Generierte DB-Aufrufschnittstellen
–
Anwendungsspezifische Zugriffsoptimierung durch Bindungsflexibilität

Theo Härder, Udo Nink, Norbert Ritter

Universität Kaiserslautern
Postfach 3049, 67653 Kaiserslautern
e-mail: haerder/nink/ritter@informatik.uni-kl.de

Zusammenfassung. Jede Datenbankprogrammierschnittstelle muß sich daran messen lassen, inwieweit sie die Vorteile der durch sie integrierten Sprachen (Datenbank- und Programmiersprache) erhält. Neben dieser allgemeinen Zielsetzung müssen heutzutage neuere Konzepte in den Bereichen Programmiersprachen und Datenbanksysteme sowie die Anforderungen komplexer Anwendungsbereiche (z. B. CAD oder Software-Entwicklung) beachtet werden. Unser Ansatz der generierten Aufrufschnittstellen soll Datenmodelle objekt-relationaler Datenbanksysteme und Typsysteme objektorientierter Programmiersprachen vor allem durch Ausnutzung erweiterter Modellierungskonzepte, wie abstrakte Datentypen und Referenzen, näher zusammenrücken. Er erlaubt neben der Schnittstellengenerierung ihre anwendungsspezifische Konfigurierung, um die DB-Verarbeitung spezieller Anwendungen gezielt optimieren zu können. Wir werden neben den Aspekten der Generierung von Aufrufschnittstellen auch Experimente beschreiben, die den durch Konfigurierung erzielbaren Leistungsgewinn der DB-Operationen (und damit der gesamten Anwendung) verdeutlichen.

Schlüsselwörter. Datenbanksysteme, Objektorientierung, Bindungszeitpunkte, Anwendungsprogrammierschnittstelle, Generische Methoden

Abstract. Application programming interfaces (API) for database systems should preserve both the strengths of the database management system and of the programming languages. In addition to this general objective, innovative concepts of both domains as well as reqirements of advanced application areas like CAD and software development must be observed. By using extended data modeling concepts such as abstract data types and references, our approach provides generated call-level interfaces thereby bringing data models of object-relational database systems and type systems of object-oriented programming languages closer together. Generated interfaces may be configured in an application-specific way in order to selectively optimize DB processing of given applications. We will describe all aspects of generating call-level interfaces as well as characteristic experiments which reveal the performance gain of DB operations (and, in turn, of the entire application) to be attributed to configuration mechanisms.

Key words. Database systems, object orientation, binding time, application programming interface, generative methods

1 Einleitung

Während Datenbanksysteme große Mengen gemeinsam genutzter Daten verwalten, verarbeiten Anwendungsprogramme diese Daten im Rahmen betrieblicher Abläufe. Oft sind verschiedene Anwendungsprogramme in unterschiedlichen Programmiersprachen (PL) geschrieben, die wiederum verschiedenartige Typsysteme besitzen. Aus diesem Grund sollten Datenbankverwaltungssysteme (DBVS) mehrsprachenfähig (*multi-lingual)* sein. Diese Eigenschaft wird dadurch erreicht, daß ein DBVS und damit auch die ihm zugrundeliegende DB-Sprache (DBL) mit einem unabhängigen Typsystem ausgestattet ist. Zum Zugriff auf eine Datenbank (DB) wird eine Anbindung der DBL an die entsprechende PL benötigt, die auch als Anwendungsprogrammierschnittstelle (kurz *API* für engl. *application programming interface*) [7, 11, 13, 16, 17] bezeichnet wird.

Abb. 1 illustriert eine grobe Sicht auf die bereits 1983 in [11] veröffentliche Klassifikation verschiedener Ansätze zur Anbindung von PL und DBL. Im unteren Teil der Abbildung ist das lauffähige (d. h. bereits übersetzte und gebundene) Anwendungsprogramm dargestellt. Es setzt sich aus dem Objektcode des eigentlichen Anwendungsmoduls (application.o) sowie den Laufzeitsystemen der Schnittstel-

le selbst (libAPI.a) und des verwendeten DBVS (libDBMS.a) zusammen. Die Abbildung macht deutlich, daß sich die verschiedenen dargestellten Ansätze in diesem Punkt nicht unterscheiden. Die Unterschiede ergeben sich vielmehr aus der Art bzw. dem Grad der Integration von DBL-Operationen[1] in PL-Anweisungen und die zu treffenden Maßnahmen zur Übersetzung und zum Binden des Quellcodes (des Anwendungsprogramms).

Die Aufrufschnittstelle (kurz CLI für engl. *call-level interface*) ist dadurch charakterisiert, daß DBL-Anweisungen von vornherein als (String-)Parameter von Anweisungen der Wirtsprache betrachtet werden und von entsprechenden (Wirtssprachen-)Prozeduren, die in der Regel in eine Bibliothek zur Verfügung stehen, an die DBVS-Schnittstelle weitergeleitet werden.

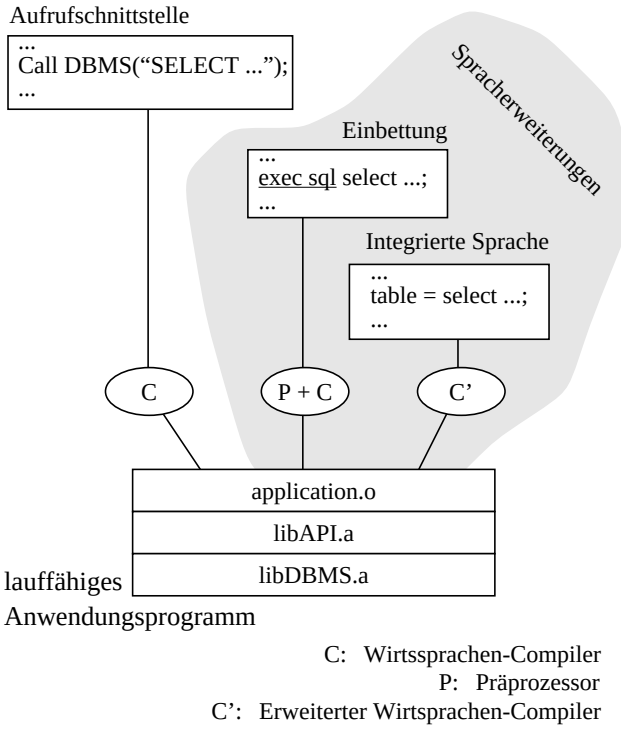

Abb. 1: Grobklassifikation von API

Während im Falle der Einbettung DBL-Anfragen ansatzspezifisch zu markieren sind (im Beispiel ist die Marke *exec sql* vorgestellt), können bei einer integrierten Sprache DBL-Anweisung direkt in PL-Anweisungen eingeflochten werden, so daß eine gewisse Orthogonalität und damit ein höherer Integrationsgrad erreicht wird. Üblicherweise geschieht die Übersetzung eines Anwendungsprogramm im Falle der Einbettung durch Vorschalten eines Präprozessorlaufs vor den Lauf des eigentlichen Wirtsprachen-Compilers. Der Präprozessor kann auf einfache Weise die eingebetteten DBL-Anweisungen anhand der vom Anwendungsprogrammierer eingefügten Markierungen erkennen und in Aufrufe an das API-Laufzeitsystem umsetzen. Diese Aufrufe liegen dann in der Wirtsprache vor, so daß das Ergebnis des Präprozessorlaufs mit dem (unveränderten) Wirtssprachen-Compiler übersetzt und anschließend gebunden werden kann. Natürlich besteht nicht die Notwendigkeit, die markierten DBL-Anweisungen durch einen Präprozessor bearbeiten zu lassen; prinzipiell kann man zur (einstufigen) Übersetzung auch einen modifizierten Wirtssprachen-Compiler heranziehen, was jedoch aufgrund der Abhängigkeit von der Compiler-Erweiterung kostspielig ist und deshalb nur selten gemacht wird. Aufgrund der Tiefe der Integration ist jedoch eine Erweiterung des Wirtssprachen-Compilers bei der integrierten Sprache vorzuziehen, obwohl auch hier prinzipiell die Möglichkeit des Einsatzes eines Präprozessors denkbar ist, der jedoch natürlich 'intelligenter' sein muß als im Falle der Einbettung. Offensichtlich gehören die Ansätze der Einbettung und der integrierten Sprache zu einem Spektrum, das sich entlang der möglichen Integrationsgrade ergibt. Wir fassen diese Ansätze daher unter dem Begriff *Spracherweiterungen* zusammen.

Anwendungsprogrammierer, als die direkten Nutzer von API, scheinen einfache, auf wohlbekannten Konzepten beruhende Schnittstellen zu bevorzugen, denn Aufrufschnittstellen (kurz CLI) wurden immer am besten *akzeptiert*, da sie zusammen mit Standard-Compilern benutzt werden können. Die Entwickler von API hingegen scheinen eleganteren Lösungen den Vorzug zu geben, wie z. B. Spracherweiterungen, die jedoch, wie oben bereits erläutert, in aller Regel Compiler-Erweiterungen bzw. Präprozessoren erfordern.

Neben der Akzeptanz kann eine Bewertung dieser bislang existierenden Ansätze der Anbindung von DBL und PL nun den verschiedensten Kriterien folgen, wie z. B. Effizienz im allgemeinen, Möglichkeiten der Fehlerbehandlung und der Zugriffskontrolle, Aufwand für den Anwendungsentwurf (Einarbeitung und Kodierung), Aufwand für die Bereitstellung der API-Funktionen selbst, Aufwand für die Übersetzung des Anwendungsprogramms und Datenunabhängigkeit [17]. Wir wollen uns hier auf die wesentlich-

1. Wir benutzten die Begriffe *DBL-Operation* und *DB-Anfrage* synonym und verstehen darunter sowohl Retrieval- als auch Einfüge-, Lösch- und Änderungsoperationen.

sten Kriterien (*Effizienz/Fehlerbehandlung und Datenunabhängigkeit*) und deren Wechselspiel beschränken.

Bezüglich der *Effizienz* ist zunächst allgemein zu sagen, daß der Übersetzungsaufwand sich gegenläufig zum Laufzeitaufwand verhält. Aus Effizienzgesichtpunkten lohnt es sich daher, soviel wie möglich der 'durchzuführenden Arbeit' in die Übersetzungsphase vorzuziehen, um damit die Laufzeit zu verkürzen. Bei Aufrufschnittstellen steht die alleinige Nutzung eines (unveränderten) Wirtssprachen-Compilers im Vordergrund, da, wie bereits gesagt, die API als wirtssprachenabhängige Bibliothek zur Verfügung steht. Dies macht deutlich, daß in diesem Ansatz keine Veranlassung besteht, mit dem DBVS zu kommunizieren und bereits zur Übersetzungszeit vorbereitende Maßnahmen für die Ausführung der im Programm enthaltenen DBL-Anweisungen zu treffen[2]. Da Spracherweiterungen schon aufgrund ihres Ansatzes angepaßte Maßnahmen zur Übersetzung erfordern, liegt eine Kommunikation mit dem DBVS zur Übersetzungszeit des Anwendungsprogrammes näher, so daß hier oft eine höhere Effizienz der Ausführung als bei Aufrufschnittstellen erreicht wird. Solche vorbereitenden Maßnahmen sind natürlich nur möglich, wenn das DBVS diese Vorgehensweise an seiner Schnittstelle unterstützt. Ähnlich der Effizienz ist die *Fehlerbehandlung* zu betrachten. Eine frühe (zur Übersetzungszeit) Fehlerkennung (aufgrund strenger Typprüfung) stabilisiert die durchzuführenden Laufzeitaktionen.

Der Aspekt der *Datenunabhängigkeit* verhält sich in dem Sinne diametral zu dem der Effizienz, daß durch frühe Bindung (Datenabhängigkeit) der eigentliche Leistungsgewinn erzielt wird. Bereits zur Übersetzungszeit getroffene Entscheidungen (z. B. hinsichtlich der Art und Weise des Zugriffs auf konkrete Daten) lassen sich zur Ausführungszeit nicht ohne größeren Aufwand revidieren. Werden zwischenzeitlich DB-Schemaänderungen durchgeführt, die solche Bindungsentscheidungen betreffen, so müssen die entsprechenden DB-Anfragen invalidiert und durch eine erneute Übersetzung und Bindung dem aktuellen Schema angepaßt werden.

Gehen wir davon aus, daß ein CLI gemäß der ursprünglichen Intension (Unterstützung von Flexibilität durch die Möglichkeit des dynamischen Erzeugens von DBL-Anweisungen) nur zur Laufzeit mit dem DBVS kommuniziert, so ergibt sich folgende Bewertung. Während CLI weniger effizient sind, aber eine höhere Datenunabhängigkeit bieten, verhalten sich Einbettungen genau umgekehrt. Die anderen, weiter oben genannten Bewertungskriterien können wie folgt eingeordnet werden: Fehlerbehandlung, Zugriffskontrolle und Kodierungsaufwand fallen in die gleiche Gruppe wie die Effizienz, wirken sich also eher nachteilig bei CLI und vorteilhaft bei Einbettungen aus. Bei Einarbeitungs-, Realisierungs- und Übersetzungsaufwand dagegen kehren sich diese Bewertungen um [17].

Wir versuchen durch einen Generierungsansatz, der die Ausnutzung von DB-Schemainformation, DBL-Anfragen und Benutzerwissen in selektiver Weise unterstützt, die Vorteile von CLI und Einbettungen zu vereinen. Der Anwendungsprogrammierer soll dabei entscheiden können, ob für einzelne DBL-Anweisungen Datenunabhängigkeit oder Effizienz der Ausführung bevorzugt wird; gemäß seiner Vorgaben sollen entsprechende API-Funktionen generiert bzw. konfiguriert werden.

Neben dem angesprochenen Problem des diametralen Verhaltens der Bewertungskriterien für Anbindungen zwingt uns die im folgenden angesprochene kontinuierliche Angleichung der Datenmodelle neuerer DBVS an die Datenmodelle von objektorientierten PL (OOPL) zu einem Überdenken der Anbindungsmechanismen. Die oben angesprochene Klassifikation kann vor dem Hintergrund der Anbindung von imperativen PL an relationale DBVS gesehen werden. Dabei standen die Überbrückung der Kluft zwischen den komplexen Datenstrukturen der PL und den einfachen Strukturen des Relationenmodells sowie das Problem des sog. *impedance mismatch* (Verarbeitung der Ergebnisse mengenorientierter DB-Anfragen in der PL) im Vordergrund. Die angesprochene Kluft zwischen PL und DBL wurde mit der Verfügbarkeit von OOPL noch größer. Heutzutage beobachten wir den Trend, auch relationale DBL mit objektorientierten Konzepten anzureichern (siehe Objekt-relationale Datenbanksysteme (ORDBVS) [22] und dazu auch den DB-Standard SQL:1999 [8, 9]), so daß sich die Kluft wieder verringert und damit die Frage der Anbindung in Form von Anwendungsprogrammierschnittstellen neu zu untersuchen ist. Dabei sind die im folgenden beschriebenen Anforderungen zu berücksichtigen, und es ist die Frage zu beantworten, ob die durch Integration von objektorientierten Konzepten (in relationale Datenmodelle) erreichte Verringerung der Distanz zwischen PL und DBL

2. Natürlich besteht diese Möglichkeit prinzipiell. Der ursprüngliche CLI-Ansatz verzichtet jedoch aus den genannten Gründen darauf.

zu einer einfacheren und damit insgesamt effizienteren Form der Anbindung führen kann.

Anforderungen

Wir haben uns bei dieser Untersuchung als Ziel gesetzt, die wesentlichen Vorteile bestehender Ansätze zu erhalten. So sollten die positiven Eigenschaften – also insbesondere Laufzeiteffizienz sowie strenge Typprüfung und frühe Fehlererkennung – eingebetteter Ansätze, wie eSQL oder SQLJ [19], genauso wie die hohe Datenunabhängigkeit der klassischen CLI-Ansätze, z. B. JDBC [3, 5], weiterhin gewährleistet werden. Erhöhter Aufwand zur Übersetzungszeit kann dabei in Kauf genommen werden, wenn die aus frühzeitigem Binden resultierenden Datenabhängigkeiten automatisch kontrolliert und bei Invalidierung an das aktuelle Schema angepaßt werden.

Die zu entwickelnden Konzepte sollten jedoch auf jeden Fall dem Anspruch der *Mehrsprachenfähigkeit* des DBVS genügen. Dies bedeutet, daß der Ansatz persistenter Programmiersprachen aufgrund der Beschränkung auf eine Programmiersprache nicht in Frage kommt. Vielmehr besteht das Ziel, einen neuen Ansatz zu entwickeln, der allgemeine Konzepte der Anbindung von OOPL an neuere, objektorientierte Konzepte unterstützende (OR)DBVS anbietet und damit größtmögliche *Unabhängigkeit* der beteiligten Sprachen gewährleistet.

Die (immer vorhandene) Effizienzanforderung ist insbesondere im Hinblick auf komplexe Anwendungsbereiche wie CAD oder Software-Entwicklung entscheidend. Solche Anwendungen weisen eine sehr hohe Referenzlokalität auf, so daß *Pufferung* von DB-Objekten in der Nähe der Anwendung (*client-seitiges Caching)* und damit eine geeignete Kontextverwaltung sowie ein effizientes Dereferenzieren (durch sog. *Pointer Swizzling*) zu entscheidenden Faktoren werden. Dies ist selbstverständlich auch bei der Bereitstellung einer (jeden) API zu beachten, da diese nicht nur Operatoren zur Verarbeitung von DB-Objekten in einer Programmiersprache verfügbar machen, sondern auch ein passendes *Verarbeitungsmodell* bereitstellen muß.

Überblick

Im nachfolgenden Kapitel werden wir zunächst die Rolle des Bindezeitpunktes für eine API herausarbeiten. Dabei werden wir die Vorteile des frühen Bindens skizzieren, die insbesondere bei Ausnutzung der Objektorientierung zum Tragen kommen. Weiterhin wird aufgezeigt, daß auch bei der Bereitstellung einer API Konfigurierung und Generierung als Mittel der Anpassung an die spezifischen Anforderungen einer gegebenen Anwendung herangezogen werden können und sollten. Kapitel 3 wird darauf aufbauend einen Überblick über die Mechanismen unseres Ansatzes der generierten Aufrufschnittstellen geben, bevor in den Kapiteln 4 bis 6 diese Prinzipien anhand eines konkreten Prototyp-Systems erläutert werden. Da die Effizienz unter den vielen Bewertungskriterien von API eines der wichtigsten und der am effektivsten meßbaren ist, vervollständigen wir unsere Erörterungen mit empirischen Leistungsbetrachtungen in Kapitel 7, bevor wir abschließend unsere wesentlichsten Ergebnisse zusammenfassen.

2 Die Rolle des Bindezeitpunkts

Der Einfluß von Bindezeitpunkten wird bereits bei der Bewertung bestehender Ansätze deutlich. Der reine CLI-Ansatz betrachtet eine DB-Anfrage im wesentlichen als einen String, der erst zur Laufzeit an das DBVS weitergereicht wird. Deshalb geschieht die Bindung der Anfrage an die Strukturen des DB-Schemas “spät (engl. late)”. Dies führt auf der einen Seite zu einem erhöhten Laufzeitaufwand, auf der anderen Seite jedoch auch zu einer geringen Empfindlichkeit gegen Schemaänderungen – also zu höherer Datenunabhängigkeit. Spracherweiterungen hingegen sind (eher) in der Lage, Anfragen schon zur Übersetzungszeit “vorzubereiten”, d. h. “früh (engl. early)” zu binden, wodurch im Vergleich zum CLI-Ansatz der Laufzeitaufwand verringert, aber die Datenabhängigkeit erhöht wird.

Um diese abstrakte Beschreibung des Aspekts der Bindung etwas zu konkretisieren, werden wir im folgenden näher erläutern, welche Aktionen beim Binden durchzuführen sind. Darüber hinaus diskutieren wir, welche Unterschiede sich ergeben, wenn man als Verarbeitungsmodell neben der server-zentrierten Verarbeitung auch eine client-seitige Verarbeitung (Anwendungspuffer) von DB-Daten unterstützen will und inwieweit durch die Ausnutzung objektorientierter Konzepte ein “flexibles Binden” erreicht werden kann.

2.1 Die klassische Anwendungsprogrammierung

Unter dem Begriff “klassische Anwendungsprogrammierung” fassen wir Mechanismen zusammen, die es einem Anwendungsprogramm ermöglichen, die server-seitige

(d. h. durch den DB-Server durchzuführende) Ausführung von deklarativen, mengenorientierten DB-Anfragen zu veranlassen.

Wir wollen zunächst kurz darlegen, was "Binden" bedeutet, ohne bereits auf den Bindungszeitpunkt einzugehen. Üblicherweise beinhaltet eine DB-Anfrage neben dem eigentlichen Operator eine Spezifikation der betroffenen bzw. relevanten Schemaelemente sowie Bedingungen zur Abgrenzung des Wirkungsbereiches der Operation auf der Ausprägungsebene. Schemaelemente können Attribute, Objekttypen (Relationen) oder auch Beziehungstypen sein. Sobald eine Anfrage an das DBVS übergeben wird, führt dieses vor der eigentlichen Ausführung der Anfrage (mindestens) die folgenden Schritte aus:

- Prüfung der in der Anfrage spezifizierten Schemaelemente gegen das DB-Schema;
- Prüfung der Zulässigkeit der Anfrage,
- Optimierung der Anfrage,
- Festlegung eines Ausführungsplans (Zugriffsmoduls) durch Auswahl der geeigneten Zugriffsoperatoren.

Im den ersten beiden Schritten werden Fragen geklärt wie: Existieren die angegebenen Objekttypen im DB-Schema? Gibt es Attribute mit den angegebenen Namen in dem entsprechenden Objekttyp? Ist der Benutzer befugt, diese Anfrage ausführen zu lassen? Die letzten beiden Schritte betreffen in der Regel aufwendige Optimierungsmaßnahmen mit einer Restrukturierung und Transformation der DB-Anfrage [7, 13]; sie wählen geeignete Zugriffspfade aus und legen somit die Reihenfolge der Anfrageausführung fest.

Sobald nun ein Ausführungsplan erstellt ist, kann das DBVS die Anfrage gemäß dieses Plans ausführen und die Ergebnisse der Anfrage an die Anwendung zurückliefern. Betrachtet man die Übersetzung des Anwendungsprogramms und die Übersetzung einer darin enthaltenen DB-Anfrage als orthogonale Aspekte, führt dies zu den prinzipiellen Möglichkeiten des frühen und späten Bindens. Läßt man die DB-Anfrage zum Zeitpunkt der Übersetzung des Anwendungsprogramms zunächst unbetrachtet und führt die oben angesprochen Schritte erst unmittelbar vor der Ausführung der Anfrage, d. h. zur Laufzeit des Anwendungsprogramms aus, so spricht man von spätem Binden. Wird hingegen zum Zeitpunkt der Übersetzung des Anwendungsprogramms das DBVS kontaktiert und zur Durchführung der angesprochenen Schritte, d. h. zur Erstellung eines Zugriffsmoduls für die Anfrage veranlaßt, so spricht man von frühem Binden. Es ermöglicht einen geringeren Laufzeitaufwand, bringt aber einen höheren Grad an Datenabhängigkeit mit sich, da möglicherweise bereits existierende Zugriffsmodule durch nachfolgende Schemaänderungen[3] invalidiert werden. Spätes Binden erhält größtmögliche Flexibilität, ist aber ineffizienter.

2.2 Programmierung von Client-Anwendungen

Bei der Programmierung von Client-Anwendungen unterscheiden sich (bei Betrachtung der logischen Abläufe) die Möglichkeiten des frühen und späten Bindens nicht von denen im vorangegangenen Abschnitt genannten. Es ist lediglich zu beachten, daß Retrieval-Operationen zu einem (oft für den Benutzer/Anwendungsprogrammierer transparenten) Laden von DB-Daten in einen anwendungsnahen (client-seitigen) Puffer führen und Manipulationsoperationen nicht direkt auf die vom DB-Server verwalteten Originaldaten, sondern auf im Anwendungspuffer liegenden Kopien dieser Daten wirken.

Es ist jedoch zu berücksichtigen, daß bezüglich des Effizienzaspektes frühe Bindungen weiter gegen späte Bindungen gewinnen. Dies liegt unter anderem daran, daß die von generischen (spät gebundenen) Operationen durchzuführenden Prüfungen entweder mittels Client/Server-Kommunikation abgewickelt oder durch einen client-seitigen Metadatenpuffer mit entsprechenden Zugriffsroutinen unterstützt werden müssen.

Offensichtlich ist eine erhebliche Leistungssteigerung zu erreichen, wenn solche Prüfungen bereits zur Übersetzungszeit durchgeführt werden. Weiterhin lassen sich Entscheidungen darüber, wie die Zugriffe auf die im Hauptspeicher (Anwendungspuffer) liegenden Anwendungsdaten am effizientesten erfolgen können, bereits früh treffen. Diese Entscheidungsfindung kann durch Wissen um die Anfrage und um die von der Anfrage betroffenen Schemaelemente unterstützt werden. Auch die im folgenden Abschnitt angesprochenen Konzepte der Objektorientierung sind hier sehr hilfreich einsetzbar.

3. Schemaänderungen können neben Objekttypen und Attributen auch Integritätsbedingungen, Indexstrukturen usw. betreffen.

2.3 Ausnutzung der Objektorientierung

Nachdem die zunächst vorrangig in Programmiersprachen realisierten Konzepte der Objektorientierung nun auch von Datenbanksystemen unterstützt werden (sollen) [22, 8, 9], unterscheiden sich die Datenmodelle der objektorientierten Programmiersprachen, wie Java oder C++, und der anzubindenden objekt-relationalen DB-Sprache, wie SQL3 [8, 9], nicht mehr so deutlich wie vorher. Die wichtigsten, nun auf beiden Seiten vorhandenen Konzepte sind Referenzen und abstrakte Datentypen. Referenzen erlauben die direkte Modellierung von (n:m)-Beziehungstypen. Abstrakte Datentypen (kurz ADTs) unterstützen die Trennung von Schnittstellen und Implementierungen. Ferner erlauben sie die anwendungsspezifische Erweiterung des Typsystems, ohne die zughörige Grammatik ändern zu müssen. Aus diesem Grund bietet es sich auch an, für die Anbindung (von Programmier- und DB-Sprache) benötigte Erweiterungen in Form von ADTs zu implementieren und damit Grammatikerweiterungen zu vermeiden.

Die Bindung von Schnittstellen und Implementierungen ist prinzipiell orthogonal. Die am häufigsten zu findenden Kombinationen sind generische Schnittstellen mit generischen Implementierungen (etwa ODBC) und früh gebundene Schnittstellen mit früh gebundenen Implementierungen (etwa eSQL)[4]. Doch auch die anderen beiden Kombinationen (siehe Abb. 2) finden ihre Berechtigung.

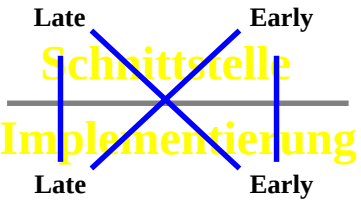

Abb. 2: Kombination von Bindungen

Folgende Daumenregeln charakterisieren die vier Kombinationen, die wir auch als *Bindungsgrade* bezeichnen:

- Generische Schnittstellen und Implementierungen (LATE/LATE oder einfach LATE) bieten folgende Vorteile. Der Aufwand sowohl für die Bereitstellung der Schnittstellenfunktionen selbst (interne Realisierung der API) als auch der Aufwand für die Übersetzung des Anwendungsprogramms sind gering. Darüber hinaus liegt größtmögliche Datenunabhängigkeit vor. Der DB-seitige Übersetzungs- und Bindungsaufwand fällt jedoch zur Laufzeit an und belastet damit die Abwicklung von DB-Anfragen.
- Generische Schnittstellen mit früh gebundenen Implementierungen (LATE/EARLY) erhöhen im Vergleich zur vorgenannten Kombination den internen Realisierungsaufwand und die Anwendungsübersetzungszeiten (da alle möglichen Fälle vorbereitet werden müssen), steigern aber die Effizienz der DB-Zugriffe. Offensichtlich läßt sich dieser Bindungsgrad nur heranziehen, wenn die Menge der möglichen Fälle zur Übersetzungszeit bekannt ist; dies ist beispielsweise bei Zugriffsoperationen auf DB-Objekten in einem Anwendungspuffer gegeben. Um jedoch eine ähnlich hohe Datenunabhängigkeit wie beim Bindungsgrad LATE(/LATE) zu erhalten, ist eine automatische Neugenerierung (mit wiederholter Übersetzung) der früh gebundenen Implementierungen nach Schemaänderungen sowie die Möglichkeit des Austauschs von Code-Fragmenten anzubieten.
- Früh gebundene Schnittstellen und Implementierungen (EARLY/EARLY oder kurz EARLY) erlauben die schnellsten DB-Zugriffe. Zusätzlich wird die Fehlerbehandlung verbessert, da sich bereits zur Übersetzungszeit Fehler erkennen und beheben lassen. Erhöhter Realisierungsaufwand und erhöhte Übersetzungszeiten fallen weniger ins Gewicht, da sie die Laufzeit der DB-Anfragen und damit die Laufzeit des Anwendungsprogramms und die Antwortzeit für den (End-)Benutzer nicht negativ beeinflussen. Probleme entstehen jedoch durch die im Vergleich zu den vorgenannten Bindungsgraden stark verminderte Datenunabhängigkeit.
- Früh gebundene Schnittstellen mit generischen Implementierungen (EARLY/LATE) dienen weniger der Effizienz der DB-Zugriffe. Sie erleichtern allenfalls den prototypischen Entwurf des API. Aufrufe sind typsicher, und die Implementierung kann jederzeit durch eine optimierte Variante ersetzt werden.

Die zweite Kombination LATE/EARLY birgt aus unserer Sicht ein hohes Akzeptanzpotential für die Zukunft, da die meisten DB-Anwendungen heutzutage mit ODBC programmiert werden und Effizienzprobleme der meistgenannte Kritikpunkt ist. Wir nennen diese zweite Kombination **virtuelles spätes Binden** (VIRTUAL_LATE), da die frühe Bindung der Implementierung dem Anwendungsprogrammierer verborgen bleibt.

4. Auch viele API für objektorientierte DBVS (OODBVS) realisieren diese Kombination.

2.4 Generierung und Konfigurierung

Um alle genannten Bindungsgrade nutzbar zu machen, schlagen wir konfigurierbare generierte Aufrufschnittstellen vor [18]. Über ein Konfigurationsprogramm lassen sich unterschiedliche Bindungsgrade für unterschiedliche DB-Anfragen einsetzen, wie in den nachfolgenden Kapiteln näher beschrieben wird.

Es hat sich gezeigt, daß die Wahl früher Bindungen die Effizienz, die Fehlerbehandlung und die Anwendungscodierung erheblich beeinflussen. Auf der anderen Seite ist mehr Aufwand für die Realisierung des API zu treiben und der Anwendungsübersetzungsaufwand steigt. Die Datenabhängigkeit wird ebenfalls erhöht. Daher ist es notwendig, frühe Bindungen möglichst nur dort einzusetzen, wo der Mehraufwand auf der einen Seite einen größeren Nutzen auf der anderen Seite bewirkt. Auch aus diesem Blickwinkel ist die Konfigurierbarkeit des API unabdingbar.

Entsprechend betrachten wir als wesentlichsten Aspekt der Generierung und Konfigurierung die *Wahl des Bindezeitpunktes.* Dabei können sowohl DB-Anfragen als auch Operationen zur Verarbeitung von DB-Daten im Anwendungspuffer spezifisch gebunden werden. Auf diese Weise wird der Anwendungsprogrammierer in die Lage versetzt, für jede einzelne DB-Anfrage zu entscheiden, ob Effizienz oder Datenunabhängigkeit (bzgl. der durch die Anfrage angesprochenen Schemaelemente) im Vordergrund stehen sollen. Wird Effizienz bevorzugt, so muß früh gebunden werden. Wir möchten an dieser Stelle noch einmal betonen, daß auf diese Weise für jede einzelne Anfrage entschieden werden kann, welcher Bindungsgrad (siehe Abschnitt 2.3) angemessen ist.

Da wir die client-seitige Verarbeitung optimieren wollen, muß eine hauptspeicherbasierte Adressierung der DB-Objekte möglich sein. Üblicherweise läßt sich die vorliegende externspeicherbasierte Adressierung über DB-Identifikatoren durch sog. *Pointer-Swizzling*-Mechanismen auf hauptspeicherbasierte Adressierung umsetzen. Daher sollte der Anwendungsprogrammierer bei der Generierung und Konfigurierung entscheiden können, welche DB-Identifikatoren zu welchen Zeitpunkten auf welche Weise in Hauptspeicherreferenzen umgesetzt werden sollen. Schließlich kann er am ehesten beurteilen, wie häufig bestimmte Anwendungsdaten während der Verarbeitung dereferenziert werden.

Als weiteren wesentlichen Aspekt der Generierung und Konfigurierung sehen wir das *Prefetching* von DB-Objekten. Auch hier sollte das Wissen des Anwendungsprogrammierers ausgenutzt werden können. Darauf aufbauend können zu bestimmten Ereignissen dynamisch a-priori beschriebene Verarbeitungskontexte geladen werden, um so nachfolgende Verarbeitungsschritte zu optimieren.

Generierte Aufrufschnittstellen sind ein sehr flexibles und leistungsfähiges Instrument zur DB-Anwendungsprogrammierung. Kernpunkte bei unserem Ansatz sind wahlfreie Bindungsgrade[5] für schema- und anfragespezifische Typinformationen sowie die Unterscheidung der Bindung von Schnittstellen und Implementierungen von abstrakten Datentypen (oder Klassen).

3 Generierte Aufrufschnittstellen

Generierte Aufrufschnittstellen sind Spezialisierungen von (regulären) Aufrufschnittstellen. Der erste Abschnitt dieses Kapitels faßt die wichtigsten ausgenutzten Vorbedingungen zusammen und gibt eine Begriffsdefinition (siehe auch [17, 18]). Der zweite Abschnitt beschreibt das Potential des Ansatzes anhand eines Überblicks über die Systemarchitektur.

3.1 Vorbedingungen und Definition

Mehrere Konzepte erweiterter DBL vereinfachen die Anbindung an OOPL:

- Erweiterte Typsysteme (Kollektionen, benutzerdefinierte Typen und Funktionen),
- Surrogatkonzepte als Grundlage für die Identifikation und Referenzierung von DB-Objekten,
- (Mengenwertige) Referenzen zwischen DB-Objekten zur direkten Umsetzung von (n:m)-Beziehungen.

Folgende Konzepte von OOPL helfen, Aufrufschnittstellen zu verbessern.

- Erweiterbare Typsysteme,
- Kapselung durch Schnittstellen für den Austausch äquivalenter implementierender Klassen,
- Polymorphie und der Subtypmechanismus als Kompromiß zwischen reinem frühen und reinem späten Binden (polymorphe Bindungen),

5. Durch wiederholte Codierung lassen sich für eine Operation sogar mehrere Bindungsgrade vorsehen.

Aufgrund der genannten OOPL-Eigenschaften lassen sich DB-Objekte wie PL-Objekte handhaben. Weiterhin können die Klassen, die DB-Objekte in der Programmiersprache repräsentieren, alle in Abschnitt 2.3 genannten Bindungsgrade verkörpern. Um frühe Bindungen für Typinformationen einzuführen, ist Codegenerierung notwendig.

Definition: *Eine **generierte** oder früh gebundene **Aufrufschnittstelle** (kurz **gCLI** für engl. generated call-level interface) ist eine Aufrufschnittstelle, die zur Übersetzungszeit von Schemata, Anfragen oder Anwendungen durch Codegenerierung erzeugt wird und somit frühe Bindungen ausnutzen kann.*

3.2 Überblick

Abb. 3 gibt einen Überblick über unseren gCLI-Ansatz. Auf der linken Seite der Abbildung sind die Komponenten zu sehen, die zur Laufzeit eines Anwendungsprogrammes zusammenwirken: das Anwendungsprogramm selbst, das generierte Laufzeitsystem (gLZS), ein Cache-Modul sowie das zugrundeliegende DBVS.

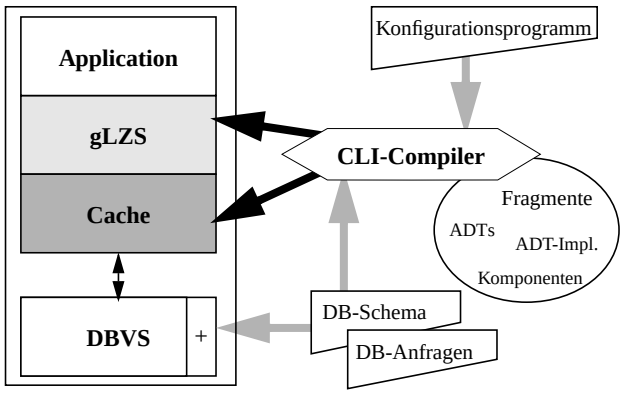

Abb. 3: Überblick über die Systemarchitektur

Ortstransparenz bzgl. der Zugriffe auf DB-Objekte wird client-seitig durch ein Cache-Modul erreicht. Dieses ist verantwortlich für die Kommunikation mit dem DBS, erlaubt (mengenorientierten) DB-Zugriff und bildet DB-Objekte auf eine zugehörige Repräsentation im Client-Cache ab.

Die eigentlichen API-Operationen werden vom generierten Laufzeitsystem (gLZS) angeboten. Dieses realisiert die operationale Semantik des API und bietet früh gebundene und optional spät gebundene Schnittstellen, die wiederum auf den Funktionen des Cache-Moduls aufsetzen. Das gLZS bietet dem Anwendungsprogrammierer somit allgemeine Schnittstellen für den DB-Zugriff (Suche, Einfügen, Ändern, Löschen, Transaktionskontrolle usw.). Darüber hinaus können spezielle Cursor bzw. Iteratoren zur Bearbeitung von DB-Objekten genutzt werden. Zusätzlich erlaubt ein DB-Katalog (auch für den Anwendungsprogrammierer zugängliche) Metadatenzugriffe.

Bevor ein gCLI nun zur Anwendungsprogrammierung bzw. zur Ausführung von Anwendungsprogrammen genutzt werden kann, muß zunächst der sog. CLI-Compiler aufgerufen werden, der die oben angesprochenen Aufgaben der Generierung und Konfigurierung wahrnimmt und dabei große Teile des gLZS und des Cache-Moduls mit generativen Mechanismen zur Verfügung stellt. Benutzerwissen bzw. anwendungsspezifische Parameter werden ihm dabei durch ein Konfigurationsprogramm mitgeteilt. Der CLI-Compiler erzeugt im wesentlichen frühe Bindungen von Typinformationen und erlaubt den Austausch (früh gebundener) äquivalenter Implementierungen (Klassen, die API-Funktionalität implementieren).

Ein tieferes Verständnis unseres gCLI-Ansatzes wollen wir dadurch schaffen, daß wir ein konkretes, vollständig implementiertes Prototyp-System erläutern. Es handelt sich dabei um einen Ansatz, der die Generierung von API-Funktionen zum Zugriff auf das Komplexobjekt-DBVS PRIMA erlaubt. Dieses nachfolgend eingeführte System setzt mit seinem Surrogatkonzept, seinen mengenwertigen Referenzen und seinem erweiterten Typsystem wesentliche Konzepte objekt-relationaler Theorie in die Praxis um.

4 PRIMA

PRIMA (Prototyp-Implementierung des Molekül-Atom-Datenmodells) wurde an der Universität Kaiserslautern entwickelt [6, 20, 4]. Es basiert auf dem Molekül-Atom-Datenmodell (MAD) und realisiert ein vollständiges DBVS. Es umfaßt u. a. geeignete Speicherungsstrukturen für die persistente Verwaltung komplexer Objekte, geschachtelte Transaktionen, die n-mengenorientierte[6] Anfragesprache MQL (*Molecule Query Language*) sowie dazugehörige Komponenten zur Übersetzung, Optimierung und Ausführung von Anfragen.

6. Mit “n-mengenorientiert” soll hervorgehoben werden, daß die Anfragesprache in der Lage ist, mehrere Ausgangsmengen in einem Schritt zu verarbeiten. Im Gegensatz dazu erlauben viele OODBVS, deren Anfragesprachen häufig nur 1-mengenorientiert sind, lediglich die Verarbeitung einer Ausgangsmenge (Kollektion) in einem Schritt.

4.1 Das Molekül-Atom-Datenmodell

Bei dem Molekül-Atom-Datenmodell (MAD) handelt es sich um ein Komplexobjekt-Datenmodell, das aufbauend auf dem Relationenmodell weiterführende Konzepte integriert [12, 20]. Die wesentlichen Konzepte von MAD werden nun im Rahmen eines Beispiels erklärt. Dazu greifen wir auf ein Teilschema des OO7-Benchmarks [2] (genannt MicroOO7) zurück. Abb. 4 zeigt das abzubildende Entity/Relationship-Diagramm.

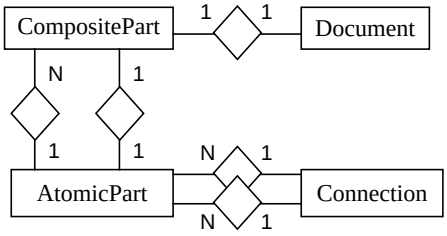

Abb. 4: Entity/Relationship-Diagramm zu MicroOO7

- *Atome* sind die Grundbausteine in MAD.

  Sie bilden die Entities des Entity/Relationship-Modells ab. Jedes Atom gehört zu einem eindeutigen Atomtyp (wie *AtomicPart* in Abb. 4), der die Attribute der zu modellierenden Entity-Menge beschreibt und dessen Extension alle Atome dieser Entity-Menge umfaßt. Die Datentypen der Attribute sind vielfältiger als im Relationenmodell. Es können neben einfachen Datentypen wie BYTE oder INTEGER auch ARRAYs oder Listen auftreten.

  Ein Surrogatkonzept dient der eindeutigen Identifikation aller Atome. Dazu muß jeder Atomtyp genau ein Attribut vom Typ IDENTIFIER enthalten. Der zugehörige Attributwert, ein DB-Identifikator, wird vom System gewartet und vor externen Manipulationen geschützt. Ein einmal vergebener DB-Identifikator bleibt für die Laufzeit des Systems innerhalb einer Datenbank eindeutig.

- *Referenztypen* dienen der Umsetzung von Beziehungen zwischen Atomtypen.

  Referenztypen sind in MAD grundsätzlich symmetrisch. So besitzt ein Referenzattribut für den Verweis vom Atomtyp *CompositePart* auf den Atomtyp *AtomicPart* ein entsprechendes Gegenreferenzattribut vom Atomtyp *AtomicPart* auf den Atomtyp *CompositePart*. Ausprägungen der Referenzattribute heißen *Referenzen*. Bei der Umsetzung wird das Surrogatkonzept wiederverwendet; Referenzen werden also durch DB-Identifikatoren repräsentiert. Demzufolge besitzt eine Referenz von einem Atom *cp* zu einem Atom *ap* über ein Referenzattribut *root* immer auch eine Gegenreferenz von *ap* zu *cp* über das Gegenreferenzattribut in *ap* (siehe Abb. 5 und Beispiel 1).

  Zusätzlich zu den Primär-Fremdschlüssel-Beziehungen im Relationenmodell können Beziehungen damit wie in OODBVS oder ORDBVS durch die systemvergebenen DB-Identifikatoren ausgedrückt werden. Durch den orthogonalen Einsatz von Listen über Referenzen können auch (n:m)-Beziehungen direkt umgesetzt und bei der späteren Verarbeitung zur Optimierung eingesetzt werden[7].

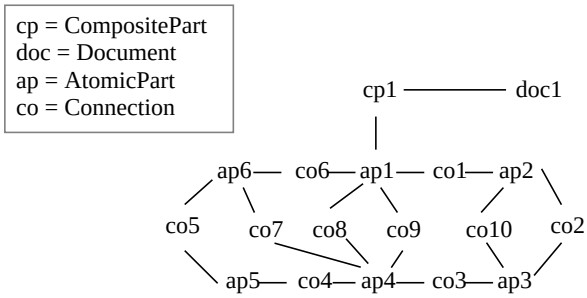

Abb. 5: Netzwerk von Atomen nach MicroOO7

Das DB-Schema besteht aus einem ungerichteten Netzwerk von Atomtypen aufgespannt von den Referenztypen zwischen den Atomtypen. Die Datenbank umfaßt darüber hinaus als Ausprägungen dieses Schemas Atome und Referenzen und bildet ebenfalls ein ungerichtetes Netzwerk wie in Abb. 5.

4.2 Die DB-Sprache MQL

Ein wesentliches Konzept in heutigen DBVS sind deskriptive und mengenorientierte DBL. Relationale DBL wie SQL und deren Erweiterungen für ORDBVS stellen in der Regel Teilsprachen zur Beschreibung von DB-Schemata, DB-Anfragen und Optimierungen zur Verfügung. Für das PRIMA-System wurde ebenfalls eine eigene DBL, die sogenannte MQL (abgekürzt von engl. *Molecule Query Language*), eingeführt. MQL enthält die folgende Teilsprachen.

7. Leider unterstützen die führenden ORDBVS auf dem kommerziellen Markt mengenwertige Referenzen bislang nur unvollständig, so daß man an dieser Stelle umständlicher modellieren muß.

- Die DDL (abgekürzt von engl. *Data Definition Language*) erlaubt die DB-Schemadefinition. Sie umfaßt Sprachmittel zur Definition und Manipulation von Atomtypen.
- Über die LDL (abgekürzt von engl. *Load Definition Language*) können Speicherungsstrukturen und Zugriffspfade für die in der DDL beschriebenen Datenstrukturen spezifiziert werden. Atomtypen werden in PRIMA, falls keine zusätzlichen Angaben gemacht wurden, durch eine standardisierte Abbildung auf eine Speicherungsstruktur abgebildet, in der alle Atome eines Atomtyps zusammenhängend abgespeichert sind.
- Die bisher vorgestellten Teilsprachen bieten ausschließlich Möglichkeiten zur Definition der logischen Datenstrukturen sowie zu deren physischer Repräsentation im System. Die DML (abgekürzt von engl. *Data Manipulation Language*) bietet dem Anwender Möglichkeiten zur Manipulation und zum Lesen der Datenbestände. Ähnlich zur DML in SQL werden Anweisungen auch in dieser Teilsprache in den folgenden drei Teilen definiert:

  Operation *Projektionsklausel*
  FROM *Molekültypspezifikation*
  WHERE *Qualifikationsklausel;*

  Dabei steht *Operation* für die eigentliche Anweisung (INSERT, DELETE, UPDATE, SELECT). In der *Molekültypspezifikation* der FROM-Klausel wird der *Molekültyp* festgelegt. Innerhalb der Molekültypdefinitionen werden alle Pfade vom eindeutigen Wurzelatomtyp zu allen Komponentenatomtypen vollständig spezifiziert. Für jedes Molekül, das diesem Typ entspricht, wird die in der WHERE-Klausel angegebene Bedingung überprüft. Alle Moleküle, die diese Bedingung erfüllen, werden der angegebenen Operation unterzogen. Dabei gibt die Projektionsklausel im Fall der SELECT-Operation an, welche Molekülteile der Ergebnismenge erscheinen.

Nachfolgend sollen DDL und DML anhand des oben eingeführten MicroOO7-Beispiels verdeutlicht werden. Beispiel 1 zeigt das MicroOO7-Schema. Es ist bis auf die Referenzattribute selbsterklärend. Referenzattribute sind vom Typ *REF_TO*. In Klammern dahinter wird das Gegenreferenzattribut angegeben. Kardinalitätsrestriktionen können herangezogen werden, um den Abbildungstyp eines Beziehungstyps näher zu spezifizieren.

Beispiel 1 MicroOO7-Schema in MQL/DDL:

```
CREATE ATOM_TYPE CompositePart(
      ID: IDENTIFIER,
      type: LIST_OF(BYTE),
      buildDate: INTEGER,
      document: REF_TO
         (Document.relPart)(0,1),
      parts: REF_TO
         (AtomicPart.partOf),
      root: REF_TO
         (AtomicPart.rootOf)(1,1)
);
CREATE ATOM_TYPE AtomicPart(
      ID: IDENTIFIER,
      myId,docId: INTEGER,
      type: LIST_OF(BYTE),
      buildDate,x,y: INTEGER,
      to: REF_TO (Connection.from),
      from: REF_TO (Connection.to),
      partOf: REF_TO
         (CompositePart.parts)(1,1),
      rootOf: REF_TO
         (CompositePart.root)(1,1)
);
CREATE ATOM_TYPE Connection(
      ID: IDENTIFIER,
      type: LIST_OF(BYTE),
      docId: INTEGER,
      from: REF_TO (AtomicPart.to),
      to: REF_TO (AtomicPart.from)
);
CREATE ATOM_TYPE Document(
      ID: IDENTIFIER,
      myid: INTEGER,
      title,text: LIST_OF(BYTE),
      relPart: REF_TO
         (CompositePart.document)(0,1)
);
```

Im Rahmen unseres Verarbeitungsmodells werden SELECT-Anfragen für die Spezifikation von Verarbeitungskontexten herangezogen. Deshalb soll nun als Beispiel eine solche SELECT-Anfrage genauer betrachtet werden. Die Anfrage in Abb. 6 baut einen einfachen Pfad (als Molekültyp der FROM-Klausel) auf[8].

In der Projektionsklausel wird hier keine Einschränkung zu der Sichtbarkeit von Teilergebnissen vorgenommen. Die FROM-Klausel startet mit dem Wurzelatomtyp *CompositePart*. Der Bindestrich "-" repräsentiert die Kanten im Molekültyp[9]. Die WHERE-Klausel schließlich formuliert eine Einschränkung für die auszuwählenden *CompositeParts*.

8. Auch hierarchische und netzwerkartige Molekültypen sind möglich.

Wir nehmen an, daß das Anfrageergebnis lediglich das eine in Abb. 6 dargestellte Molekül (mit Wurzelatom *cp1*) enthält, das offensichtlich die in der FROM-Klausel der Anfrage spezifizierte Struktur aufweist.

```
SELECT ALL
FROM CompositePart - AtomicPart - Connection
WHERE CompositePart.type = 'type002';
```

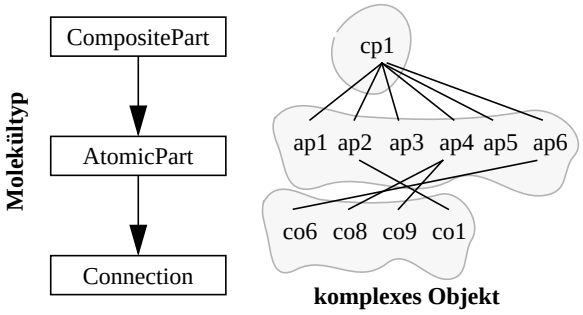

Abb. 6: DB-Anfrage und zugehöriges Ergebnismolekül

5 Die Konfigurationssprache

Der CLI-Compiler basiert auf einer Konfigurationssprache, wodurch die API-Generierung von der eigentlichen Anwendungsentwicklung entkoppelt wird. Über ein Konfigurationsprogramm können unterschiedliche Bindungsgrade eingestellt und Anwendungen optimiert werden.

Unser Ansatz ist typbasiert, um den Mehraufwand für die Einführung von frühen Bindungen insbesondere für "kleine" Operationen wie den Attributzugriff klein zu halten. Alle Ausprägungen eines Typs gehorchen also derselben Einstellung. In einem wertebasierten Ansatz dagegen kann die zu verwendende Strategie wertabhängig, d. h. für einzelne Objekte eingesetzt werden. Dies erfordert jedoch letztendlich zusätzliche Laufzeitprüfungen, was der frühen Bindung zuwiderläuft, oder die Migration der Strategie zur Laufzeit.

Unsere Erfahrungen haben gezeigt, daß es sinnvoll ist, die Konfigurationssprache auf Konzepten der DB-Sprache und auf der Schnittstelle des API aufzubauen. Dadurch ist die Konfiguration implementierungsunabhängig und portabel.

9. Besteht genau ein Beziehungstyp zwischen den jeweils angegebenen Atomtypen, so kann in der Molekültypspezifikation auf die Angabe der zugehörigen Attributnamen verzichtet werden.

Zur Notation werden in Anlehnung an die Definition der Grammatik von MQL folgende Metasymbole verwendet:

#	Abschluß einer Definition
[]	Option
{}	Auswahl
[]+	Wiederholung, mindestens einmal
[]*	Wiederholung, optional

Terminalsymbole sind entweder groß geschrieben oder unterstrichen. Ein Konfigurationsprogramm (*cl_program*) besteht aus einer Folge von Konfigurationsblöcken. Dabei kann es sich jeweils um einen Anfragekonfigurationsblock oder um einen atomtypspezifischen Konfigurationsblock handeln (*query_cntrl_block* bzw. *at_cntrl_block*). Wir konzentrieren uns hier auf Anfragekonfigurationsblöcke.

```
cl_program ::=
    BEGIN
        [query_cntrl_block | at_cntrl_block]+
    END #

query_cntrl_block ::=
    DEFINE_QUERY name AS mol_query
        [BINDING binding ;]
        [SWIZZLING swizzling ;]
        [query_rule ;]*
    END_QUERY #
```

Ein Anfragekonfigurationsblock führt in jedem Fall zur Spezialisierung der generischen Anfrageklasse. Dafür wird für die Anfrage eine eigene Klasse erzeugt. Die Anfrage erhält einen Namen (*name*), mit dem sie in PRIMA abgelegt werden kann. Die AS-Klausel nimmt die zugehörige MQL-Spezifikation der Anfrage auf (*mol_query*).

In der BINDING-Klausel bestimmt man den Bindungsgrad für den Molekül-Cursor zur Verarbeitung der Ergebnismolekülmenge.

```
binding ::= LATE | VIRTUAL_LATE | EARLY #
```

- LATE legt fest, daß für alle Rollen[10] der Zugriff über eine spät gebundene Schnittstelle erfolgt. Dabei wird in der Implementierung der Zugriff über eine Zuordnungstabelle für Rollennamen und Rollen-Cursor verwendet. Die Abfrage der Metadaten zum Molekültyp geschieht jeweils nur bei der Initialisierung des Molekül-Cursors.

10. In unserem Beispiel korrespondieren Rollenamen mit Atomtypnamen. Im allgemeinen kann ein Atomtypname mehrfach in einer Molekültypspezifikation auftreten, wobei jedoch für jedes Vorkommen ein eindeutiger Rollenname zu vergeben ist.

- EARLY legt fest, daß für jede Rolle eigene Zugriffsmethoden erzeugt werden sollen. Zum Übersetzungszeitpunkt wird die Rolleninformation in die Implementierung der zugehörigen Methoden geschrieben. Außerdem werden im Molekül-Cursor von vorneherein Instanzenvariablen angelegt, die den Zugriff beschleunigen, da sie den Weg über die Zuordnungstabelle einsparen.
- VIRTUAL_LATE stellt zunächst die gleichen spät gebundenen Schnittstellen wie LATE zur Verfügung. Im Gegensatz zu LATE erfolgen aber frühe Bindungen für die Implementierungen der Schnittstellen. So werden etwa zur Übersetzungszeit die Namen der zugelassenen Rollen in die Implementierungen geschrieben. Damit handelt es sich also um virtuell spät gebundene Schnittstellen.

In der SWIZZLING-Klausel kann zwischen drei Grundeinstellungen für das *Pointer Swizzling* gewählt werden.

```
swizzling ::= NONE | LAZY | EAGER #
```

- NONE legt fest, daß kein *Pointer Swizzling* durchgeführt werden soll. Daraus folgt, daß bei Einlagern von Atomen keine Referenztransformation erfolgt. Für die nachfolgenden Zugriffe muß zur Auflösung von Referenzen eine Umsetzungstabelle konsultiert werden.
- LAZY legt fest, daß Referenzen bei ihrer ersten Dereferenzierung transformiert werden.
- EAGER legt fest, daß alle Referenzen beim Laden der sie umgebenden Instanzen (hier Moleküle) transformiert werden.

Schließlich lassen sich einfache Regeln (*query_rule*) spezifizieren, mit denen *Pointer Swizzling, aber auch Prefetching* präziser eingestellt werden können:

```
query_rule ::=  ON query_event
                DO query_action #

query_event ::= exec_event | mol_event #

exec_event ::= execute | restore #

mol_event ::=   firstMol | nextMol |
                previousMol |
                openMoleculeCursor #

query_action ::= qswizzle_action #

qswizzle_action ::=  SWIZZLE query_granule #

query_granule ::= {CURRENT | ALL} #
```

Die ON-Klausel bestimmt das Ereignis und die DO-Klausel die auszuführende Aktion. Beispielsweise kann ein Ereignis die Ausführung einer API-Operation und die zugehörige Aktion etwa eine Zeigertransformation sein (*swizzle_action*). Dabei läßt sich bestimmen, welche Referenzen umgesetzt werden sollen. Hier kann man zwischen dem aktuellen Molekül und der gesamten Ergebnismolekülmenge wählen.[11]

Anmerkungen zum *Pointer Swizzling* in PRIMA/API

In der Literatur werden eine Reihe von Techniken zum *Pointer Swizzling* unterschieden [14, 23, 10, 15,]. Dabei kristallisieren sich im wesentlichen drei Dimensionen heraus.

- Die erste Dimension, *copy/in-place*, unterscheidet, ob die Referenztransformation in Originaldaten aus der Datenbank (*in-place*) oder in Objektkopien (*copy*) geschieht. Diese Kopien können etwa durch Extraktion von Objekten aus DB-Seiten erzeugt werden. Werden diese Kopien ausschließlich für das *Pointer Swizzling* erzeugt, dann ist der Speicheraufwand des *Copy Swizzling* viel höher als der des *In-Place Swizzling*. Oft werden jedoch aufgrund der Umsetzung des Verarbeitungsmodells auf dem Client ohnehin Kopien der DB-Daten erzeugt, so daß sich ein *Copy Swizzling* anbietet.
- Die zweite Dimension, *direct/indirect*, unterscheidet, ob bei der Referenztransformation Deskriptoren eingesetzt werden (*indirect*) oder nicht (*direct*). Die aus der Transformation resultierende Referenz zeigt dann zunächst auf einen Deskriptor, der wiederum auf die zu referenzierende Instanz verweist. Der Vorteil von *Direct Swizzling* ist die eingesparte Indirektion bei der Dereferenzierung. *Indirect Swizzling* wird dagegen häufig eingesetzt, wenn Instanzen aufgrund einer flexiblen Speicherverwaltung oder zur Verdrängung verschoben werden müssen. Verwendet man hier *Direct Swizzling*, so sind bei der Aktualisierung im schlimmsten Fall alle Instanzen im Verarbeitungskontext daraufhin zu überprüfen, ob sie Referenzen auf das verschobene Objekt enthalten. Häufig greift man hier auch auf Rückwärtsreferenzlisten (kurz RRL für engl. *re-*

11. Aus Übersichtlichkeitsgründen führen wir an dieser Stelle vereinfachte Definitionen an. Der eigentliche Ansatz erlaubt weitergehende Spezifikationen, deren Diskussion hier zu weit führen würde. Es sei lediglich angemerkt, daß unterschiedliche Aktionen hinsichtlich der Umsetzung der die Moleküle aufspannendenen Referenzen und weiterer Referenzen spezifiziert werden können. Weiterhin können auch kleinere Granulate, wie einzelne Atome als Granulate angegeben werden, was jedoch im allgemeinen zu einem unverhältnismäßig hohen, internen Mehraufwand führt und daher nur in speziellen Fällen sinnvoll nutzbar ist.

verse reference list) zurück, die sich die betreffenden Partner merken.

- Die dritte Dimension, *eager/lazy*, unterscheidet, ob DB-Referenzen im voraus (*eager*) oder aber erst beim eigentlichen Zugriff (*lazy*) transformiert werden. Beim *Eager Swizzling* werden also alle betroffenen DB-Referenzen in einem Schritt (meist bei der Einlagerung der zugehörigen DB-Daten in den Hauptspeicher) transformiert. Das *Lazy Swizzling* transformiert nur die DB-Referenzen, die während der Verarbeitung auch tatsächlich dereferenziert werden. Damit ist hier die Anzahl der transformierten Referenzen minimal[12]. Allerdings muß beim nachfolgenden Zugriff immer nachgefragt werden, ob eine Referenz transformiert ist oder nicht. Diese Abfrage nennt man das *Lazy-If*. Beim *Eager Swizzling* ist das *Lazy-If* nicht notwendig und kann eingespart werden.[13] Aus diesem Grund ist *Eager Swizzling* effizienter als *Lazy Swizzling*, zumindest wenn der überwiegende Teil der Referenzen auch tatsächlich verfolgt wird. Ist dies nicht der Fall, dann sind die Kosten des Transformationsvorgangs größer als die erzielten Einsparungen.

In OODBVS wird der Vorgang der Referenztransformation häufig an das Nachladen von Objekten geknüpft. Dies ist insbesondere dann sinnvoll, wenn es sich bei dem OODBVS um einen *Object Server* handelt. Durch die zusammengefaßte Durchführung des Einlagerns eines Objekts und der Transformation aller seiner Referenzen läßt sich zum einen durch die gemeinsame Verwendung der Objektinformation der Aufwand beider Aktionen reduzieren und zum anderen das *Lazy-If* beim späteren Zugriff einsparen. Es handelt sich also um eine Form des *Eager Swizzling*. Zur besseren Unterscheidung definieren wir sog. Swizzling-Granulate (etwa *Query*, *Object* und *Reference*). Diese geben das Granulat an, auf die sich eine Spezifikation der Referenztransformation bezieht. Im gerade diskutierten Fall handelt es sich also um *Object Swizzling*. Auch ein *Lazy Object Swizzling*, d. h. das Durchführen der Referenztransformation zum Zeitpunkt des ersten Zugriffs auf eine der Referenzen, die von einem Objekt ausgehen, ist denkbar. Das *Lazy Object Swizzling* deutet also das Verfolgen einer Referenz eines Objekts als Hinweis, alle Referenzen im Objekt zu transformieren. Dadurch wird jedoch das *Lazy-If* für nachfolgende Zugriffe erzwungen.

Für die Transformation der PRIMA-Referenzen gelten nun folgende Eigenschaften. Da, wie oben gezeigt, *Eager Swizzling* und *Lazy Swizzling* unterschiedlich effizient sein können, ist es sinnvoll, eine Auswahlmöglichkeit anzubieten. Ihr Einsatz sollte auch (dynamisch) ausgeschaltet werden können, etwa dann, wenn bekannt ist, daß keine wiederholten Dereferenzierungen auftreten. Wir stellen diese drei Möglichkeiten in unserem Prototyp zur Verfügung. Als Vorgabe ist *Eager Swizzling* festgelegt, d. h., bei der Einlagerung in den Anwendungspuffer werden alle relevanten DB-Referenzen in den Molekülen in Hauptspeicherreferenzen transformiert. Darüber hinaus gibt es Fälle, in denen der Einsatz von *Lazy Swizzling* sehr sinnvoll sein kann. Ein solcher Fall liegt etwa vor, wenn bekannt ist, daß die Anwendung ihren Verarbeitungskontext abschnittsweise verarbeitet. Dies ist beispielsweise gegeben, wenn zwischen dem Weiterschalten von einem Molekül auf das nächste extensive intramolekulare Verarbeitung stattfindet. In diesem Fall kann man das *Lazy-If* an das Öffnen bzw. Fortschalten des Molekül-Cursors binden. Alle nachfolgenden intramolekularen Zugriffe können davon ausgehen, daß die Referenzen, die sie verfolgen, bereits transformiert sind. Wir nennen diese Art der Referenztransformation *Molecule Pointer Swizzling*.

6 Anwendungsprogrammierung

Ein kleines Beispiel soll die Verarbeitung von Molekülen, die mit der vorgestellten Anfrage aus der Datenbank ausgewählt und in einem Anwendungspuffer auf der Client-Seite abgelegt werden, verdeutlichen. Aus Platzgründen wird hier nur ein Programmausschnitt gezeigt, der die Traversierung eines Moleküls repräsentiert. Nacheinander werden dazu die verschiedenen Bindungsgrade LATE, VIRTUAL_LATE und EARLY sowie entsprechende Möglichkeiten der Beeinflussung des Pointer Swizzling veranschaulicht.

Zur Verarbeitung von Anfrageresultaten bieten wir geschachtelte Cursor an. Solche sind aus flachen Cursorn aufgebaut, die entsprechend dem Molekültyp voneinander ab-

12. Berücksichtigt man, daß sich *Pointer Swizzling* erst lohnt, wenn transformierte Referenzen mindestens zweimal verfolgt werden, ist das *Lazy Swizzling* nicht minimal. Minimal wäre eine Technik, die genau diese Referenzen transformiert.

13. Die Einsparung des *Lazy-If* ist jedoch nur gewinnbringend, wenn sein Anteil an der Dereferenzierung genügend groß ist und die Dereferenzierungen zusammengenommen den Löwenanteil an der von einer Anwendung verbrauchten CPU-Zeit besitzen.

hängen. Der flache Cursor für *AtomicPart* verläßt unter normalen Umständen nicht die Menge der Söhne, die durch den aktuellen Vater, den flachen Cursor für *CompositePart*, vorgegeben ist. Und der flache Cursor für *Connection* wird nur solche *Connections* adressieren, die zu dem *AtomicPart* gehören, daß vom aktuellen flachen Cursor auf *AtomicPart* identifiziert ist. Wenn also *ap2* das aktuelle *AtomicPart* in Abb. 6 ist, dann kann nur *co1* mit dem Cursor für *Connection* bearbeitet werden. Alle flachen Cursor stehen wiederum unter der Kontrolle des zugehörigen geschachtelten Cursors, um konsistente Zustände und Operationen zu garantieren.

6.1 Der Bindungsgrad LATE

Um die obligatorisch verfügbare, spät gebundene Schnittstelle zu verwenden, ist folgende Anweisung notwendig:

```
import MoleculeAPI.*;
```

In diesem Paket (Java package) sind die Schnittstellen und Implementierungen des API gegeben. Die wichtigsten sind `Query` und `MoleculeCursor` für die inter- bzw. intramolekulare Verarbeitung.

Als erstes muß eine Instanz der Anfrage erzeugt werden:

```
Query myQuery = new Query("
    SELECT ALL
    FROM CompositePart - AtomicPart -
        Connection
    WHERE CompositePart.type = 'type002';
");
```

Der Aufruf der Methode *compile* liefert den Anfragetext an den MQL-Compiler:

```
myQuery.compile();
```

Der Name der Anfrage oder der Name der Anwendung wird für die Entkopplung der Übersetzung von der Ausführung der Anfrage verwendet.

Der Aufruf der Methode *execute* fordert PRIMA auf, die Anfrage auszuwerten. Dabei wird eine Transaktion geöffnet:

```
myQuery.execute();
```

Intern dient der Name der Anfrage als Parameter für die Ausführungsoperation. Die Ergebnismenge wird zum Client transportiert und dort im Anwendungspuffer eingelagert. Der Ergebnismengen-Cursor wird dabei geöffnet.

Folgende Anweisung setzt den Cursor auf das erste Molekül der Ergebnismenge:

```
myQuery.firstMolecule();
```

Zur intramolekularen Verarbeitung muß zunächst ein Molekül-Cursor geöffnet werden. Danach werden das erste *CompositePart* und darunter das erste *AtomicPart* identifiziert. Schließlich erfolgt der Attributzugriff:

```
MoleculeCursor mol = myQuery.openMoleculeCursor();
mol.first("CompositePart").first("AtomicPart");
mol.getAttribute("AtomicPart", "x", someInt);
```

6.2 Der Bindungsgrad VIRTUAL_LATE

Die virtuell spät gebundene Schnittstelle wird nur auf Anforderung erzeugt. Dazu ist ein Konfigurationsprogramm notwendig. Es definiert zunächst *Lazy Pointer Swizzling* als Grundeinstellung (SWIZZLING LAZY). Eine Anfrage namens *demo* wird definiert (DEFINE_QUERY), der Bindungsgrad für sie eingestellt (BINDING VIRTUAL_LATE) und eine Optimierungsregel zum *Pointer Swizzling* spezifiziert (ON ...):

```
BEGIN
SWIZZLING LAZY;
DEFINE_QUERY demo AS
        SELECT ALL
        FROM CompositePart - AtomicPart -
            Connection
        WHERE CompositePart.type = 'type002';
        BINDING VIRTUAL_LATE;
        ON execute DO SWIZZLE ALL;
END_QUERY;
END
```

Dieses Konfigurationsprogramm dient als Eingabe für den CLI-Compiler. Dieser extrahiert den Anfragetext, läßt die Anfrage vom MQL-Compiler übersetzen, speichert resultierende Metadaten und generiert zusätzlichen Code für das API.

Die Aufrufe der API-Operationen im Anwendungsprogramm sind dieselben wie bei Nutzung der spät gebundenen Schnittstelle (siehe vorherigen Abschnitt). Die Anfragedefinition kann allerdings auch ausgelassen werden, da sie schon durch das Konfigurationsprogramm bekannt ist.

6.3 Der Bindungsgrad EARLY

Auch dieser Bindungsgrad erfordert ein Konfigurationsprogramm:

```
BEGIN
SWIZZLING LAZY;
DEFINE_QUERY demo AS
        SELECT ALL
        FROM CompositePart - AtomicPart -
            Connection
        WHERE CompositePart.type = 'type002';
        BINDING EARLY;
        ON execute DO SWIZZLE ALL;
END_QUERY;
END
```

Im Unterschied zu dem im vorangegangenen Abschnitt angegebenen Konfigurationsprogramm wurde nur die BINDING-Anweisung geändert. Die relevanten Anweisungen im Anwendungsprogramm sehen nun aber wie folgt aus.

Zunächst müssen wiederum die Schnittstellen und Klassen importiert werden. Das bereitgestellte Paket heißt *Generated*:

```
import MoleculeAPI.Generated.*;
```

Anschließend wird der Anfragerepräsentant erzeugt. Der Name der Anfrage findet sich als Teil des Klassennamens wieder:

```
Query_demo myQuery = new Query_demo();
```

Übersetzung und Ausführung der Anfrage sind durch Aufruf der zu diesen Zwecken bereitgestellten Operationen abzuwickeln:

```
myQuery.compile();
myQuery.execute();
```

Folgende Anweisung setzt den Ergebnis-Cursor auf das erste Element des Anfrageresultats.

```
myQuery.firstMolecule();
```

Die intramolekulare Verarbeitung sieht dann folgendermaßen aus:

```
MoleculeCursor_demo mol =
myQuery.openMoleculeCursor();
mol.first_CompositePart().first_AtomicPart();
mol.AtomicPart().set_x(someInt);
```

Offensichtlich sind alle Bezeichner, die in den vorangegangenen Abschnitten (als Atomtyp- oder Attributnamen) eingeführt wurden, in die Namen der Klassen und Methoden eingeflossen. Dies verdeutlicht, daß nur für tatsächlich im DB-Schema vorhandene und in der betrachteten Anfrage auftretende Schemaelemente spezifische Operatoren zur Verarbeitung der Ergebnismenge zur Verfügung stehen. Diese wiederum sind aufgrund ihrer frühen Bindungen laufzeitoptimiert.

7 Experimente

Nachdem wir unseren Ansatz durch eine DB-Sprache (am Beispiel von MQL) und eine Konfigurationssprache konkretisiert hatten, haben wir im letzten Kapitel die Spezifikation der verschiedenen Bindungsgrade und ihre Nutzung in einer Anwendung skizziert. Um Aussagen über seine Effizienz zu gewinnen, sind empirische Untersuchungen erforderlich. Allerdings läßt sich eine Leistungsbewertung unseres Ansatzes nicht „flächendeckend“ durchführen, da diese anwendungsspezifisch zu erfolgen hat, wobei die Art der DB-Operationen und ihre Aufrufhäufigkeit eine dominierende Rolle spielen. Darüber hinaus erlaubt unser Generierungsansatz die Variation eines Spektrums von Parametern (mit reichhaltigen anwendungs- und umgebungsspezifischen Optionen zum Caching, Swizzling und Prefetching), so daß sich schwerlich typische Einsatzfälle isolieren lassen. Aus diesen Gründen wollen wir uns vielmehr auf die Analyse von Eckpunkten und charakteristischen DB-Operationen beschränken, um „Daumenwerte“ (Indikatoren) für den Leistungsgewinn bei früher Bindung (EARLY vs. LATE) zu erhalten.

Daher beschreiben wir nur für einige charakteristische Zugriffsoperationen, wie sie vor allem bei der client-seitigen DB-Verarbeitung auftreten, eine Reihe von Messungen, deren Ergebnisse den durch früh gebundene Schnittstellen erreichbaren Effizienzgewinn aufzeigen und damit den Nutzen unseres gCLI-Ansatzes verdeutlichen. Bei den experimentellen Messungen wurden (wie schon bei der Implementierung des Prototypen) das JDK 1.1.5 (kurz für engl. *Java Development Kit*) und die Klassenbibliothek JGL (kurz für engl. *Java Generic Library*) unter Solaris 2.5 und Windows95 eingesetzt.

Um die Meßergebnisse besser beurteilen zu können, sollen die Einflüsse aller Abstraktionsebenen unserer Implementierung untersucht werden, wie sie in Abb. 7 dargestellt sind. Die unterste Ebene wird gebildet durch die von der

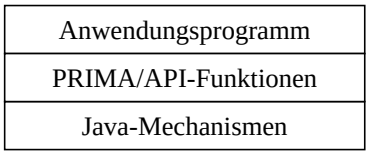

Abb. 7: Abstraktionsebenen für Messungen

Implementierungssprache Java angebotenen Mechanismen, die in unserer Realisierung genutzt werden. Hierbei sehen wir die verschiedenen Möglichkeiten des Attributzugriffs, teilweise abhängig von Speicherverwaltung und Bindungsgrad, als typisch an. Die mittlere Ebene ist gegeben durch die (generierten) API-Funktionen, deren Effizienz wesentlich von der Leistung der angesprochenen Java-Mechanismen mitbestimmt wird. Charakteristisch in unserem Sinne sind hier die Navigation über Molekül-Cursor (Verfolgen von Referenzen) und der Attributzugriff bei Atomen. Die oberste Ebene ist verkörpert durch ein, wiederum in Java geschriebenes, Anwendungsprogramm, das die vom gCLI angebotenen Schnittstellen zur Manipulation von DB-Objekten im Anwendungspuffer nutzt. Bei unseren Experimenten ziehen wir eine navigierende Anwendung auf einer Molekülmenge heran, welche die auf der zweiten Ebene bereitgestellten API-Funktionen für DB-Zugriffe einsetzt.

Da wir hauptsächlich an den Größenordnungen und Relationen der Zugriffszeiten interessiert sind, erfolgt die jeweilige Darstellung der Meßergebnisse normiert bezüglich des jeweils besten Wertes.

Java

Auf der untersten der zu betrachtenden Ebenen, gegeben durch die Implementierungssprache Java, sollen Zugriffe untersucht werden, die in PRIMA/API häufig eingesetzt werden. Dazu gehören der Zugriff auf Variablen, der indizierte Zugriff auf Einträge in Feldstrukturen (Array, Vector, JGL-Array)[14] und die schlüsselbasierte Suche in Hash-Tabellen.

Bei der Durchführung der Experimente wurden die einzelnen Operationen in einer Schleife jeweils 10.000 mal wiederholt, um einen stabilen Mittelwert zu erhalten. Tabelle 1 zeigt die relativen Werte (Verlustfaktoren) dazu; die Zeilen sind unabhängig voneinander normiert. So bedeuten die Einträge der ersten Zeile, daß der Vector-Zugriff um den Faktor 17, der JGL-Array-Zugriff um den Faktor 25 und der Hash-Map-Zugriff um den Faktor 62 langsamer ist als der Array-Zugriff, falls der Overhead der Schleifenbearbeitung vernachlässigt wird.

Tabelle 1: Verlustfaktoren beim Zugriff in Java

	Array	Vector	JGL-Array	Hash Map
ohne Overhead	1	17	25	62
mit Overhead	1	13	14	53
mit Overhead, JIT	1	6	7	13

Damit ist der Array-Zugriff erwartungsgemäß am schnellsten. Die Höhe der Verlustfaktoren bei den anderen Zugriffsarten läßt sich damit erklären, daß die zugehörigen Datenstrukturen (Vector, JGL-Array, Hash-Tabelle) als Implementierungen oberhalb der *Java Virtual Machine* (JVM) interpretiert, während Array-Zugriff sowie Zugriff auf Instanzenvariablen bereits direkt von Java angeboten werden.

Die Meßwerte geben bereits gewisse Auskünfte über die Leistungsunterschiede zwischen frühen und späten Bindungen. Ein früh gebundener Attributzugriff besteht zur Laufzeit lediglich aus einem Direktzugriff auf eine Feldposition (unabhängig davon, ob nun Array, Vector oder JGL-Array verwendet wurde). Der spät gebundene Zugriff kann erst zur Laufzeit den genauen Speicherungsort des angegebenen Attributs ermitteln, um dann auf den Wert zugreifen zu können. Letzteres geschieht gewöhnlich über eine Hash-Struktur. Somit weisen früh gebundene und spät gebundene Attributzugriffe mindestens die in Tabelle 1 ausgewiesenen Leistungsunterschiede auf.

Der Vergleich der ersten beiden Zeilen zeigt, daß der Overhead der Schleifenbearbeitung die Verlustfaktoren beeinflußt. So sind die in der zweiten Zeile der Tabelle angeführten Verlustfaktoren (für Vector-, JGL-Array und Hash-Map-Zugriff) jeweils geringer als die zugehörigen Werte der ersten Zeile. Es ist zu erwarten, daß bei einer weiteren Erhöhung des Overhead (der Schleifenbearbeitung) sich die Verlustfaktoren (für Vector-, JGL-Array und Hash-Map-Zugriff) noch weiter einander annähern (als bereits in Tabelle 1 ausgewiesen). Daher ist zu beachten, daß hoher Zusatzaufwand wie beispielsweise komplexe Auswertungen das Optimierungspotential früher Bindungen nicht uneingeschränkt zum Tragen kommen läßt.

14. ARRAY ist eine von Java direkt angebotene Datenstruktur, während Vector und JGL-Array oberhalb von Java definierte Klassen sind, die eine mehr oder weniger komfortable Unterstützung bieten, deren Detaillierung jedoch unseren Rahmen sprengen würde. Es sei lediglich betont, daß sich die Nutzung dieser Strukturen für unsere Zwecke anbietet.

Messungen, die nach dem Einsatz eines JIT-Compiler[15] durchgeführt wurden (siehe dritte Zeile in Tabelle 1), zeigten, daß der Array-Zugriff im Vergleich zur JVM-Interpretation nicht schneller wurde. Bei den anderen Zugriffen jedoch war eine Geschwindigkeitssteigerung um den Faktor zwei bei Vector (13/6) und JGL-Array(14/7) bis vier bei Hash-Map (53/13) erreichbar[16]. Dies weist darauf hin, daß die Verlustfaktoren bei zukünftigen Java-Versionen und stetig verbesserter Compiler-Technik weiter reduziert werden können, wenn auch der Unterschied in der Größenordnung nicht kompensierbar ist.

Tabelle 1 berücksichtigt nicht den Zugriff auf Instanzenvariablen, der unseren Messungen zufolge etwa zwei bis sechs mal schneller ist als der Array-Zugriff. Dies könnte bei der Abbildung von Attributen ausgenutzt werden, wenn anstelle von Arrays Instanzenvariablen eingesetzt werden.

Für den allgemeinen Fall liefert Tabelle 1 die Aussage, daß die optimale Nutzung von Java-Mechanismen (Array für die Realisierung früher Bindungen und Hash-Map für späte Bindungen) eine Verbesserung um den Faktor 53 (bei Nutzung der JVM) bzw. 13 (bei Verwendung eines JIT-Compilers) ermöglicht.

PRIMA/API

Für PRIMA/API sollen Attributzugriffsmethoden über Cursor und Atome einander gegenübergestellt werden. Aufrufe dieser Methoden im Anwendungsprogramm sehen etwa wie folgt aus:

- Spät gebundener Zugriff über Cursor:
  `cursor.getAttribute("AtomicPart", "x");`
- Früh gebundener Zugriff über Cursor:
  `cursor.getAtomicpart_x();`
- Spät gebundener Zugriff über bereits identifiziertes Atom: `atomicPart.getAttribute("x");`
- Früh gebundener Zugriff über bereits identifiziertes Atom: `atomicPart.getx();`

15. *JIT* steht für *just in time* und bedeutet in diesem Zusammenhang, daß der Java-Byte-Code nicht von der JVM interpretiert wird, sondern in Maschinencode der jeweiligen Plattform übersetzt und dann direkt ausgeführt wird.

16. Man beachte, daß sich diese Faktoren nur deshalb direkt aus Tabelle 1 ablesen lassen, da der durch die JVM ausgeführte Array-Zugriff genauso so lang dauerte wie die direkte Ausführung nach Verwendung eines JIT-Compilers. Daher sind die beiden unteren Zeilen der Tabelle auch direkt miteinander vergleichbar.

Tabelle 2 zeigt die entsprechenden Verlustfaktoren für den Zugriff über Molekül-Cursor sowie über bereits identifizierte Atome. Die erste Zeile enthält die Werte, die sich ohne Verwendung eines JIT-Compilers ergeben und die zweite Zeile die Werte nach Nutzung eines solchen. Obwohl aus der Tabelle aufgrund der unabhängigen Normierung der einzelnen Zeilen nicht direkt ablesbar, möchten wir anmerken, daß die absoluten Zeiten bei Verwendung des JIT-Compilers auch hier um den Faktor zwei bis drei besser sind.

Tabelle 2: Verlustfaktoren beim Zugriff mit PRIMA/API

	Molekül-Cursor		**Atom**	
	LATE	**EARLY**	**LATE**	**EARLY**
ohne JIT	73	14	53	1
mit JIT	58	8	51	1

Die folgenden Erläuterungen können an den Werten der zweiten Zeile der Tabelle nachvollzogen werden. Der spät gebundene Zugriff über den Molekül-Cursor geschieht wesentlich langsamer (Verlustfaktor 58) als der entsprechende früh gebundene Zugriff (Verlustfaktor 8). Letzterer erfolgt sogar deutlich schneller als der generische Zugriff über ein bereits identifiziertes Atom (Verlustfaktor 51)[17]. Ist ein Atom identifiziert, so ist der früh gebundene Zugriff über dieses Atom im Vergleich am schnellsten. Dies verspricht insbesondere dadurch einen sehr hohen Nutzen, daß das identifizierte Atom bzw. ein Verweis auf dieses im Cursor zwischengespeichert werden kann und so für wiederholte Attributzugriffe zur Verfügung steht.

Zusammenfassend verdeutlicht Tabelle 2, daß die generierte, früh gebundene API-Operation zum Attributzugriff auf ein identifiziertes Atom etwa um den Faktor 50 schneller ist als die spät gebundene Variante. Ein ähnlich hoher Faktor läßt sich auch für die entsprechenden API-Operationen zum Zugriff über den Molekül-Cursor beobachten.

Anwendung

Wir wollen nun untersuchen, inwieweit der durch Bindung erzielte Leistungsgewinn der PRIMA/API-Operationen der Anwendung zugute kommt. Als repräsentative Anfrage diene die bereits in Abschnitt 4.2 (siehe Abb. 6) dargestellte SELECT-Anweisung. Als beispielhafte Anwendung auf

17. Die eigentliche Identifikation des Atoms wurde hier noch nicht einmal berücksichtigt. Sie fällt in der Regel auch nur einmal pro Atom an, unabhängig von der Anzahl der Attributzugriffe auf das Atom.

dem Anfrageresultat haben wir die Auswertungsfunktion *Tap* (des OO7-Benchmarks [2]) implementiert, die durch Navigation in einer geschachtelten Schleife *CompositeParts* und die jeweils zugehörigen *AtomicParts* aufsucht. Tabelle 3 zeigt die durchschnittlichen Verlustfaktoren (später Bindungen gegen frühe Bindungen) für Moleküle mit 100 und 1.000 Atomen. Für die Mittelwertbildung wurden je drei Durchgänge á 20 Wiederholungen durchgeführt.

Um den Einfluß von Optimierungen, die sowohl die früh als auch die spät gebundene Schnittstelle betreffen, zu verdeutlichen, sind in Tabelle 3 zwei Spalten von Verlustfaktoren eingetragen. Die linke Spalte enthält die Werte bei eingeschaltetem und die rechte die Werte bei ausgeschaltetem *Pointer Swizzling*[18]. Bei eingeschaltetem *Pointer Swizzling* werden die vom Anwendungsprogrammlauf initiierten Dereferenzierungen nicht über assoziative Zugriffe auf eine Zuordnungstabelle (hier Hash-Tabelle) aufgelöst, sondern können durch einfaches Verfolgen von Hauptspeicherreferenzen abgewickelt werden.

Tabelle 3: Verlustfaktoren für die Operation *Tap*

	Cursor LATE / Cursor EARLY	**Cursor LATE / Cursor EARLY**
	Ø mit Pointer Swizzling	Ø ohne Pointer Swizzling
Molekül mit 100 Atomen	7,6	3,5
Molekül mit 1000 Atomen	7,2	3,4

Zunächst einmal ist zu beobachten, daß das Effizienzverhältnis relativ unempfindlich auf eine Erhöhung der Anzahl der Atome im Molekül reagiert (vergleiche Werte der beiden Tabellenzeilen).

Wird kein *Pointer Swizzling* durchgeführt, so ergeben sich wesentlich geringere Verlustfaktoren (3,4 bzw. 3,5) als im Falle der Nutzung des *Pointer Swizzling* (7,2 bzw. 7,6). Dies deutet darauf hin, daß die früh gebundene Variante stärker von Optimierungen wie *Pointer Swizzling* profitiert. Dafür sehen wir folgenden Grund. Wie bereits mehrfach angesprochen besteht der größte Teil der lokalen Verarbeitung auf dem Client aus Atom- und Attributzugriffen. Zur Abwicklung dieser ‘Arbeit’ führt die früh gebundene Variante im wesentlichen durch das *Pointer Swizzling* optimierte Dereferenzierungen und (Java-)Array-Zugriffe aus (siehe oben). Der Aufwand, den die spät gebundene Variante dagegen neben den notwendigen Dereferenzierungen treiben muß, liegt wesentlich höher (z. B. Hash-Map-Zugriffe statt Array-Zugriffe).

Dies verdeutlicht ein besonders gewinnbringendes Zusammenwirken früher Bindungen mit weiteren Optimierungsmaßnahmen, wie z. B. *Pointer Swizzling*.

Anmerkungen zum Bindungsgrad VIRTUAL_LATE

Die in diesem Kapitel erläuterten Experimente beschreiben im wesentlichen das Leistungspotential früh gebundener CLI-Operationen im Vergleich mit den traditionell spät gebundenen CLI-Operationen. Den Bindungsgraden EARLY und LATE haben wir in diesem Artikel jedoch auch den Bindungsgrad VIRTUAL_LATE als aussichtsreiche Variante hinzugefügt. Da dieser Bindungsgrad generische, d. h. spät gebundene, Schnittstellen mit früh gebundenen Implementierungen assoziiert, ist eine Effizienz zu erwarten, die in der Nähe des Leistungsverhaltens des Bindungsgrads EARLY liegt. Jedoch wird VIRTUAL_LATE immer ein geringfügig schlechteres Verhalten aufweisen als EARLY, da, wie bereits in Kapitel 5 beschrieben, alle möglichen Fälle der Parametrisierung der Operation in der Implementierung ‘vorhergesehen sein’ müssen und sich daher ein höherer Laufzeitaufwand ergibt als bei reiner früher Bindung.

8 Zusammenfassung

Generierte Aufrufschnittstellen für DBVS sind konfigurierbare Spezialisierungen klassischer Aufrufschnittstellen. Sie gestatten dem Anwender, Bindungsgrade für API-Operationen zu wählen. Neben den Bindungsgraden EARLY und LATE erlauben die Konzepte der Objektorientierung ein ganzes Spektrum an Bindungsgraden, wobei wir insbesondere den Bindungsgrad VIRTUAL_LATE für effektiv nutzbar halten. Durch die Möglichkeiten, schemaspezifische und anfragespezifische Typinformation für frühe Bindungen ausnutzen, verschiedene API-Operationen mit unterschiedlichen Bindungsgraden versehen und darüber hinaus dieselbe API-Operation mehrfach mit verschiedenem Bindungsgrad einsetzen zu können, ergibt sich eine sehr hohe Bindungsflexibilität, die die anwendungsspezifische Zugriffsoptimierung von DB-Anwendungen unterstützt.

Neben der Wahl des Bindungsgrades für einzelne API-Operationen erlaubt unser Ansatz der generierten Aufrufschnittstellen die Kombination der Bindungsgrade mit wei-

18. Bei den Messungen wurde *Eager Swizzling* durchgeführt.

teren Optimierungsmaßnahmen wie *Pointer Swizzling* und *Prefetching* von Verarbeitungskontexten. Diese Flexibilität erzielen wir durch den Einsatz von Generierung und Konfigurierung.

Das Optimierungspotential von frühen Bindungen im allgemeinen und im Zusammenhang mit *Pointer Swizzling* im speziellen haben wir anhand von Messungen evaluiert. Für einen speziellen Anwendungsfall (Operation *Tap* des OO7-Benchmarks) konnten wir einen Leistungsgewinn um den Faktor 7,5 messen. Da wir jedoch hinsichtlich unserer Implementierung (verstärkte Nutzung von Instanzenvariablen statt Java-Arrays zur Attributspeicherung, Verbesserung der internen Methoden zur Navigation zu Atomen im Cursor-Kontext) als auch bezüglich einer verstärkten Einbeziehung von anwendungsspezifischem Wissen (*Prefetching* von Verarbeitungskontexten) weitere Verbesserungsmöglichkeiten sehen, halten wir höhere Effizienzgewinne für realistisch. Insgesamt glauben wir, daß das Optimierungspotential generierter Aufrufschnittstellen mit einen Faktor 10 noch nicht vollständig ausgeschöpft ist.

Danksagung

Wir danken A. Lambert und H. Loeser für ihre Implementierungsarbeiten.

Literatur

[1] Atkinson, M., Morrison, R.: Orthogonally Persistent Object Systems. VLDB Journal 4(3), 319-401 (1995)

[2] Carey, M. J. , DeWitt, D. J., Naughton, J. F.: The OO7 Benchmark. SIGMOD Record 22(2), 12-21 (1993)

[3] Geiger, K.: Inside ODBC. Microsoft Press, Redmond, WA (1995)

[4] Gesman, M.: Parallele Anfrageverarbeitung in Komplexobjekt-Datenbanksystemen. Shaker, Aachen (1997)

[5] Hamilton, G., Cattell, R., Fisher, M.: JDBC[tm] Database Access with Java[tm] - A Tutorial and Annotated Reference. Computer & Eng. Publ. Group, The Java Series (1997)

[6] Härder, T., Meyer-Wegener, K., Mitschang, B., Sikeler, A.: PRIMA - A DBMS Prototype Supporting Engineering Applications. Proc. 13th Int. VLDB Conf., Brighton, UK, 433-442 (1987)

[7] Härder, T., Rahm, E.: Datenbanksysteme - Konzepte und Techniken der Implementierung. Springer, Berlin (1999)

[8] ISO: Final Committee Draft (FCD), Database Language SQL - Part 1: SQL/Framework. 1998.

[9] ISO: Working Draft, Database Language SQL - Part 2: SQL/Foundation (1998)

[10] Kemper, A., Kossmann, D.: Adaptable Pointer Swizzling Strategies in Object Bases. VLDB Journal 4(3), 519-566 (1995)

[11] Lacroix, M. Pirotte, A.: Comparison of Database Interfaces for Application Programming. Information Systems 8(3), 217-229 (1983)

[12] Mitschang, B.: Extending the Relational Algebra to Capture Complex Objects. Proc. 15th Int. VLDB Conf., Amsterdam, 297-305 (1989)

[13] Mitschang, B.: Anfrageverarbeitung in Datenbanksystemen - Entwurfs- und Implementierungskonzepte. Vieweg (1994)

[14] Moss, J.E.B.: Working with Persistent Objects: To Swizzle or Not to Swizzle. IEEE Trans. on Software Engineering 18(8), 657-673 (1992)

[15] McAuliffe, M. L., Solomon, M. H.: A Trace-Based Simulation of Pointer Swizzling Techniques. Proc. 11th Int. Conf. on Data Engineering, Taipei, Taiwan, 52-61 (1995)

[16] Neumann, K.: Kopplungsarten von Programmiersprachen und Datenbanksprachen. Informatik-Spektrum 15(4), 185-194 (1992)

[17] Nink, U.: Anbindung von Entwurfsdatenbanken an objektorientierte Programmiersprachen. Shaker, Aachen (1999)

[18] Nink, U., Härder, T., Ritter, N.: Generating Call-Level Interfaces for Advanced Database Application Programming with SQL3. Proc. 25th Int. VLDB Conf., Edinburgh, 575-586 (1999)

[19] Oracle Technical White Paper: SQLJ: Embedded SQL in Java (1997)

[20] Schöning, H.: Anfrageverarbeitung in Komplexobjekt-Datenbanksystemen. Deutscher Universitäts-Verlag, Wiesbaden (1993)

[21] Suzuki, S., Kitsuregawa, M., Takagi, M.: An Efficient Pointer Swizzling Method for Navigation Intensive Applications. Persistent Object Systems, M. Atkinson, D. Maier, V. Benzaken (eds.), Springer, Berlin, 79-95 (1995)

[22] Stonebraker, M., Brown, P.: Object-Relational DBMSs - Tracking the Next Great Wave. 2nd edition. Morgan Kaufmann, San Mateo, CA (1999)

[23] White,S.J., DeWitt, D.J.: A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. Proc. 18th Int. VLDB Conf., Vancouver, Morgan Kaufmann, San Mateo, CA, 419-431 (1992)