Minimal-Invasive Indexintegration

Transparente Datenbankbeschleunigung

Alexander Adam
dimensio informatics GmbH
Brückenstraße 4
09111 Chemnitz
alad@dimensio-informatics.de

Sebastian Leuoth
dimensio informatics GmbH
Brückenstraße 4
09111 Chemnitz
lese@dimensio-informatics.de

Wolfgang Benn
Technische Universität
Chemnitz
Straße der Nationen 62
09107 Chemnitz
benn@cs.tu-chemnitz.de

Keywords

Datenbankenerweiterung, Indizierung, Transparent

ZUSAMMENFASSUNG

Aktuelle Datenbanksysteme bieten dem Nutzer eine enorme Funktionsvielfalt [12, 8]. Selbst sehr spezielle Gebiete wie z. B. Geodatentypen [9, 14] werden unterstützt. Abhängig vom verwendeten Datenbanksystem können diese Fähigkeiten durch einen Nutzer noch erweitert werden. Beispiele hierfür wären Funktionen und nutzerdefinierte Datentypen [12, 8]. Alle diese Erweiterungen sollten natürlich nicht die Geschwindigkeit des Datenbanksystems negativ beeinflussen. So gibt es neben normalen Indexen auch Funktionsindexe und Indexe für nutzerdefinierte Datenhaltungen [20, 2]. Die Möglichkeiten, zu indizieren sind dabei, je nach Datenbankhersteller, vorbestimmt und selbst nicht erweiterbar. Allen diesen Techniken ist weiterhin gemein, dass sie direkt am Datenbanksystem ansetzen und teils auch in der Anfragesprache sichtbar sind. Es ist daher nicht einfach möglich, eine Anwendung, die fest vorgegeben ist, mittels solcher Techniken zu beschleunigen. In diesem Papier wollen wir eine Möglichkeit vorstellen, mit der eine solche Beschleunigung auch dann noch möglich ist, wenn weder das Datenbanksystem noch die Anwendung im Zugriff des Nutzers stehen. Verdeutlicht wird dieses am Beispiel der JDBC-Schnittstelle [15].

1. EINLEITUNG

Datenbanksysteme haben in den letzten Jahrzehnten ein breites Anwendungsspektrum erschlossen. Funktionalitäten wurden oft aufgrund der Bedürfnisse der Anwender hinzugefügt [17]. So gibt es heute nutzerdefinierte Datentypen, nutzerdefinierte Datenablagen und selbst elementare Dinge wie Indexe, die durch einen Nutzer in ihrer Struktur bestimmt werden.

Allen gemein ist eine gewisse Abhängigkeit der Implementation vom Datenbankhersteller und einer damit verbundenen schlechten Portabilität der entwickelten Module. Besonders im Bereich der Indizierung um den es in diesem Papier gehen soll, stellen sich aber noch weitere Probleme dar.

Normalerweise kann ein Index auf einer beliebigen Tabelle und deren Spalten definiert werden. Die Anwendung bemerkt nur insoweit etwas von dieser Aktion, als dass Anfragen schneller beantwortet werden sollten.

Ein kleines Beispiel soll dies verdeutlichen. Die folgende Anfrage gibt alle Mitarbeiterinformationen zurück, die zu Mitarbeitern mit mehr als 1000 Euro Gehalt gehören:

Listing 1: SQL-Anfrage mit Bedingung im `WHERE`-Teil

```
SELECT   *
FROM     mitarb
WHERE    mitarb.gehalt > 1000
```

Für die Anwendung, die diese Anfrage an die Datenbank stellt, ist es vollkommen unerheblich, ob sich auf der Gehaltsspalte der Mitarbeitertabelle ein Index befindet oder nicht. Die Anfrage würde, ob nun mit oder ohne Index, immer gleich aussehen.

Wir nehmen nun an, dass die Gehaltsspalte durch einen anderen Typ ersetzt werden soll, Gründe hierfür könnten eine effizientere Speicherung oder bessere Zugriffsmöglichkeiten sein. Wir nehmen weiter an, dass das Datenbanksystem für den neuen Typ den >-Operator nicht mehr anbietet. Es muss nun eine Vergleichsfunktion geschrieben werden, die die Aufgabe des >-Operators übernimmt. Diese tiefgreifende Umstrukturierung scheint nun bis in die Anfrage durch, ist also nicht mehr transparent:

Listing 2: SQL-Anfrage mit Anfrage an eine nutzerdefinierte Funktion im `WHERE`-Teil

```
SELECT   *
FROM     mitarb
WHERE    pruefe( mitarb.gehalt, 1000) = 1
```

Eine Anwendung, die der Nutzer nicht verändern kann, würde also von diesem neuen Typ ausschließlich dann profitieren, wenn der Hersteller diese Möglichkeit vorsieht oder standardmäßig einbaut. Für bestehende Umgebungen ist all dies also keine Option.

Im obigen Fall würde noch die Möglichkeit des automatischen SQL-Umschreibens einen Ausweg bieten [12, 8]. Dabei wird eine materialisierte Sicht angelegt. Anschließend wird der Datenbank mitgeteilt, dass bestimmte Anfragen nicht so gestellt werden sollen, wie sie der Anwender abgesetzt hatte,

24^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 29.05.2012 - 01.06.2012, Lübbenau, Germany.

sondern in veränderter Form auf der materialisierten Sicht abgearbeitet werden. Das Vorgehen muss vom Datenbanksystem aber auch unterstützt werden. Je nach Ausprägung müssen außerdem genaue Übereinstimmungsmerkmale angegeben werden, mit denen das Umschreiben ausgelöst wird. Scheitert das Umschreiben, weil es eine kleine Varianz in der Anfrage gibt, wird das Ergebnis u. U. falsch.

Im Folgenden werden wir eine Möglichkeit aufzeigen, wie es ohne einen Eingriff – weder bei der Anwendung noch bei der Datenbank – möglich ist, einen Index weitestgehend transparent in ein bestehendes System zu integrieren. Dazu wird zunächst untersucht, an welchen Stellen und wie in die Kommunikation von Anwendung und Datenbanksystem eingegriffen werden kann. Anschließend wird auf die Herausforderungen eingegangen, die sich bei dem hier genutzten Ansatz zeigen und wie diese gelöst werden können.

2. INTEGRATIONSPUNKTE

2.1 Überblick

Um mögliche Integrationspunkte zu finden, muss zunächst untersucht werden, wie eine Anwendung mit einem Datenbanksystem kommuniziert. Üblicherweise werden hierfür Datenbanktreiber eingesetzt. Das sind Programmbibliotheken, die die Anfragen in ein dem Datenbanksystem verständliches Protokoll überführen und dieses dann übermitteln. Der Datenbankserver dekodiert das Protokoll und arbeitet die darin enthaltenen Anweisungen ab.

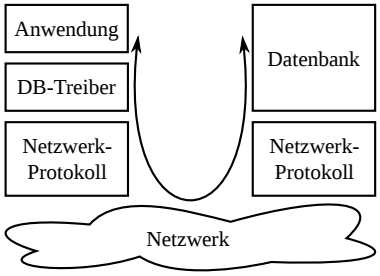

Abbildung 1: Schematische Darstellung des Zusammenspiels zwischen Anwendung und Datenbank. Die Anwendung verwendet ein API für einen Datenbanktreiber, welcher vom Datenbankhersteller zur Verfügung gestellt wird. Dieser Treiber ist dann für die Kommunikation mit dem Datenbanksystem über ein Netzwerk verantwortlich.

Das beschriebene Szenario – abgebildet in Abbildung 1 – offenbart drei Möglichkeiten, eine Integration vorzunehmen:

- den Datenbanktreiber,
- die Kommunikation über das Netzwerk und
- das Datenbanksystem selbst.

Im Folgenden werden wir uns auf den Datenbanktreiber, also die Anwendungsseite, konzentrieren. Die Vorgehensweise bei der Integration in die Kommuniktion ist bereits in [1] beschrieben. Möglichkeiten, ein Datenbanksystem zu erweitern, wurden in den letzten Jahrzehnten vielfach an anderer Stelle beschrieben [4, 20, 2].

2.2 Datenbanktreiber

Der Datenbanktreiber ist die Schnittstelle für einen Anwendungsprogrammierer, um mit dem Datenbanksystem zu kommunizieren. Er stellt Funktionen bereit, um Verbindungen zu verwalten und Datenbankanfragen abzuarbeiten (sei es nun SQL oder irgendeine andere Art der Anfrage) [5, 11]. Es ist wichtig, zu beobachten, dass dabei jeder Abarbeitungsschritt von der Anwendung ausgelöst wird. Alle Ergebnisse einer Anfrage werden nicht einfach als Resultat einer `query()`-Funktion zurückgegeben, sondern müssen aktiv angefordert werden. Eine typische Schrittfolge, die eine Anwendung verwenden könnte, ist in Listing 3 aufgezeigt.

Listing 3: Mögliche, stark reduzierte, Schrittfolge bei der Nutzung einer Datenbankbibliothek, hier JDBC. Es wird zunächst eine Verbindung geöffnet, dann ein Statement mit ungebundenen Variablen vorbereitet und gebunden. Schließlich werden die angefragten Daten abgeholt.

```
Connection conn = DriverManager.getConnection(
                  "jdbc:mysql://localhost/testdb",
                  "username",
                  "password");
Statement ps = conn.prepareStatement(
                  "SELECT * FROM mitarb
                   WHERE gehalt > ?");
ps.setInt(1, 1000);
ResultSet rs = ps.executeQuery();
String name = rs.getString("name");
ps.close();
conn.close();
```

Wie nun stellt sich hier eine Möglichkeit zur Integration dar? Es ist möglich, vor jede Funktion, die eine Anwendung vom originalen Datenbanktreiber aufruft, eine eigene Funktion zu setzen. Die Anwendung ruft nun die eigene Funktion, ohne dies zu bemerken, und die eigene Funktion ruft schließlich die originale. Da nun alle Daten, die von einer Anwendung zum Datenbanksystem gesendet werden, vorher analysiert und verändert werden können, stellt sich so dieser Integrationspunkt dar. Außerdem können eigene Funktionen auf der so abgefangenen Datenbankverbindung „huckepack“ aufgesetzt werden.

Ein erster Gedanke, dies zu realisieren, könnte in die Richtung einer *DLL-Injection* [18] gehen. Das bedeutet das komplette Ersetzen des vom Datenbankhersteller bereitgestellten Treibers durch einen eigenen. Dieser kann, da die Protokolle nicht zwingend offengelegt sein müssen, die Kommunikation mit dem Datenbanksystem nicht selbst übernehmen, sondern ruft den ursprünglichen Datenbanktreiber. Abhängig von der Anzahl der zu implementierenden Funktionen kann dies ein möglicher Weg sein. Eine weit verbreitete solche Schnittstelle, um aus einer Javaanwendung mit einem Datenbanksystem in Verbindung zu treten, ist JDBC. Mit seinen vielen Klassen und mehr als 1000 zugehörigen Methoden [6] wäre es eine sehr langwierige Aufgabe, einen solchen sogenannten Wrapper [7] zu implementieren. In einem unserer Produkte namens *Cardigo* ist dieses aber bereits implementiert und erleichtert so die Aufgabe enorm. Weitere Projekte und Arbeiten zu diesem Thema sind unter anderem auch bei [19, 22, 21] und [13] zu finden.

Cardigo ist ein Rahmenwerk, welches u. a. alle Klassen und Methoden enthält, die das JDBC-API anbietet. Anstatt selbst ein kompletter Treiber zu sein, bietet es lediglich einen Wrapper für einen „richtigen“ Datenbanktreiber. Das Ziel dieses Produktes ist es, einem Anwender die Möglichkeit zu geben, die Datenbankkommunikation – in diesem Falle hauptsächlich Anfragen, Ergebnisse u. s. w. – zu loggen oder gar zu verändern. Mit diesem Werkzeug können wir nun die Integration des Index angehen.

Technisch ist zur Integration von Cardigo nichts weiter nötig, als ein geänderter Verbindungsparameter für die Anwendung. Dieser bewirkt, dass anstelle des originalen JDBC-Treibers nun Cardigo geladen wird. Der Aufbau dieser veränderten Umgebung ist in Abbildung 2 verdeutlicht.

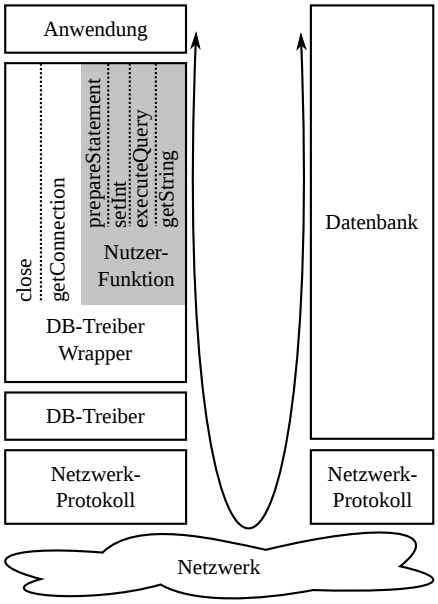

Abbildung 2: Anwendung, die Cardigo durch einen veränderten Kommunikationsparameter nutzt. Der Code, den ein Nutzer schreibt, umfasst typischerweise nicht den gesamten API-Umfang von JDBC, weshalb er hier nicht über die gesamte Breite dargestellt ist. Exemplarisch sind die in Listing 3 verwendeten Methoden dargestellt.

3. INTEGRATION

Bevor wir die Integration des Index weiter betrachten, wollen wir kurz darauf eingehen, wie ein in ein Datenbanksystem integrierter Index arbeitet: Wird eine Anfrage an die Datenbank gestellt und treffen einige der verwendeten Attribute die im Index enthaltenen, so wird der Index verwendet. Dabei wird die Anfrage vom Index bearbeitet und im Ergebnis entsteht einen Liste von Ergebniskandidaten. Diese können in Form von RowIDs [3] oder auch einfach als Primärschlüssel vorliegen. Das Datenbanksystem überprüft dann nur noch die Datensätze, deren Identifikatoren der Index als Ergebnis lieferte. Auf diese Art wird der Aufwand, den das Datenbanksystem beim Laden der Datensätze und ihrer Verifikation hat, erheblich verringert [10].

Im Folgenden wird zunächst die *logische Integration* betrachtet, d. h., wie es überhaupt möglich ist, Ergebnisse eines Index an das Datenbanksystem zu übertragen. Es schließt sich eine Betrachtung an, die die *programmtechnische Integration* betrachtet, da sich hier noch einige weitere Probleme auftun.

3.1 Logische Integration

Die Grundidee der Integration ist es nun, die Anfrage, die eine Anwendung absetzt, so zu verändern, dass sie das Verhalten eines datenbankinternen Index nachahmt. Die Anfrage aus Listing 1 könnte nun, wie in Listing 4 aufgezeigt, um das Primärschlüsselattribut erweitert werden.

Listing 4: SQL-Anfrage, die um die Ergebnisse eines externen Index erweitert wurde

```
SELECT   *
FROM     mitarb
WHERE    gehalt > 2000
  AND    id IN (4, 18)
```

Dieses Vorgehen vertraut darauf, dass der Optimierer des Datenbanksystems erkennt, dass es sich bei den Werten in der `IN`-Klausel um Primärschlüssel handelt. Er muss diese möglichst am Anfang der Anfrageverarbeitung einbeziehen, um die Menge der zu untersuchenden Tupel zu minimieren.

Durch Tests haben wir herausgefunden, dass die Anzahl der in `IN`-Listen enthaltenen Elemente eine Obergrenze hat. Durch die Disjunktion mehrerer `IN`-Listen kann diese zu einem gewissen Grad ausgeglichen werden. Somit ist es möglich, sehr lange Anfragen zu erzeugen, Allerdings gibt es auch eine Grenze für die maximale, vom Datenbanksystem zugelassene, Anfragelänge. Bei IBM DB2 und Oracle ist diese Maximallänge bspw. 64kB [8, 16].

Ein weiterer Aspekt, der bei diesen langen Anfragen betrachtet werden muss, ist, dass diese vom Datenbanksystem auch geparst werden. Werden die Anfragen lang, so steigen auch deren Parsezeiten, selbst optimistisch betrachtet ist dieser Zusammenhang linear. Auch systematisch bedingt, ist, dass der Optimierer zu lange und viele `IN`-Listen nicht beachtet, selbst wenn es sich um Primärschlüssel handelt. Mit Hinweisen an den Optimierer kann hier gegengesteuert werden. Zusammenfassend lässt sich aber sagen, dass ab einer gewissen Anfragelänge, der Gewinn durch den Index sich im schlechtesten Falle sogar ins Gegenteil verkehren kann.

Um diese Beschränkungen zu umgehen, können die Ergebnisse des Index auch in eine Tabelle in der Datenbank eingefügt werden. Hier gibt es wieder mehrere Möglichkeiten:

1. Einfügen in eine Tabelle, die angelegt ist
2. Einfügen in eine temporäre Tabelle

In beiden Fällen muss allerdings die Datenbank in der Weise modifiziert werden, dass die Anwendung, in die der Index integriert wurde, das Recht hat, auf diese Tabellen zu schreiben. Die Tabellen müssen prinzipiell eine Spalte für die Anfrage und eine Spalte für die Ergebnisse des Index besitzen.

Werden über eine Sitzung Anfragen nur sequenziell bearbeitet, haben temporäre Tabellen den Vorteil, dass zwischen verschiedenen Datenbanksitzungen nicht mittels der eben beschriebenen zusätzlichen AnfrageID-Spalte in dieser

Tabelle unterschieden werden muss. Das ist darin begründet, dass temporäre Tabellen für jede Sitzung als leere neue Tabellen erscheinen. Die Aktionen einer Sitzung wirken sich nicht auf den Inhalt der temporären Tabelle in einer anderen Sitzung aus.

Ein bisher nicht zur Sprache gekommener Punkt ist das Aktualisieren des Index. Natürlich muss ein datenbankexterner Index über `INSERT`-, `UPDATE`- und `DELETE`-Operationen informiert werden. Der einfachste Weg ist, wenn alle Operationen, die auf der Datenbank laufen, über die gleiche abgefangene Schnittstelle gehen und so direkt gelesen werden können. In der Realität ist dies jedoch unpraktisch, da diese Voraussetzung nicht zu 100% gewährleistet werden kann. Trigger sind eine Variante, wie Veränderungen in der Datenbank nach außen gereicht werden können, diese müssen jedoch integriert werden dürfen. Hier ergeben sich damit auch Grenzen, über die hinaus unser Ansatz nicht angewendet werden kann. Eine andere Möglichkeit sind statische Datenhaltungen, die nur in definierten Zeitabschnitten aktualisiert werden, bspw. einmal pro Monat. Hier kann ein statischer Index genutzt werden, der über eine simple Zeitsteuerung aktualisiert wird.

3.2 Programmtechnische Integration

Oft halten sich Anwendungsprogrammierer an die Beispiele der Autoren der jeweiligen Datenbankschnittstelle. Jedoch erlauben alle APIs auch eine freiere Nutzung, d. h., dass die Schrittfolge der Kommandos nicht festgelegt ist und es verschiedene Gründe geben kann, die so aufgezeigten Standardroutinen zu verwerfen. Listing 3 zeigt zunächst einen Standardweg auf. Für diesen wollen wir nun die Schritte, die ein datenbankexterner Index verfolgen kann, aufzeigen:

- Statement vorbereiten (`conn.prepareStatement(...)`):
  if Anfrage relevant **then**
  Anfrage speichern
  end if
- Variablen binden (`ps.setInt(...)`):
  if Anfrage war relevant **then**
  Bindung speichern
  end if
- Statement ausführen (`ps.executeQuery()`):
  if Anfrage war relevant **then**
  Index anfragen
  Indexergebnisse in DB laden
  Anfrage modifizieren
  Datenbank anfragen
  end if
- Ergebnisse holen (`rs.getString(...)`):
  hier sind keine weiteren Schritte notwendig

Beobachtungen an realen Programmen zeigen, dass das Binden von Variablen teils mehrfach auf die gleichen Variablen angewendet wird. Mit dem eben beschriebenen Verfahren ist dies kein Problem. Auch Operationen, die ein Statement näher beschreiben, sind nach wie vor ausführbar, da alle Bestandteile erhalten bleiben und nur Ergänzungen vorgenommen werden.

4. ERGEBNISSE UND AUSBLICK

Das hier beschriebene System war bereits erfolgreich im Einsatz. Die Latenzen für das reine Abfangen der Datenbankaufrufe bewegen sich im einstelligen Mikrosekundenbereich, also dem, was für einen Funktionsaufruf erwartet werden kann. Hinzu kommt die Zeit für die Indexanfrage. Für einen realen Gewinn muss natürlich die Zeit für die Integration, die Indexanfrage und den Einbau der Ergebnisse in das Statement in Summe geringer sein, als die einer Anfrage ohne den externen Index.

Es zeigte sich auch, dass, wird eine Integration über Tabellen angestrebt, es verschiedene Arten gibt, die Ergebnisse aus der Ergebnistabelle zu entfernen. In 3.1 wurden die verschiedenen Arten von Tabellen hierfür beschrieben. Wenn keine Spalte für die AnfrageID verwendet werden muss, so können die Ergebnisse mit einem `TRUNCATE` entfernt werden. Dieses wird erheblich schneller ausgeführt, als ein `DELETE FROM ... WHERE anfrage_id == <current_id>`.

Unsere weitere Arbeit beschränkt sich nicht nur auf eine Integration in JDBC, die wir hier aufgezeigt haben, sondern geht auch darüberhinaus auf native Datenbanktreiber ein, die dann jedoch herstellerspezifisch sind. Hier müssen andere Mechanismen angewandt werden, eine Integration elegant zu vollziehen, die Prinzipien bleiben jedoch die gleichen. Auch der bereits vorgestellte Ansatz, einen Proxy zu integrieren, der das Protokoll, welches das Datenbanksystem im Netzwerk verwendet, versteht, wurde weitergeführt und zeigte sich bereits im Einsatz als wertvolle Hilfe. Das oben bereits erwähnte Cardigo dient uns dabei als Werkzeugkasten, der alle diese Möglichkeiten vereint.

5. LITERATUR

[1] A. Adam, S. Leuoth, and W. Benn. Nutzung von Proxys zur Ergänzung von Datenbankfunktionen. In W.-T. Balke and C. Lofi, editors, *Grundlagen von Datenbanken*, volume 581 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2010.

[2] E. Belden, T. Chorma, D. Das, Y. Hu, S. Kotsovolos, G. Lee, R. Leyderman, S. Mavris, V. Moore, M. Morsi, C. Murray, D. Raphaely, H. Slattery, S. Sundara, and A. Yoaz. *Oracle Database Data Cartridge Developers Guide, 11g Release 2 (11.2)*. Oracle, July 2009.

[3] P. Bruni, F. Bortoletto, R. Kalyanasundaram, G. McGeoch, R. Miller, C. Molaro, Y. Ohmori, and M. Parbs. *DB2 10 for z/OS Performance Topics*. IBM Form Number SG24-7942-00, IBM Redbooks, June 2011.

[4] Desloch et al. PatNr. US 6,338,056 B1 – Relational Database Extender that Supports User-Defined Index Types and User-Defined Search, Apr. 1999.

[5] R. Elmasri and S. B. Navathe. *Grundlagen von Datenbanksystemen (3. Aufl., Bachelorausgabe)*. Pearson Studium, 2009.

[6] M. Fisher, J. Ellis, and J. C. Bruce. *JDBC API Tutorial and Reference*. Pearson Education, 3 edition, 2003.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Boston, MA, January 1995.

[8] IBM. *SQL Reference, Volume 1*. IBM Corporation, Nov. 2009.

[9] IBM Deutschland GmbH. *DB2 Spatial Extender und Geodetic Data Management Feature – Benutzer- und Referenzhandbuch*, July 2006.

[10] T. Lahdenmäki and M. Leach. *Relational database index design and the optimizers: DB2, Oracle, SQL server, et al*. Wiley-Interscience, 2005.

[11] T. Langner and D. Reiberg. *J2EE und JBoss: Grundlagen und Profiwissen ; verteilte Enterprise-Applikationen auf Basis von J2EE, JBoss & Eclipse*. Hanser, 2006.

[12] D. Lorentz and M. B. Roeser. *Oracle Database SQL Language Reference, 11g Release 2 (11.2)*. Oracle, Oct. 2009.

[13] A. Martin and J. Goke. P6spy. `http://sourceforge.net/projects/p6spy/`.

[14] C. Murray. *Oracle Spatial Developers Guide, 11g Release 2 (11.2)*. Oracle, Dec. 2009.

[15] G. Reese. *Database Programming with JDBC and Java, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2nd edition, 2000.

[16] B. Rich. *Oracle Database Reference, 11g Release 2 (11.2)*, Sept. 2011.

[17] G. Saake, K.-U. Sattler, and A. Heuer. *Datenbanken: Konzepte und Sprachen, 3. Auflage*. mitp-Verlag, Redline GmbH, 2008.

[18] J. Shewmaker. Analyzing dll injection, 2006. GSM Presentation, Bluenotch.

[19] M. Smedberg. Boilerplate JDBC Wrapper. `http://blog.redfin.com/devblog/2011/03/boilerplate_jdbc_wrapper.html`.

[20] K. Stolze and T. Steinbach. DB2 Index Extensions by example and in detail, IBM Developer works DB2 library. Dec. 2003.

[21] Thedwick, LLC. jdbcgrabber. `http://code.google.com/p/jdbcgrabber/`.

[22] C. Wege. Steps out of Integration Hell - Protocol Interception Wrapper. In A. Rüping, J. Eckstein, and C. Schwanninger, editors, *EuroPLoP*, pages 455–458. UVK - Universitaetsverlag Konstanz, 2001.