data\Zielsystem.xml#
                         id("L_Qual")/self::parameter"/>
  ....
<!-- Abhängigkeiten -->
  <dependency from="FF_ZNr" to="L_ZNr"/>
  <dependency from="L_Qual" to="E_Qual"/>
  <dependency from="FF_ZNr" to="E_ZNr"/>
  ....
</map>
```

Abb. 3 DTD der Abbildungsspezifikation und ein Auszug der XML-Beschreibung unseres Beispiels.

Für die vorgestellte Abbildungssprache sehen wir folgende, für uns wichtige Eigenschaften: sie ist einfach, leichtgewichtig und unabhängig von einem Ausführungsmodell. Auf diese Weise kann je nach Anforderungen eine entsprechende Ausführungsumgebung gewählt werden.

Die so beschriebene Abbildung kann nun mit Hilfe von XSLT in jedes beliebige ASCII-Format transformiert werden, wie z. B. Java-Code für einen Applikations-Server, IDL für CORBA-Implementierungen, C++-Code

für selbstentwickelte Ausführungskomponenten oder als Eingabe für kommerzielle Produkte.

3 Anbindungsmodell

Nachdem in Abschnitt 2 verdeutlicht wurde, dass wir eine kombinierte Integration von Daten und Funktionen mit Hilfe eines FDBS erreichen wollen, und in Abschnitt 3 die reine Funktionsintegration betrachtet wurde, wollen wir uns als Nächstes auf die Verbindung dieser beiden Integrationsansätze konzentrieren. Da die SQL-Schnittstelle als die externe Benutzerschnittstelle erhalten bleiben soll, ist das FDBS das führende System, an welches die (globalen) Anfragen gestellt werden. Dies bedeutet aber auch, dass die Ausführungskomponente der Funktionsintegration an das FDBS angebunden und von diesem angesprochen werden muss. Demnach hat das FDBS zu erkennen, welche der Teile der Anfrage vom FDBS auszuführen sind und welche von der Komponente zur Integration der Funktionen (kurz KIF) übernommen werden müssen. Demzufolge ist ein Mechanismus einzusetzen, der es dem FDBS erlaubt, die KIF anzusteuern und ihre (Zwischen-) Ergebnisse zu übernehmen.

3.1 Verfügbare Anbindungsmechanismen

Der aktuelle SQL-Standard [10] stellt drei Mechanismen zur Verfügung: Wrapper nach SQL/MED (*Management of External Data*, [11]), benutzerdefinierte Tabellenfunktionen (*user-defined table functions*) und gespeicherte Prozeduren (*stored procedures*).

Wrapper unterstützen die Anbindung von SQL- sowie Nicht-SQL-Quellen an einen SQL-Server, in unserem Fall das FDBS, und ermöglichen die „engste“ Anbindung an das FDBS. Der sich in Arbeit befindende Standard SQL/MED definiert Wrapper als einen Mechanismus, mit dem ein SQL-Server auf externe Daten zugreifen kann, die von fremden Servern (*foreign server*) verwaltet werden. Dabei gibt es pro Server-Typ je einen Wrapper. Ein Wrapper kann für den Zugriff auf mehrere Server desselben Typs genutzt werden. Die externen Daten werden dem FDBS in Form von fremden Tabellen (*foreign tables*) bekannt gemacht. Dabei handelt es sich um abstrakte Tabellen, die in einer SQL-Anfrage referenziert werden können und deren Katalogeinträge aufzeigen, in welchem externen Server sie tatsächlich liegen. Im aktuellen Dokument ist bis dato nur der lesende Zugriff festgelegt. Ankündigungen zufolge und der Tatsache, dass kommerzielle Produkte den schreibenden Zugriff bereits heute unterstützen, ist zu erwarten, dass der schreibende Zugriff auch in den Standard aufgenommen wird. Der Wrapper stellt aber nicht nur eine Schnittstelle dar, sondern kann auch die bestehende Funktionalität des fremden Servers durch zusätzliche Funktionalität erweitern. Diese erweiterte Funktionalität kann in die globale Anfrageoptimierung eingebunden werden. Aus diesem Grund kann sich die Entwicklung eines Wrappers als komplex erweisen, wenn z. B. die Funktionalität der einzubindenen Quelle durch den Wrapper ergänzt werden soll, um lokale Optimierungen zu unterstützen. Mit dem Einsatz von Wrappern können Daten als auch föderierte Funktionen innerhalb einer globalen Anfrage abgerufen und deren Ergebnisse im FDBS zusammengeführt und weiterverarbeitet werden. Der Zugriff auf die föderierte Funktion erfolgt in einer SELECT-Anweisung wie auf eine Tabelle:

```
SELECT Entscheidung
FROM KaufeKomponente
WHERE KompName='xyz' AND ZuliefererNr=1234
```

Benutzerdefinierte Tabellenfunktionen sind eine weitere Möglichkeit, um Nicht-SQL-Quellen aus Sicht des FDBS einzubinden. Benutzerdefefinierte Tabellenfunktionen sind externe Funktionen, die eine Tabelle anstatt einem skalaren Wert zurückliefern. Mit solch einer Funktion kann dem FDBS fast jede Datenquelle als Tabelle vorgegaukelt werden, indem beispielsweise ein Java-Programm geschrieben wird, das die gewünschten Daten ermittelt, anhand der Eingabeparameter filtert und das Ergebnis zeilenweise an das FDBS zurückgibt. Tabellenfunktionen werden mit den benötigten Eingabewerten in der FROM-Klausel der SQL-Anweisung referenziert. Diese Handhabung erlaubt es, die Tabellenfunktionen in der SQL-Anweisung wie eine Tabelle zu behandeln. Tabellenfunktionen sind schnell umzusetzen, haben aber den Nachteil, dass sie zum einen nur lesenden Zugriff unterstützen und zum anderen keinen Beitrag zur Anfrageoptimierung liefern. Auch bei dieser Variante werden globale Anfragen unterstützt, die Daten als auch föderierte Funktionen referenzieren. Mit IBMs DB2 [6] wird die Anweisung folgendermaßen formuliert:

```
SELECT Entscheidung
FROM TABLE KaufeKomponente('xyz', 1234)
          AS KK
```

Gespeicherte Prozeduren stellen die dritte Alternative dar, über die externe Quellen aufgerufen werden können. Der Aufbau der Prozeduren ist vergleichbar mit den Tabellenfunktionen; sie werden mit den entsprechenden Eingabewerten aufgerufen und liefern ihr Ergebnis als Tabelle zurück. Mit Hilfe von Prozeduren kann lesender als auch schreibender Zugriff umgesetzt werden. Da sie jedoch nicht aus einer SELECT-Anweisung heraus, sondern mit einer expliziten CALL-Anweisung gestartet werden (z. B. `CALL KaufeKomponente('xyz', 1234)`), ist es nicht möglich, den Zugriff auf Daten und Funktionen innerhalb einer globalen SQL-Anweisung zu verwirklichen. Dies ist darin begründet, dass die gespeicherten Prozeduren lediglich die vorab spezifizierte Funktionsintegration implementieren, jedoch nicht auf Datenbanken zugreifen. Soll also dynamisch die Integration von Funktionen und Daten erfolgen und soll diese Integration von dem FDBS durchgeführt werden, so ist dies aufgrund der

expliziten CALL-Anweisung nicht möglich. Ist die Zusammenführung von Ergebnissen aus den SQL-Quellen als auch aus den Anwendungssystemen gewünscht, so muss dies über zwei DBS-Anweisungen realisiert und die Ergebnismengen müssen in der Anwendung zusammengesetzt werden.

3.2 Abstrakte Tabellen

Alle aufgeführten Alternativen müssen zumindest die programmiertechnische Verbindung zwischen FDBS und Funktionskomponente implementieren. Sie müssen also in der Lage sein, eine Teilanfrage oder zumindest Eingabewerte von dem FDBS entgegenzunehmen, die entsprechenden Funktionen des angekoppelten Systems aufzurufen und das Ergebnis in Form von Tabellen zurückzuliefern. Dies bedeutet vor allem, dass eine Abbildung von Funktionen auf Tabellen und umgekehrt zu unterstützen ist. Bei solch einer Abbildung müssen die konzeptionellen Unterschiede zwischen einem Daten- und einem Funktionenmodell überwunden werden. Ein naheliegender Ansatz stellt eine Funktion als Tupel ihrer Ein- und Ausgabeparameter dar, d. h., die Spalten einer Tabelle werden durch die Parameter der Funktion festgelegt. Auf diese Weise kann die Funktion über den Zugriff auf eine sie repräsentierende abstrakte Tabelle aufgerufen werden. Versucht man, den Funktionsaufruf mittels einer SQL-Anweisung auf der abstrakten Tabelle nachzubilden, so entspricht der lokalen Funktion `String Kaufentscheid(int Grad, int KompNr)` mit der Ausgabe `Entscheidung` und den Eingabeparametern `Grad` und `KompNr` unseres Beispiels die folgende SELECT-Anweisung:

```
SELECT Entscheidung
FROM Kaufentscheid
WHERE Grad=konstante AND KompNr=konstante
```

Das bedeutet, dass nur die Ausgaben der Funktion projiziert werden. Außerdem müssen die benötigten Werte für die Eingabeparameter über Konstantenprädikate mit den korrespondierenden Spalten der abstrakten Tabelle übergeben werden, wobei der Vergleichsoperator derjenigen Semantik entsprechen muss, die von der Funktion vorgegeben ist.

Bei der Auswahl einer geeigneten Anbindungsvariante zeigt sich, dass gespeicherte Prozeduren keine nahtlose Integration innerhalb einer Anfrage gestatten und deshalb nicht eingesetzt werden sollten. Benutzerdefinierte Tabellenfunktionen ermöglichen eine bessere Integration in globale Anfragen über Daten und Funktionen hinweg, erlauben jedoch nicht den schreibenden Zugriff. Somit stellen Wrapper die vielversprechendste Alternative dar, da sie eine allgemeingültige und mächtige Umsetzung der Anbindung bieten. Mit Hilfe der Wrapper bleiben die Funktionszugriffe für den Anwender vollkommen transparent, da er nur mit Tabellen arbeitet und sich somit keiner speziellen Vorgehensweisen oder Einschränkungen bewusst sein muss. Überdies können Wrapper in der Form implementiert werden, dass sie zur globalen Anfrageoptimierung beitragen. Zwar wird der Standard in seiner ersten Version nicht den schreibenden Zugriff enthalten, da diese Funktionalität aber bereits von den führenden Datenbankherstellern angeboten wird, ist fest damit zu rechnen, dass sie in folgenden Versionen Teil des Standards sein wird. Aus diesen Gründen fokussieren wir in den folgenden Abschnitten auf den Wrapper-Ansatz.

4 Optimierungsmodell

Im folgenden Abschnitt soll betrachtet werden, inwiefern Optimierungsaspekte bei der vorgestellten Architektur berücksichtigt werden können, wenn die Anbindung der KIF an das FDBS mittels eines standardisierten Wrappers erfolgt. Wie in Abschnitt 3 aufgezeigt, muss der Wrapper zumindest eine struktur- und bedeutungstreue Abbildung der Funktionsaufrufe von SQL-Anfragen gewährleisten können. Überdies ist es wünschenswert, dass er weitere Funktionalität übernehmen kann, die eine Optimierung der globalen Anfrage ermöglicht. Eine häufig genutzte Größe, die im verteilten Umfeld Optimierungsmaßnahmen wirksam bestimmt, ist die Anzahl der Ergebniszeilen, die zwischen den Systemen zu transportieren sind. Die Minimierung dieser Größe wird vor allem durch eine Verschiebung größtmöglicher Anteile der Verarbeitung der Daten an deren Quelle erreicht (das so genannte *Pushdown*). Die Funktionalitätserweiterung seitens des Wrappers macht allerdings nur dann Sinn, wenn er auf der Quellseite platziert wird, denn nur in diesem Fall lässt sich der Datentransport zwischen Quelle und FDBS minimieren. Befindet sich der Wrapper dagegen auf der FDBS-Seite, so sind nach wie vor alle zu verarbeitenden Daten über das Netzwerk zu bewegen.

Bevor wir das Optimierungspotenzial von Wrappern näher diskutieren, führen wir eine Klassifizierung möglicher Wrapper-Funktionalität ein. Das grundlegende Ziel dabei ist, soviel Optimierungseffekte mit so wenig Implementierungsaufwand wie möglich umzusetzen. Anschließend gehen wir auf Operationen ein, deren Ausführung im FDBS der im Wrapper vorzuziehen ist.

Die *Kernfunktionalität* des Wrappers besteht aus all jenen Funktionen, die benötigt werden, um zwischen den abstrakten Tabellen und den Funktionen abbilden zu können. Hierzu wird für jede Referenz auf eine abstrakte Tabelle innerhalb einer SQL-Anweisung deren vollständiger Inhalt materialisiert und zur Verfügung gestellt.

Unter *Basisfunktionalität* bezüglich abstrakter Tabellen verstehen wir die Projektion bzw. Selektion auf eine Teilmenge der Spalten bzw. der Zeilen. Des Weiteren können beliebige Boolesche Kombinationen von Vergleichsprädikaten mit Konstanten aufgeführt werden.

Diese Prädikate enthalten einen Operator $\Theta \in \{=, \neq, <, \leq, >, \geq\}$.

Die *erweiterte Funktionalität* zeichnet sich schließlich durch mächtigere Operationen wie Gruppierung und Aggregation, Teilanfragen bezüglich abstrakter Tabellen und Mengenvergleiche aus. Außerdem könnte ein Verbund zweier abstrakter Tabellen innerhalb des Wrappers implementiert werden, um die Kommunikation, verursacht durch Wrapper-Aufrufe seitens des FDBS, zu minimieren. Andererseits werden Verbunde im FDBS sehr effizient umgesetzt, so dass kritisch abgewogen werden sollte, ob eine Reimplementierung des Verbundes gerechtfertigt ist.

Da der Wrapper nur Anfragen erhält, die abstrakte Tabellen referenzieren, besteht keine Notwendigkeit, Funktionalität zu implementieren, die über die Verarbeitung dieser Anfragen (Subselects) hinausgeht (dazu gehören Operationen wie die Vereinigung, Differenz und Durchschnitt). Separate Teilanfragen werden normalerweise durch unabhängige Anfragen repräsentiert, die anschließend durch das FDBS zusammengesetzt werden. Teilanfragen bezüglich Tabellen im FDBS sollten nicht durch den Wrapper ausgeführt werden, da das FDBS diese Funktionen in der Regel besser handhaben kann.

Der Transfer der Ergebnismenge vom Wrapper zum FDBS wird mittels einer abstrakten Tabelle (*abstract table queue, ATQ*) vorgenommen. Diese ATQ lässt sich entweder im Pipeline-Modus nach Verfügbarkeit von Ergebniszeilen oder im Blockmodus am Ende des Funktionsaufrufes bereitstellen. Des Weiteren können mit der Ergebnismenge nützliche Informationen zurückgeliefert werden, die für die globale Anfrageverarbeitung wichtig sind. Ein Beispiel hierfür wäre ein Vermerk, dass die Ergebnismenge bereits sortiert ist. Diese Information sollte als Hinweis an die Anfrageverarbeitung vermittelt werden, damit diese nicht weitere Sortierungen vornimmt.

4.1 Grundlegende Optimierungsaspekte

Als nächstes soll die Umsetzung der Optimierungsaspekte in formaler Form betrachtet werden, die durch die Basis- und erweiterte Funktionalität unterstützt werden. Wir beschränken uns dabei auf jene Aspekte, die durch die Einbeziehung von Funktionen auftreten. Optimierungsaspekte wie der Pushdown von Operationen und das Reduzieren der zu übertragenden Datenmenge zwischen FDBS und Funktionskomponente wurden bereits ausgiebig analysiert [21] und werden nicht näher betrachtet.

Zunächst wird eine formale Darstellung der Anfrageverarbeitung vorgestellt, die auf der relationalen Algebra basiert, welche um zwei Operatoren erweitert wird. Ausgehend von der allgemeinen Signatur einer Funktion $f(i_1, ..., i_m, o_1, ..., o_n)$ definieren wir I und O als die Mengen der Eingabe- bzw. Ausgabeparameter. Somit kann die Funktionssignatur auch als $f(I, O)$ dargestellt werden. In unserem Beispiel hat die föderierte Funktion `KaufeKomponente(IN ZuliefererNr int, IN KompName String, OUT Entscheidung String)` die Menge $I := \{\textit{ZuliefererNr}, \textit{KompName}\}$ an Eingabeparametern und die Ausgabeparametermenge $O := \{\textit{Entscheidung}\}$ und lässt sich darstellen als `KaufeKomponente(I, O)`.

Geht man davon aus, dass die Abbildung von Funktionen auf abstrakte Tabellen vorsieht, dass die Parameter der Funktion die Spalten der Tabelle festlegen, dann wird bei einer einfachen Abbildung die Funktion $f(i_1, ..., i_m, o_1, ..., o_n)$ als abstrakte Tabelle $T_a(i_1, ..., i_m, o_1, ..., o_n)$ oder kurz $T_a(I, O)$ beschrieben. Ein Funktionsaufruf kann daher als folgender algebraischer Ausdruck dargestellt werden:

$$f(I, O) \leftrightarrow \pi_O\big(\sigma_I(T_a)\big)$$

Daraus folgt, dass ein Funktionsaufruf äquivalent zu einer Selektion auf einer abstrakten Tabelle T_a ist, die durch Prädikate angewendet auf die Werte der Eingabeparameter spezifiziert ist. Außerdem erfolgt eine Projektion auf die Ausgabeparameter. Da dieser Term die Funktionalität einer Funktion exakt beschreibt, wird dafür ein neuer Operator φ eingeführt:

$$\varphi_{I,O}(T_a) = \pi_O\big(\sigma_I(T_a)\big)$$

Da sich der Operator φ immer auf die Parametermengen I und O bezieht, wird er im Folgenden ohne die explizite Angabe der Parameter verwendet. Um die Funktionalität einer SELECT *-Anweisung, also die Projektion auf alle Attribute $\pi_{I,O}$ abbilden zu können, müssen die Werte der Eingabeparameter zu den entsprechenden Ergebniszeilen des Funktionsaufrufes konkateniert werden. Für diese Operation führen wir den Operator κ ein:

$$\begin{aligned}\kappa_I\big(\varphi(T_a)\big) &= I \parallel \varphi(T_a) = I \parallel \pi_O\big(\sigma_I(T_a)\big)\\ &= \pi_{I,O}\big(\sigma_I(T_a)\big) = \sigma_I(T_a)\end{aligned}$$

Mit Hilfe dieser Definitionen wird im Folgenden die Funktionalität diskutiert, die von dem Wrapper angeboten werden sollte, um die Funktionalität des reinen Funktionsaufrufes zu erweitern. Aus Platzgründen wird nur die Basisfunktionalität partiell betrachtet. Dabei sollen vor allem die Auswirkungen der Funktionsaufrufe auf die Anfrageverarbeitung verdeutlicht werden. Ausführlichere Beschreibungen finden sich in [4].

Als Vertreter der Basisfunktionalität wird die Selektion herangezogen, wobei wir uns auf die Untersuchung von Selektionen basierend auf Eingabeparametern beschränken. Zunächst werden jedoch einige Voraussetzungen eingeführt, die für die folgenden Betrachtungen wesentlich sind. Wie bereits beschrieben, müssen die Werte der Eingabeparameter den Prädikaten in der WHERE-Klausel der zu evaluierenden SQL-Anweisung entnommen werden. Nur wenn die WHERE-Klausel für jeden

einzelnen Eingabeparameter ein Vergleichsprädikat spezifiziert, das der Semantik der Funktion entspricht, kann die Ergebnismenge mittels eines einzigen Funktionsaufrufs ermittelt werden. In allen anderen Fällen müssen die fehlenden Eingabewerte durch zusätzliche Funktionsaufrufe kompensiert werden. Im ungünstigsten Fall müssen sogar alle Eingabewerte ermittelt werden. Um eine solche Substitution durchführen zu können, d. h., die ursprüngliche Anfrage durch mehrere Funktionsaufrufe zu simulieren, müssen Restriktionen bezüglich der Wertebereiche der Eingabeparameter beachtet werden:

- Die Kardinalität $K(T_a, i)$ jedes Eingabeparameters i einer abstrakten Tabelle T_a muss endlich sein.
- Die Elemente seines Wertebereichs $W(T_a, i)$ müssen aufzählbar sein.

Andernfalls kann die Anzahl der benötigten kompensierenden Funktionsaufrufe nicht bestimmt werden, so dass die Anfrage zurückgewiesen wird.

Bei der Betrachtung der Selektion auf Eingabeparametern nehmen wir an, dass der im Prädikat spezifizierte Vergleichsoperator demjenigen Operator entspricht, der von der Funktion implementiert ist, und dass weiterhin alle erforderlichen Eingabewerte mittels Konstantenprädikate gegeben sind. In diesem Fall verursacht die Selektion genau einen Funktionsaufruf.

Fehlt der Wert eines Eingabeparameters i, so müssen kompensierende Funktionsaufrufe ermittelt werden. Das bedeutet, dass für jedes einzelne Element der Wertemenge $W(T_a, i)$ des fehlenden Eingabewertes eine Funktion aufgerufen werden muss. Die restlichen bekannten Eingabewerte bleiben dabei dieselben. Die Anzahl der Aufrufe wird somit multipliziert mit der Kardinalität jedes unspezifizierten Eingabewertes. Somit bewirkt eine Operation $\sigma_{i_1}(T_a)$ mit nur einem spezifizierten Eingabewert i_1 die folgende Anzahl n_{fc} von Funktionsaufrufen:

$$
\begin{aligned}
n_{fc} &= 1 \times K(T_a, i_2) \times K(T_a, i_3) \times \cdots \times K(T_a, i_m) \\
&= 1 \times \prod_{k=i_2}^{i_m} K(T_a, k)
\end{aligned}
$$

Das bedeutet also: wenn $I = i_1, ..., i_m$ und $\sigma_{(i_1=1)}(T_a)$ ausgeführt werden soll, so müssen $K(T_a, i_2) \times K(T_a, i_3) \times \cdots \times K(T_a, i_m)$ kompensierende Funktionen ausgeführt und deren Ergebnismengen vereinigt werden.

Ähnliche Aspekte müssen betrachtet werden, wenn der Vergleichsoperator eines Prädikates nicht demjenigen entspricht, der von der Funktion implementiert ist. Eine Funktion $f(i, o)$ berechnet normalerweise das Prädikat ($i = Konstante$), während die SQL-Anweisung beispielsweise auch ein Prädikat ($i \leq Konstante$) spezifizieren kann. In solchen Fällen sind mehrere Funktionsaufrufe notwendig, um das erwünschte Ergebnis zu erhalten. Dabei müssen wieder die oben aufgeführten Restriktionen beachtet werden, um die erforderlichen Funktionsaufrufe ermitteln zu können. In unserem Beispiel sind diesbezüglich neben einem Funktionsaufruf mit ($i = Konstante$) weitere Aufrufe für jedes Element aus der Wertemenge von i auszuführen, die kleiner als die spezifizierte Konstante sind. Daraus folgt, dass die Vereinigung der Ergebnismengen mehrerer Selektionen ermittelt werden muss. Ausgehend von $W(T_a, i) = \{1, 2, 3, \cdots, 10\}$ ergibt sich folgendes Beispiel:

$$
\begin{aligned}
\sigma_{i \leq 5}(T_a) = \sigma_{i=5}(T_a) \cup \sigma_{i=4}(T_a) \cup \sigma_{i=3}(T_a) \\
\cup\, \sigma_{i=2}(T_a) \cup \sigma_{i=1}(T_a)
\end{aligned}
$$

4.2 Kostenmodell

Abschließend sollen neue Aspekte des Kostenmodells betrachtet werden, die durch die Hinzunahme von Funktionen auftreten. Im Allgemeinen enthalten die Kosten einer Anfrage die Kosten für die Kommunikationszeit, Prozessorzeit, Platten-E/A sowie Hauptspeicher. In unserem Fall stellen die Kosten für die Kommunikation und die Plattenzugriffe den Löwenanteil bei der Anfrageausführung dar. Was die Kommunikationszeit betrifft, so greifen die existierenden Kostenmodelle für die heterogene Anfrageverarbeitung auch in unserem Fall. Zur Minimierung der Kommunikationszeit zwischen FDBS und Datenquelle sollte die zu transportierende Datenmenge möglichst stark reduziert werden. Dies wird in erster Linie durch das Verschieben möglichst vieler Operationen zur Datenquelle bzw. dem Wrapper erreicht.

Trotzdem muss die traditionelle Kostenberechnung für den Zugriff auf eine Tabelle T überarbeitet werden, wenn zukünftig auch Funktionsaufrufe enthalten sind. Selinger et al. [19] führten die folgende Kostenabschätzung basierend auf Plattenzugriffen und Prozessorzeit ein:

$$
\mathit{Cost}(T) = \mathit{pagefetches}(T) + W \times \mathit{systemcalls}(T)
$$

In dieser Formel beschreibt die Funktion *pagefetches* die Anzahl der physischen Zugriffe auf externe Speichermedien (in den meisten Fällen den Platten). Die Funktion *systemcalls* charakterisiert die Anzahl der Aufrufe an das Zugriffssystem, welche sich von der Anzahl der für die Anfrageverarbeitung benötigten Zeilen ableiten lässt. Damit ist diese Funktion ein guter Indikator für die erwartete Prozessorlast. Der Faktor W dient zur Berücksichtigung der Art der Systemauslastung. Ist das System CPU-lastig, so sollte W eher groß gewählt werden, um Ausführungspläne mit geringem CPU-Bedarf zu bevorzugen. Bei einem E/A-lastigen System sollte W hingegen eher klein gewählt werden, um eine Auswahl von E/A-intensiven Ausführungsplänen zu vermeiden.

Die Kosten für unseren Ansatz können mit dieser Formel jedoch nicht abgeschätzt werden, da nicht auf Basistabellen, sondern auf abstrakte Tabellen zugegriffen wird, welche wiederum durch einen oder mehrere Funktionsaufrufe aufgebaut werden. Somit ersetzen wir

die Funktion *systemcalls* mit der Funktion *wrapcalls*. Diese Funktion beschreibt die Anzahl der Aufrufe an das Quellsystem bzw. den Wrapper, die benötigt wird, um den spezifizierten Inhalt der abstrakten Tabelle T_a zu ermitteln. Des Weiteren führen wir die Funktion *datacomm* ein, welche die Kommunikationskosten für den Transport der Ergebniszeilen vom Quellsystem zum FDBS charakterisiert. Zudem wird der physische Zugriff auf sekundären Speicher durch die Anzahl der ausgeführten Funktionen im Quellsystem ersetzt. Diese Funktion wird *funcexec* genannt. Da dieser Funktionsaufruf dem Aufruf einer föderierten Funktion entsprechen kann und diese wiederum von der Anzahl der integrierten lokalen Funktionsaufrufe abhängig ist, wird ein Faktor A eingeführt, welcher die Anzahl dieser lokalen Funktionen anzeigt. Dabei geht man von der vereinfachten Annahme aus, dass die Ausführungszeit der föderierten Funktion linear zu der Anzahl ihrer lokalen Funktionen steigt. Faktor W kann nach wie vor genutzt werden, um bei der Optimierungsentscheidung die Art des Ausführungsplanes zu berücksichtigen. Wenn z. B. der Prozess der föderierten Funktion um einiges langsamer ist als der des FDBS, d. h., das System ist „funktionslastig“, dann sollte W eher klein gewählt werden, um die Selektion von funktionsintensiven Lösungen zu vermeiden. Aus diesen Betrachtungen ergibt sich die folgende Kostenabschätzung für den Zugriff auf eine abstrakte Tabelle T_a:

$$\begin{aligned} \mathit{Cost}(T_a) = A \times \mathit{funcexec}(T_a) + W \times \mathit{wrapcalls}(T_a) \\ + \mathit{datacomm}(T_a) \end{aligned}$$

Um die Anzahl der Funktionsaufrufe ermitteln zu können, muss der Anfrageprozessor die Restriktionen bezüglich der Kardinalität als auch des Wertebereiches der Eingabeparameter kennen (siehe Abschnitt 4.1). Viele weitere Parameter, die normalerweise in den Statistiken für den Optimierer abgelegt werden, können gar nicht oder nur für Attribute, die Eingabeparameter bezeichnen, ermittelt werden. Aus diesem Grund ist es sehr schwierig, Selektivitäten für Prädikate anzugeben, um die Größe von Zwischenergebnissen abschätzen zu können. Daher müssen Defaultwerte für Selektivitäten, wie in [18] aufgeführt, herangezogen werden.

Nach Betrachtung aller Operationen wird deutlich, dass deren Verschiebung zum Wrapper die Kosten nicht nur durch geringere Datenmengen senkt, sondern auch die Anzahl der Wrapper- und Funktionsaufrufe erheblich reduzieren kann. Nichtsdestotrotz zeigt sich, dass das Ziel, die komplette Funktionalität von SQL durch kompensierende Funktionen unterstützen zu wollen, unangemessen ist. Grund hierfür ist, dass die Zahl der benötigten kompensierenden Funktionen regelrecht explodieren kann. Ein kleines Rechenbeispiel soll dies verdeutlichen. Angenommen eine Funktion hat drei Eingabeparameter mit einer Kardinalität von lediglich zehn für jeden dieser Parameter. Wenn eine Selektion vorliegt, die jedoch den Wert für nur einen dieser Parameter spezifiziert, dann sind bereits $10 \times 10 = 100$ kompensierende Funktionsaufrufe notwendig, um die fehlenden Werte auszugleichen. Noch gravierender ist das Ergebnis, wenn gar kein Wert spezifiziert wird, denn dann erhält man sogar $10 \times 10 \times 10 = 1000$ Funktionsaufrufe. Daher ist es sinnvoll, weitere Alternativen für den Gebrauch für die Referenzierung der föderierten Funktionen mittels abstrakter Tabellen in Betracht zu ziehen, welche die große Anzahl der Funktionsaufrufe verringern. Dies führt leider auch dazu, dass sich der Benutzer der Einschränkungen oder zumindest der besonderen Handhabung abstrakter Tabellen bewusst sein muss.

Eine Alternative könnte ein erweiterter Parser sein, der bei der Zergliederung der SQL-Anfrage feststellt, ob beispielsweise für alle Eingabeparameter Werte spezifiziert wurden. Ist dies nicht der Fall, so wird die Anfrage mit einer Fehlermeldung zurückgewiesen und der Benutzer auf die fehlenden Informationen aufmerksam gemacht. Auf diese Weise könnten alle kompensierenden Funktionsaufrufe aufgrund fehlender Eingabewerte vermieden werden. Ein anderer Ansatz verschiebt die Spezifikation der Eingabewerte von Vergleichsprädikaten zu benutzerdefinierten Funktionen in der WHERE-Klausel, die immer zusammen mit den referenzierten abstrakten Tabellen eingesetzt werden müssen. Mit Hilfe der benutzerdefinierten Funktionen können die benötigten Eingabewerte übergeben werden, so dass auch hier keine Kompensationsfunktionen aufgrund fehlender Werte nötig sind. Ein Nachteil bei dieser Vorgehensweise ist jedoch, dass der Benutzer wissen muss, bei welchen Tabellen es sich um abstrakte Tabellen handelt und welche benutzerdefinierte Funktion er in diesem Zusammenhang einzusetzen hat. Die Entscheidung, welche Alternative anzustreben ist, hängt sicherlich von den Systemen und den Benutzern ab.

5 Ausführungsmodell

Nachdem in den letzten Abschnitten aufgezeigt wurde, wie die Anbindung des KIF an das FDBS umgesetzt werden kann und welche Funktionalität im Wrapper sinnvoll ist, bleibt noch zu untersuchen, wie das Ausführungsmodell zu dem in Abschnitt 2 eingeführten Abbildungsmodell aussehen kann. Dazu wird zunächst der von uns gewählte Ansatz vorgestellt und anschließend verwandte Ansätze betrachtet.

5.1 Ausführungskomponente zur Funktionsintegration

Wie beschrieben, kann die Funktionsabbildung mittels eines Abhängigkeitsgraphen dargestellt werden. Bei der Wahl einer geeigneten Ausführungskomponente soll auf existierende Technologien in bereits verfügbaren Produkten zurückgegriffen werden. In unserem Fall sind somit all jene Technologien interessant, welche die Abar-

beitung von gerichteten Graphen, also definierten Abläufen unterstützen. Dazu gehören z. B. Workflow-Systeme und Message Broker. Bei anderen Ansätzen ist einiges an Entwicklungsaufwand zu investieren, um eigene Lösungen zu implementieren. Dies kann beispielsweise auf Basis eines Applikations-Servers geschehen. Aus der in Abschnitt 2 eingeführten Abbildungssprache können für alle Alternativen zumindest Eingaben (wie z. B. Konfigurationsvorgaben) für existierende Produkte oder auch ganze Code-Module für selbst entwickelte Lösungen mit Hilfe von XSLT generiert werden.

Der erste Prototyp setzt auf einem Workflow-Managementsystem (WfMS) auf, wobei die folgenden Aspekte ausschlaggebend für die getroffene Wahl sind:

– Das Schema eines Workflow-Prozesses entspricht dem Konzept unseres Abhängigkeitsgraphen. Beide Ansätze arbeiten gerichtete Graphen ab, wobei die Aktivitäten des Workflows den lokalen Funktionen unserer Integrationslösung entsprechen, der Workflow-Datenfluss den Abhängigkeiten und der Workflow-Kontrollfluss der Funktionsausführungsreihenfolge.
– Mit einem WfMS können ohne jegliche Modifikation des FDBS sehr komplexe Abbildungsszenarien umgesetzt werden.
– Ein WfMS ermöglicht eine generische Realisierung föderierter Funktionen und bietet dem FDBS transparenten Zugriff zu heterogenen Plattformen.
– Es verfügt über Schnittstellen zu vielen der zu integrierenden Quellsysteme und realisiert somit eine verteilte Programmierung über heterogene Anwendungen hinweg.
– Über die Build-Time-Komponente ist die Wartbarkeit relativ einfach, so dass Änderungen leicht eingebracht werden können. Die von den meisten Produkten unterstützte Visualisierung des Workflow-Prozesses als Graphen erleichtert dem Benutzer die Definition der Abbildung.

Der von uns eingesetzte Workflow ist ein so genannter *Production Workflow*, der im Gegensatz zum traditionellen Verständnis des Workflows mit Benutzerinteraktion und Dokumenten (*people-driven workflow*) den automatisierten Ablauf von Anwendungsaufrufen darstellt [13]. Eben dieser traditionelle Einsatz von WfMS lässt im ersten Augenblick an der Umsetzbarkeit dieser Idee zweifeln. Es kommen unweigerlich Fragen bezüglich der Angemessenheit als auch der Performanz unserer Lösung auf. Insbesondere auf den letzten Punkt wollen wir in Abschnitt 6 näher eingehen.

Weiterhin ist von Interesse, welchen Aufwand zur Umsetzung solch eine kombinierte Lösung mit sich bringt. Da die Unterstützung von Wrappern seitens der Hersteller noch nicht gegeben ist, soll hier die Kopplung auf Basis benutzerdefinierter Tabellenfunktionen beschrieben werden. Betrachtet man die Architektur von der globalen Anfrage ausgehend, so muss zur Kopplung von FDBS und WfMS je eine benutzerdefinierte Tabellenfunktion pro föderierter Funktion, also pro Workflow-Prozess implementiert werden. Deren Aufgabe ist, den Workflow-Container mit den von dem FDBS übergegebenen Eingabeparametern zu füllen, den entsprechenden Workflow-Prozess zu starten und die Ergebnisse zeilenweise an das FDBS zurückzugeben. Die föderierte Funktion und damit der Workflow-Prozess kann direkt mittels der Build-Time-Komponente des WfMS definiert werden. Normalerweise legt jedes Workflow-System seine Prozessdefinitionen in einem ASCII-Format ab, so dass dieses mittels XSLT in das vorgestellte XML-Format der Funktionsabbildung überführt werden kann. Eine solche XSLT-Abbildung muss definiert werden. Abschließend erfolgt der Zugriff des WfMS auf die zu integrierenden lokalen Funktionen. Da das von uns eingesetzte WfMS-Produkt lediglich ausführbare Programme (*executables*) ansprechen kann, muss je nach Beschaffenheit des lokalen Systems und dessen Funktionen ein Adapter erstellt werden. Dieser Adapter ist ein Programm, das die Eingabeparameter aus einem Workflow-Container ausliest, die lokale Funktion ausführt und anschließend die Ausgabeparameter in den Workflow-Container zurückschreibt. Will man diese Adapter sehr klein halten, so muss pro lokaler Funktion ein Adapter erstellt werden. Baut man ihn dagegen in einer generischen Form, so reicht ein Adapter pro lokalem System aus, der Informationen über die verfügbaren lokalen Funktionen enthält.

5.2 Verwandte Ansätze zur Funktionsintegration

Betrachtet man verwandte Ansätze auf dem Gebiet der Integration, so lassen sich diese in unterschiedliche Kategorien einteilen. Zum einen kann man grundsätzlich zwischen Ansätzen in der Wissenschaft und solchen in der Industrie unterscheiden. Zum anderen findet man Ansätze, die sich primär um die Integration von Daten bemühen, während andere auf die Integration von Funktionen bzw. APIs fokussieren. Die Kombination beider Ansätze wird nur selten behandelt. Da die Datenintegration ein bereits ausgiebig diskutiertes Thema darstellt, konzentrieren wir uns zunächst auf die Funktionsintegration. Ein industrieller Einsatz einer solchen Middleware ist nur mit Hilfe von ausgereiften, kommerziell verfügbaren Lösungen denkbar. Daher sollen vor allem industrielle Ansätze näher beleuchtet werden.

Auf der industriellen Seite erweisen sich drei Ansätze als interessante Lösungen zur Integration von Anwendungssystemen, die diskutiert werden sollen: der Einsatz von Message Brokern, von J2EE Application Servern sowie den derzeitig sehr populären Web Services. Jede dieser Lösungen wird bezüglich ihrer Vorgehensweise kurz umrissen und anschließend bewertet.

5.2.1 Message Broker Message Broker [16] basieren auf nachrichtenorientierter Middleware (*message-oriented*

middleware, MOM) – oder besser bekannt als Message-Queueing-Systeme –, die eine asynchrone Kommunikation über Nachrichten zwischen Systemen ermöglicht. Diese Systeme können unterschiedlichsten Typs sein, wie z. B. DBMS, Legacy-, TP- oder PDM-Systeme. Der Message Broker stellt eine Art Vermittler dar, welcher den Transfer der Nachrichten zwischen den beteiligten Systemen verwaltet. Dabei übernimmt er mehrere Rollen: er sorgt für die richtige Verteilung der Nachrichten (*message routing*) und nimmt notwendige Transformationen der derselben vor. Außerdem unterstützt er die Definition eines Kontrollflusses der Nachrichten auf Basis eines Regelwerks. Mit Hilfe diese Regelwerks kann die Verarbeitung und Verteilung der Nachrichten über Regeln wie Bedingungsüberprüfungen, mathematischen Funktionen oder auch Datentypkonvertierungen definiert werden.

Soll nun die Abbildungslogik einer Funktionsintegration mittels eines Message Brokers implementiert werden, so muss diese Logik mit Hilfe des vom Message Broker unterstützten Regelwerks erfolgen. Handelt es sich um ein mächtiges Regelwerk, so können hinter den Werten eines dedizierten Feldes der Nachricht unterschiedliche Kontrollflüsse hinterlegt werden, um Nachrichten für die einzelnen Anwendungssysteme zu generieren. Kann das Regelwerk nur einfache Flüsse unterstützen, so kann die Information darüber, welche Nachrichten generiert werden müssen, beispielsweise in einer Datenbank abgelegt werden, die als Teil des Kontrollflusses abgefragt wird. Die über Nachrichten zurückgemeldeten Ergebnisse werden vom Message Broker zusammengeführt (Transformationen) und dem aufrufenden System zur Verfügung gestellt.

Im Vergleich zu dem vorgestellten Ansatz auf Basis eines Workflow-Systems ist festzustellen, dass die Abbildungsmächtigkeit eines Message Brokers bei weitem nicht an die des Workflow-Systems heranreicht. Werden die Abbildungen komplexer, so können sie nicht mehr ausreichend durch das Regelwerk des Message Brokers unterstützt werden. Der Einsatz eines Message Brokers ist in solchen Fällen zu empfehlen, in welchen vielmehr die Verbindung zwischen mehreren Systemen als deren Integration im Vordergrund steht. In unserem Fall muss man jedoch von komplexen Abbildungen ausgehen, so dass der Einsatz eines Message Brokers nicht ausreichend ist.

5.2.2 J2EE Application Server Ein Integrationsansatz auf Basis von J2EE [20] stützt sich auf einen Standard und verringert die Abhängigkeit von einem bestimmten Hersteller. Die J2EE-Architektur verkörpert einen Mehr-Schichten-Ansatz, bei welchem zwischen Präsentation, Geschäftslogik und Datenhaltung unterschieden wird. Für die Funktionsintegration ist die Schicht mit der Geschäftslogik relevant, da hier die Abbildung der föderierten Funktionen auf die lokalen Funktionen mit Hilfe von EJBs (*Enterprise JavaBeans*) implementiert wird, die von den Anwendungssystemen in der Datenhaltungsschicht zur Verfügung gestellt werden. Im Gegensatz zu den bisher vorgestellten Ansätzen muss die Abbildungslogik vollständig selbst implementiert werden, d. h., man kann nicht auf bestehende Engines zurückgreifen, wie dies bei Workflow-Systemen oder Message Brokern der Fall ist.

Die Umsetzung einer Funktionsintegration auf Basis eines J2EE-Applikations-Servers stellt sich grob umrissen folgendermaßen dar. Zunächst wird der Zugriff auf die Daten in Form von Entity Beans ermöglicht. Dieser Zugriff kann entweder direkt auf die Datenbank erfolgen oder über existierende APIs und Wrapper stattfinden. Die Abbildungslogik selbst, d. h. die Festlegung, welche Systeme in welcher Reihenfolge aufgerufen werden und welche Abhängigkeiten zwischen ihnen bestehen, muss mit Session Beans implementiert werden. Auf diese Weise wird jede föderierte Funktion durch eine Session Bean ausgeführt. Zur Standardisierung der Zugriffschnittstelle auf die zu integrierenden Systeme wurde die *J2EE Connector Architecture* definiert. Diese Architektur soll die Integration vereinfachen, indem vor allem der Zugriff auf die Systeme standardisiert ist und somit das Hinzufügen von neuen Systemen erleichtert wird. Dazu gehören u. a. Funktionalitäten wie Verbindungsverwaltung, Transaktionsverwaltung und Sicherheitsverwaltung.

Bei diesem Ansatz ist vor allem positiv zu bewerten, dass er sich auf Standards stützt. Des Weiteren sollte es möglich sein, alle denkbaren Integrations- und Abbildungsszenarien umzusetzen, da man sie direkt mit Java implementiert und somit die ganze Mächtigkeit dieser Programmiersprache einsetzen kann. Nichtsdestotrotz muss man aber die Logik von Hand implementieren, was zu altbekannten Problemen führen kann, wie mangelnde Dokumentation und die erschwerte Wartung und Erweiterung der bestehenden Integration durch Wegnahme oder Hinzufügen von Systemen. Der Einsatz eines J2EE-Ansatzes scheint nur dann sinnvoll, wenn die umzusetzende Logik sich nicht mit anderen Systemen realisieren lässt. Dabei sollte aber soviel wie möglich des Programmiercodes mit Hilfe der beschriebenen Abbildungssprache generiert werden, um zumindest eine ausreichende Dokumentation sicherzustellen.

5.2.3 Web Services Ein weiterer Ansatz, der vor allem in den letzten beiden Jahre große Beachtung gefunden hat, sind Web Services. Mit Hilfe von Web Services möchte man die Integration von Systemen nicht nur innerhalb einer Abteilung oder einer Organisation, sondern über Unternehmensgrenzen hinweg ermöglichen. Das Web erlaubt es auf einfachem Weg, dass Systeme ihre Funktionalität, oder auch Services, zur Verfügung stellen. Web Services basieren auf offenen Standards, wie z. B. SOAP, und nehmen Anfragen anderer Systeme entgegen. Diese Anfragen werden über eine leichtgewichtige und herstellerunabhängige Kommunikationstechnologie verschickt.

Übertragen auf die angestrebte Funktionsintegration bedeutet dies, dass die Anwendungssysteme ihre Funktionen als Web Services anbieten. Diese werden mit Hilfe der Sprache WSDL (*Web Services Description Language*, [23]) beschrieben und in einer Registry abgelegt, dem UDDI (*Universal Description, Discovery, and Integration*). Bei der Umsetzung der Abbildungslogik können die Web Services jedoch nicht weiterhelfen. Diese Logik muss in einer separaten Komponente implementiert werden und die föderierten Funktionen werden als weitere Web Services angeboten. Der Aufruf einer föderierten Funktion gestaltet sich dann folgendermaßen. Eine Applikation möchte eine föderierte Funktion aufrufen und sucht diese zunächst in der UDDI Registry. Hat sie die gesuchte Funktion gefunden, so ruft sie in der Integrationskomponente den entsprechenden Service auf. Dieser Service wiederum bedient sich all jener von den Anwendungssystemen bereitgestellten Web Services, die er für die Abbildung der föderierten Funktion benötigt. Das Endergebnis wird an die anfragende Komponente zurückgegeben.

Der große Vorteil dieses Ansatzes ist sicherlich darin zu sehen, dass eine Verbindung über Unternehmensgrenzen hinweg möglich ist und die bisherigen Probleme wie beispielsweise Firewalls umgeht. Web Services ermöglichen aber in erster Linie eine gemeinsame Sprache zwischen den Systemen, aber keineswegs eine Integration, wie sie in diesem Aufsatz beschrieben wurde. Denn auch hier ist wieder eine spezielle Komponente für die Abbildungslogik zu finden, sei es selbst implementiert oder durch Einsatz anderer bereits besprochener Technologien wie Workflow-Systeme und Message Broker.

5.2.4 Abschließende Bemerkungen Die aufgeführten Lösungsansätze unterstützen in erster Linie die Integration von Anwendungssystemen, nicht aber die Integration von Datenbanksystemen auf einer reinen Datenebene. Das bedeutet, dass eine Verarbeitung der zu integrierenden Daten in der Form, wie man es von Datenbanksystemen kennt, nicht möglich ist. Da aber genau dies eine der gestellten Anforderungen darstellt, ist der Einsatz von Technologien allein zur Funktionsintegration nicht ausreichend. Der Zugriff auf Daten als auch Funktionen kann zwar unterstützt werden, aber die effiziente Weiterverarbeitung der Daten ist nicht sichergestellt. Daher scheint der Einsatz eines Datenbanksystems – in unserem Fall in Form eines föderierten Datenbanksystems – unvermeidlich. Letztendlich sollen Daten integriert und verarbeitet werden, auch wenn diese nicht direkt, sondern nur über Funktionen abrufbar sind.

6 Empirische Untersuchungen

Um ein Gefühl für die Umsetzbarkeit unseres Lösungsansatzes zu bekommen, wurden zwei Prototypen implementiert, wovon der eine die Integrationsarchitektur mit dem WfMS realisiert. Der zweite Prototyp ersetzt das WfMS durch die direkte Anbindung der einzelnen lokalen Funktionen an das FDBS, wobei die eigentliche Integration der Funktionen innerhalb des FDBS verwirklicht wird. Auf beiden Prototypen wurden mehrere, unterschiedlich komplexe Funktionsabbildungen implementiert, damit der Performanzvergleich der Ergebnisse einen ersten Eindruck des WfMS-Mehraufwands vermitteln
kann. Da die Workflow-Komponente nur für die Funktionsintegration eingesetzt wird, wurden globale Anfragen gemessen, die nur auf föderierte Funktionen zugreifen. Obwohl in Abschnitt 3 die Wrapper-Anbindung favorisiert wurde, erfolgte die Kopplung bei beiden Prototypen mangels Wrapper-Unterstützung seitens der Datenbankhersteller mit benutzerdefinierten Tabellenfunktionen. Da auf Basis der benutzerdefinierten Tabellenfunktionen nur lesender Zugriff implementiert werden kein, wurden keine Schreibzugriffe und kein Mehrbenutzerbetrieb getestet.

Die realisierten Architekturen basieren auf DB2 UDB V7.1 [6] und MQ Series Workflow V3.2.2 [7] von IBM. Beim Workflow-Prototyp erfolgt die Funktionsintegration durch das WfMS und die daraus resultierenden föderierten Funktionen werden mittels benutzerdefinierten Tabellenfunktionen dem FDBS zur Verfügung gestellt (siehe Abb. 4). Aus Sicherheitsgründen erlaubt es DB2 nicht, dass ein Datenbankanwendungsprozess E/A-Operationen mit anderen Systemen als dem DBS selbst vornimmt. Der Grund hierfür ist, dass der Prozess Zugang zum kompletten Adressraum des DBS hat und er somit gezielt oder unbeabsichtigt Daten des DBS unkontrolliert modifizieren kann. Aus diesem Grund kann die Tabellenfunktion das WfMS nicht direkt starten. Um dieses Problem zu umgehen, wurde ein Controller zur Entkopplung dieser beiden Prozesse eingeführt.

Beim zweiten Prototyp werden zunächst alle lokalen Funktionen direkt über so genannte Zugriffstabellenfunktionen (*access user-defined table function*, A-UDTF) an das FDBS angebunden. Auch hier war ein Controller einzufügen, da die Anwendungssysteme in unserem Testszenario durch Java-Programme mit JDBC-Zugriff auf Daten simuliert wurden. Da DB2 den Zugriff von Tabellenfunktionen auf Datenbanken per JDBC nicht unterstützt, musste hier erneut eine Entkopplung von Prozessen stattfinden. Die Integrationslogik, die im Workflow-Prototyp durch das WfMS bereitgestellt wird, wird über so genannte Integrationstabellenfunktionen (*integration UDTF*, I-UDTF) innerhalb des FDBS implementiert. In diesen Integrationstabellenfunktionen werden die Zugriffstabellenfunktionen in SELECT-Anweisungen referenziert und auf diese Weise die benötigten Daten abgefragt und weiterverarbeitet.

Die Messungen der Antwortzeiten für die Ausführung der unterschiedlichen föderierten Funktionen auf beiden Prototypen sind in Abb. 5 dargestellt und zeigen, dass beim direkten Vergleich die Workflow-Lösung bis zu drei-

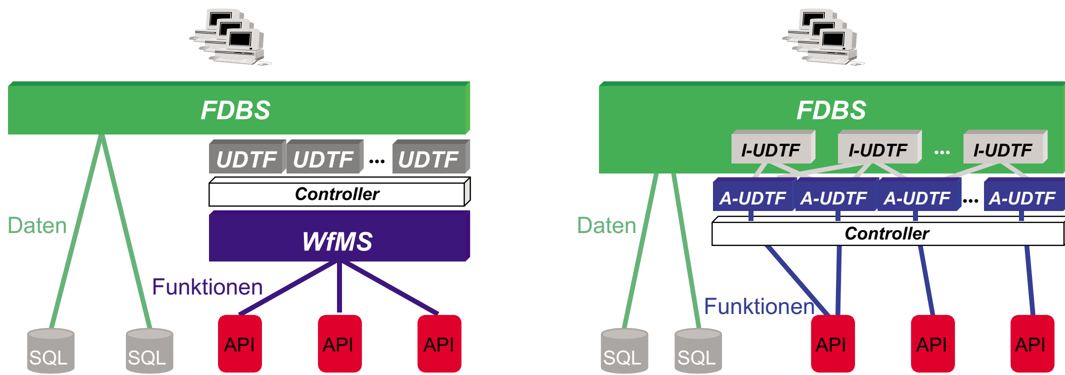

Abb. 4 Die Prototyp-Architekturen mit und ohne WfMS.

mal langsamer ist als die Alternativlösung. Des Weiteren sieht man, dass bei ansteigender Anzahl der lokalen Funktionen die Ausführungszeiten des Workflow-Ansatzes schneller steigen. Bedenkt man jedoch, dass hier ein zweites, sehr großes Softwarepaket in der Architektur anzubinden war, so ist der Zeitunterschied zwischen den Prototypen nicht übermäßig groß.

Ein anderer interessanter Aspekt ist die Aufteilung der benötigten Zeit in einzelne Phasen. Abb. 6 verdeutlicht, wieviel Zeit die einzelnen Schritte bei der föderierten Funktion `GetNoSuppComp` benötigen. Bei der Workflow-Lösung nimmt die Ausführung der Aktivitäten mit 51% den größten Anteil an der Gesamtzeit in Anspruch. Weitere 23% werden benötigt, um vor dem WfMS-Einsatz das Aufrufen und Ausführen der Tabellenfunktionen und des Controllers vorzubereiten. Auffällig ist, dass das Starten der Workflow-Prozessinstanz sowie die Java-Umgebung für das Java-API des WfMS mit 10% relativ viel Zeit in Anspruch nehmen. Dieser relative Anteil wird jedoch kleiner, wenn die Anzahl der zu integrierenden Funktionen steigt, da seine Dauer konstant bleibt – unabhängig von den ausgeführten Aktivitäten.

Betrachtet man die Alternativarchitektur, so ergeben sich andere Verhältnisse. Das Starten und Beenden der Integrationstabellenfunktion beansprucht 20% der Zeit. Dies entspricht ungefähr dem Workflow-Ansatz mit 22% für Starten (9%), Ausführen (11%) und Beenden (2%) der Tabellenfunktion. Die Ausführungsdauer der drei Zugriffstabellenfunktionen beläuft sich auf 49% der Gesamtlaufzeit, während die eigentliche Ausführung der lokalen Funktionen lediglich 6% der Zeit benötigt. Auffallend bei dieser Architektur ist, dass die Controller insgesamt 25% der Zeit beanspruchen. An dieser Stelle sind somit Optimierungspotentiale vorhanden, wenn der Controller beispielsweise durch eine effizientere Anbindungsmaßnahme ersetzt werden kann.

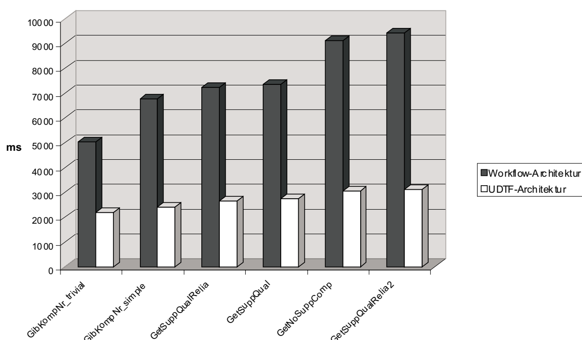

Abb. 5 Ausführungszeiten beider Prototypen.

Zusammenfassend können wir feststellen, dass ein Workflow-System nicht so viel Mehraufwand mit sich bringt wie befürchtet. Die resultierenden Ausführungszeiten bleiben in einem akzeptablen Rahmen: sie sind nur zwei- bis viermal langsamer als bei einer vergleichbaren Lösung ohne WfMS, obwohl man auf den ersten Blick aufgrund des Einsatzes zweier Server sicherlich einen größeren Unterschied erwarten würde. Nichtsdestotrotz möchten wir darauf hinweisen, dass die Performanz nicht das wichtigste Kriterium für unseren Ansatz ist. Essentielle Merkmale sind für uns die Unterstützung von komplexen Abbildungen, einfache Handhabung und vor allem die Tatsache, dass die Entwickler als auch das FDBS bezüglich verteilter Programmierung über heterogene Schnittstellen hinweg entlastet werden.

7 Zusammenfassung

In diesem Aufsatz haben wir einen Ansatz zur Integration von Datenbank- und Anwendungssystemen bzw. Daten und Funktionen vorgestellt. Ein solch kombinierter Ansatz wird zunehmend relevant, da der Zugriff auf Datenbanken immer häufiger nur über die APIs der zugehörigen Anwendung möglich ist. Mangels technischer Unterstützung erfolgt derzeit eine Art Integration dieser Systeme durch den Benutzer. Zur Unterstützung des Benutzers wurde das Konzept der föderierten Funktionen eingeführt, welches die Integration von lokalen Funktionen heterogener Anwendungssysteme umsetzt. Die vorgestellte Architektur basiert auf der Kopplung eines FDBS und einem WfMS mit Hilfe eines Wrappers. Das WfMS setzt hierbei die Integration der lokalen Funktio-

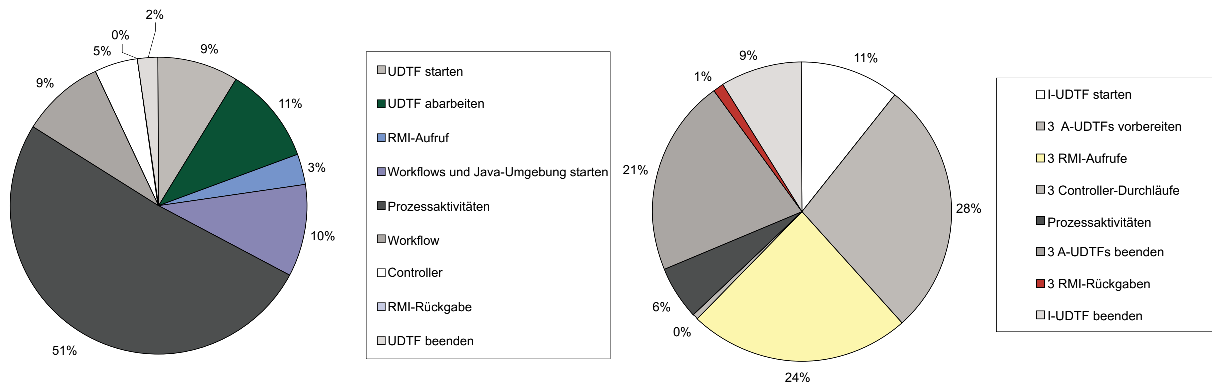

Abb. 6 Relativer Zeitbedarf der Verarbeitungsschritte beim Workflow-Ansatz (links) und beim UDTF-Ansatz (rechts).

nen um und dessen resultierende föderierte Funktionen können anschließend mittels des FDBS mit weiteren Daten aus relationalen Datenbanksystemen integriert werden. Da sämtliche Aspekte der Datenintegration in den vergangenen Jahren bereits ausgiebig behandelt wurden, lag der Fokus dieses Aufsatzes auf der Verwirklichung der Funktionsintegration. Hierzu wurde ein Ansatz entwickelt, der sich in ein Beschreibungs- und ein Ausführungsmodell aufteilt. Für das Beschreibungsmodell wurde eine Abbildungssprache auf Basis von XML entwickelt, mit deren Hilfe die Abbildung von föderierten Funktionen auf lokale Funktionen spezifiziert werden kann. Diese Abbildungssprache ist so gestaltet, dass sie einfach, leichtgewichtig und vor allem unabhängig von der zugehörigen Ausführungskomponente ist. Für unseren Prototyp wählten wir ein WfMS als Ausführungskomponente. Durch die Kopplung mit dem FDBS konnte man an SQL als Anfragesprache festhalten, in der die globalen Anfragen mit Referenzen zu Tabellen als auch Funktionen formuliert werden. Als Kopplungsmechanismus wurde das Konzept des Wrappers ausgewählt und untersucht, inwiefern zusätzliche Funktionalität innerhalb des Wrappers zur globalen Anfrageoptimierung beitragen kann. Hierbei wurde aufgezeigt, wie relationale Operationen auf Funktionen abgebildet werden und wie sich daraus resultierend das Kostenmodell ändert. Abschließend wurden die Ergebnisse eines Performanzvergleiches zweier Prototypen vorgestellt, um zu verdeutlichen, dass der Mehraufwand durch ein WfMS bedeutend geringer ist, als man zunächst annehmen würde. Auch dies trägt zur Attraktivität eines WfMS-Einsatzes zur Funktionsintegration bei.

Danksagung

Wir danken den anonymen Gutachtern für die konstruktive Kritik, welche die Lesbarkeit unseres Aufsatzes verbessern half.

Literatur

1. O. A. Bukhres, A. K. Elmagarmid (Hrsg). *Object-Oriented Multidatabase Systems – A Solution for Advanced Applications.* Prentice Hall, 1996.
2. S. Conrad. *Föderierte Datenbanksysteme – Konzepte der Datenintegration.* Springer-Verlag, 1997.
3. P. Dadam. *Verteilte Datenbanken und Client/Server-Systeme.* Springer-Verlag, 1996.
4. K. Hergula, T. Härder. *How Foreign Function Integration Conquers Heterogeneous Query Processing.* Proc. Int. Conf. on Information and Knowledge Management, Atlanta, 2001, S. 215-222.
5. A. R. Hurson, M. W. Bright, S. H. Pakzad (Hrsg). *Multidatabase Systems: An Advanced Solution for Global Information Sharing.* IEEE Computer Society Press, 1994.
6. IBM. *DB2 UDB V7.* 2001; zu beziehen über `http://www.ibm.com/db2/`
7. IBM. *MQ Series Workflow.* 2001; zu beziehen über `http://www.ibm.com/mqs/`
8. ISO. *ISO 10303 – Industrial Automation Systems and Integration: Product Data Representation and Exchange, Part 11 – Description Methods: The EXPRESS Language Reference.* 1994.
9. ISO TC184 SC4 WG11 N002. *Proposed Syntax for EXPRESS-X.* 2001.
10. ISO & ANSI. *Database Languages – SQL – Part 2: Foundation.* Internationaler Standard, 1999.
11. ISO & ANSI. *Database Languages – SQL – Part 9: Management of External Data.* Working Draft, January, 2002.
12. A. Kim (Hrsg). *Modern Database Systems – The Object Model, Interoperability, and Beyond.* Addison-Wesley, 1995.
13. F. Leymann, D. Roller. *Production Workflow: Concepts and Techniques.* Prentice Hall, 2000.
14. F. d. F. Rezende, K. Hergula. *The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways.* Proc. 24th Int. Conf. on Very Large Data Bases, New York, 1998, S. 146-157.
15. SAP AG. *Das R/3 System.* 2001; zu beziehen über `http://www.sap-ag.de/products/r3/`
16. R. Schulte. *Message Brokers: A Focussed Approach to Application Integration.* Gartner Group, Strategic Analysis Report SSA R-401-102, 1996.
17. SDRC Corp.. *Metaphase.* 2001; zu beziehen über `http://www.metaphasetech.com`

18. P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price. *Access Path Selection in a Relational Database Management System.* Proc. ACM SIGMOD Conf. on Management of Data, Boston, 1979, S. 251-260.
19. A. P. Sheth, J. A. Larson. *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Computing Surveys 22(3), 1990, S. 183-236.
20. Sun Microsystems Inc. *Java 2 Platform Enterprise Edition.* 2002; zu beziehen über `http://java.sun.com/j2ee/`
21. M. Tork Roth, P. Schwarz. *Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources.* Proc. 24th Int. Conf. on Very Large Data Bases, Athen, 1997, S. 266-275.
22. Workflow Management Coalition. Standards zu beziehen über `http://www.wfmc.org`
23. World Wide Web Consortium. *Extensible Markup Language (XML) 1.0.* W3C Recommendation, 1998; XML und verwandte Standards zu beziehen über `http://www.w3c.org`