Generierung Relevanter Testdatenbanken

Carsten Binnig

Systems Group, ETH Zurich
carsten.binnig@inf.ethz.ch

Abstract: In heutigen Softwareentwicklungsprojekten ist das Testen eine der kosten- und zeitintensivsten Tätigkeiten. Wie ein aktueller Bericht des NIST [RTI02] zeigt, verursachten Softwarefehler in den USA im Jahr 2000 zwischen $22,2$ und $59,5$ Milliarden Dollar an Kosten. Demzufolge wurden in den letzten Jahren verschiedene Methoden und Werkzeuge entwickelt, um diese hohen Kosten zu reduzieren. Viele dieser Werkzeuge dienen dazu die verschiedenen Testaufgaben (z.B. das Erzeugen von Testfällen, die Ausführung von Testfällen und das Überprüfen der Testergebnisse) zu automatisieren. Jedoch existieren nur wenige Forschungsarbeiten zur Automatisierung der Tests von Datenbankanwendungen (wie z.B. eines E-Shops) bzw. von relationalen Datenbankmanagementsystemen (DBMS).

Die diesem Artikel zugrunde liegende Doktorarbeit diskutiert ein wichtiges Problem aus diesem Bereich: Die *Generierung von Testdatenbanken*. Zur Erzeugung einer Testdatenbank existieren verschiedene Forschungsprototypen sowie auch kommerzielle Datenbankgeneratoren. Jedoch sind dies meist Universallösungen, welche die Testdatenbanken unabhängig von den auszuführenden Testfällen erzeugen. Demzufolge weisen die generierten Testdatenbanken meist nicht die notwendigen Datencharakteristika auf, die zur Ausführung bestimmter Testfälle notwendig sind. Im Gegensatz zu diesen Werkzeugen beschreibt dieser Artikel innovative Ansätze, die zur Generierung von relevanten Testdatenbanken dienen. Die generelle Idee ist, dass der Benutzer explizit für einen oder mehrere Testfälle die notwendigen Bedingungen an die Testdaten deklarativ formulieren kann. Diese Bedingungen werden dann dazu genutzt, um eine Testdatenbank zu generieren, die die gewünschten Datencharakteristika aufweist, welche zur Ausführung der Testfälle notwendig sind.

1 Einführung

In heutigen Softwareentwicklungsprojekten ist das Testen eine der kosten- und zeitintensivsten Tätigkeiten. Wie ein aktueller Bericht des NIST [RTI02] zeigt, verursachten Softwarefehler in den USA im Jahr 2000 zwischen $22,2$ und $59,5$ Milliarden Dollar an Kosten. Demzufolge wurden in den letzten Jahren verschiedene Methoden und Werkzeuge entwickelt, um diese hohen Kosten zu reduzieren. Viele dieser Werkzeuge dienen dazu, die verschiedenen Testaufgaben (z.B. das Erzeugen von Testfällen, die Ausführung von Testfällen und das Überprüfen der Testergebnisse) zu automatisieren.

Jedoch existieren nur wenige Forschungsarbeiten zur Automatisierung der Tests von Datenbankanwendungen (wie z.B. eines E-Shops) bzw. von relationalen Datenbankmanagementsystemen (DBMS). Hierzu sind neue Lösungen erforderlich, da das Verhalten der zu

testenden Anwendung bzw. des Datenbankmanagementsystems sehr vom Inhalt der Datenbank abhängig ist. Folglich ergeben sich für den Test von Datenbankanwendungen oder von Datenbankmanagementsystemen neue Probleme und Herausforderungen im Vergleich zum traditionellen Testen von Anwendungen ohne Datenbank. In [HKL07] wird beispielsweise gezeigt, dass traditionelle Ansätze zur Testfallausführung nicht optimal sind für Datenbankanwendungen und dass spezielle Scheduling-Algorithmen, die den Zustand der Datenbank beachten, die Ausführungszeit wesentlich reduzieren können.

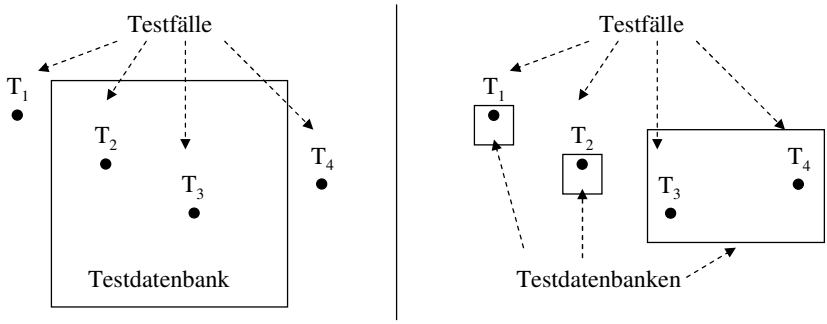
Die diesem Artikel zugrunde liegende Doktorarbeit [Bin08] diskutiert ein bestimmtes Problem aus diesem Bereich: Die *Generierung von Testdatenbanken*. Die Generierung von Testdatenbanken ist eine der entscheidenden Aufgaben für den funktionalen Test von Datenbankanwendungen und Datenbankmanagementsytemen (im weiteren Verlauf als *Testobjekt* bezeichnet), da ein bestimmtes Verhalten sich nur mit Hilfe von Testdaten überprüfen lässt, die bestimmte Eigenschaften erfüllen.

Ein einfaches Beispiel hierfür ist die Anmeldefunktion einer Web-Anwendung, die Benutzer sperrt welche schon mehr als drei Fehlversuche mit falschem Passwort ausgeführt haben. Um diese Funktion sorgfältig zu testen, ist eine Testdatenbank erforderlich, die Benutzer mit weniger als drei Fehlversuchen enthält (für den positiven Fall) bzw. auch Benutzer mit mehr als drei Fehlversuchen (für den negativen Fall).

Ein weiteres Beispiel ist das Testen des Anfrage-Optimierers eines DBMS. Das Ziel des Anfrage-Optimierers ist es, einen Ausführungsplan mit minimaler Ausfürungszeit für eine gegebene Anfrage und eine gegebene Datenbank zu erzeugen. Der Optimierer verwendet für seine Entscheidungen Statistiken der Datenbank und leitet daraus die Kardinalitäten der Zwischenergebnisse ab, um z.B. die Reihenfolge der Verbund-Operatoren festzulegen bzw. auch für die Auswahl von physischen Ausführungsoperatoren. Folglich, um eine bestimmte Funktionalität des Anfrage-Optimierers zu testen, ist es notwendig, Testdatenbanken zu generieren, die für eine gegebene Anfrage bestimmte Kardinalitäten der Zwischenergebnisse liefern.

Zur Erzeugung einer Testdatenbank existieren bereits verschiedene Forschungsprototypen wie auch kommerzielle Datenbankgeneratoren. Jedoch sind die existierenden Datenbankgeneratoren meist Universallösungen, welche die Testdatenbanken unabhängig von den auszuführenden Testfällen erzeugen. Demzufolge weisen die generierten Testdatenbanken meist nicht die notwendigen Charakteristika auf, die zur Ausführung von bestimmten Testfällen notwendig sind (siehe Abbildung 1 a).

Im Gegensatz zu diesen Ansätzen beschreibt dieser Artikel innovative Ansätze (*Reverse Query Processing* in Abschnitt 2, *Multi Reverse Query Processing* in Abschnitt 3 und *Symbolic Query Processing* in Abschnitt 4), die zur Generierung von Testdatenbanken für unterschiedliche Anwendungsfelder dienen. Die generelle Idee aller Ansätze ist es, dass der Benutzer explizit für einen oder mehrere Testfälle die notwendigen Bedingungen an die Testdaten deklarativ formulieren kann. Diese Bedingungen werden dann dazu genutzt, um eine oder mehrere Testdatenbanken zu generieren, die die gewünschten Datencharakteristika aufweisen, welche zur Ausführung der Testfälle notwendig sind (siehe Abbildung 1 b).



(a) Zufällige Testdatenbanken (b) Testfallspezifische Testdatenbanken

Abbildung 1: Ansätze zur Generierung von Testdatenbanken

2 Reverse Query Processing

Traditionell ist die Eingabe bei der Anfragebearbeitung eines DBMS die Datenbank D sowie die SQL-Anfrage Q. Die Ausgabe der Anfragebearbeitung ist das Ergebnis R (d.h. $Q(D) = R$). Die grundlegende Idee von *Reverse Query Processing* (RQP) ist es, den Datenfluss der Anfragebearbeitung umzudrehen, d.h. die Eingabe von RQP ist die SQL-Anfrage Q und ein mögliches Ergebnis R der Anfrage sowie das Datenbankschema S (inklusive der Integritätsbedingungen). Die Ausgabe ist eine Datenbankinstanz D, die die Eigenschaft hat, dass sie das definierte Ergebnis für die gegebene Anfrage liefert (d.h. $Q(D) = R$) und dass die Instanz D dem Datenbankschema S entspricht.

Die Hauptanwendung von RQP ist die Generierung von Testdatenbanken. Die Idee ist, dass die Anfrage Q und das Ergebnis R eine Sicht auf die relevanten Daten einer Datenbank definieren, die zur Ausführung eines Testfalls notwendig sind. Für OLAP-Anwendungen besteht z.B. die Spezifikation der Testdatenbank direkt aus der Anfragedefinition Q einer multidimensionalen Analyse (auf einer bestimmten Aggregationsstufe), dem Schema S des Cubes und einem möglichen Analyseergebnis R. In [BKL07] haben wir gezeigt, dass RQP für alle Anfragen des TPC-H Benchmarks [TPC] eine geeignete Testdatenbank generieren kann. Für OLTP-Anwendungen ist es häufig notwendig, mehr als eine Anfrage und ein Ergebnis zur Spezifikation der Testdatenbank zu verwenden, was in Abschnitt 3 beschrieben wird. Weitere Anwendungen werden in der diesem Artikel zugrunde liegenden Doktorarbeit [Bin08] beschrieben[1].

Prinzipiell können unterschiedliche Datenbankinstanzen D existieren, die für eine gegebene Anfrage Q ein Ergebnis R zurückliefern. In Abhängigkeit vom Anwendungskontext sind bestimmte Instanzen besser geeignet als andere.Für den funktionalen Test einer Datenbankanwendung macht es beispielsweise Sinn, dass RQP eine möglichst kleine Testdatenbank erzeugt, um die Laufzeit der Testausführung zu minimieren. Für Performanztests ist hingegen eine möglichst große Datenbankinstanz sinnvoll. Für andere Anwendungen können wiederum Testdatenbanken mit völlig anderen Eigenschaften besser geeignet sein.

[1] RQP wird aktuell bei Microsoft SQL Server zur Generierung von Testdatenbanken eingesetzt [BKLSB08]

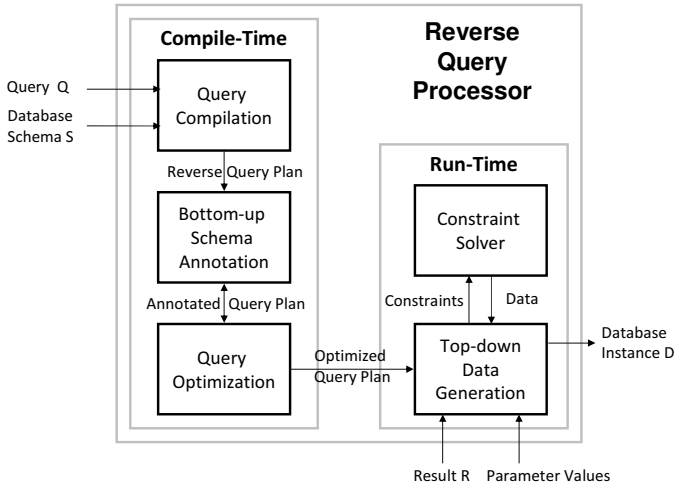


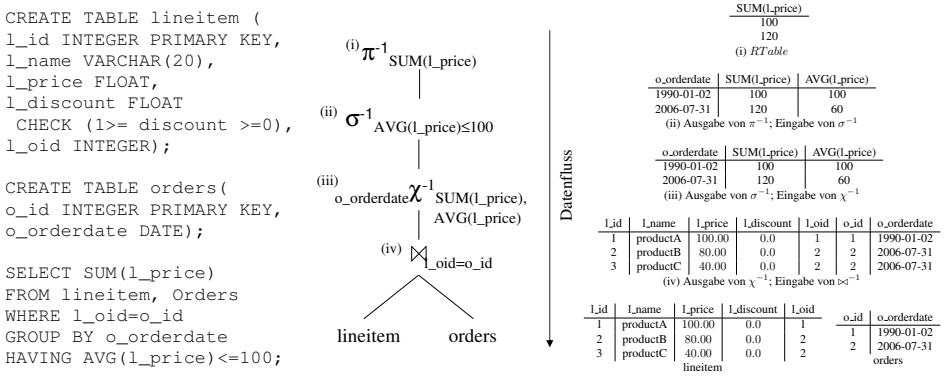
Abbildung 2: RQP Architektur

Abbildung 2 zeigt die Architektur unseres RQP Prototyps wie in [BKL07] beschrieben. In vielen Punkten ist die Architektur an die eines traditionellen Anfrage-Prozessors eines DBMS angelehnt: Eine SQL-Anfrage wird zuerst geparst und in einen relationalen Ausdruck transformiert, danach optimiert und ausgeführt. Der Hauptunterschied bei RQP liegt darin, dass (1) an Stelle der relationalen Algebra eine revers relationale Algebra verwendet wird, (2) die Optimierungsregeln unterschiedlich sind und (3) dass neue Algorithmen zur Ausführung der revers relationalen Operatoren notwendig sind. Außerdem wird nach dem Parsern ein zusätzlicher Schritt (die Bottom-up Annotation) eingefügt, welcher Informationen aus dem Datenbankschema in den revers relationalen Ausdruck propagiert (was zur Ausführung des Ausdrucks notwendig ist). Weitere Details zu dieser Phase und zur Implementierung sind in [BKL07, Bin08] zu finden.

Die revers relationale Algebra kann allgemein als eine reverse Variante der relationalen Algebra gesehen werden (mit Erweiterungen für den Gruppierungs- und Aggregationsoperator [GMUW01]). Die Ausführung eines revers relationalen Ausdrucks propagiert dementsprechend das Ergebnis von der Wurzel des Ausdrucks[2] zu den Blättern und erzeugt dadurch neue Daten in der Datenbank.

Die revers relationale Aggregation als Beispiel erzeugt neue Zeilen, wobei die relationale Aggregation mehrere Eingabezeilen eines Zwischenergebnisses zu einer Zeile zusammenfasst. Um neue Daten zu generieren, die den Bedingungen der Anfrage (z.B., einem `WHERE`-Prädikat) und den Integritätsbedingungen des Datenbankschemas entsprechen, wird ein Modellprüfer von einzelnen revers relationalen Algebraoperatoren aufgerufen. Ein Beispiel für die Prozessierung eines kompletten reverse relationalen Ausdrucks wird in Abbildung 3 gezeigt.

[2] Wie in der relationalen Algebra kann ein revers relationaler Ausdruck als Baum dargestellt werden.



(a) Datenbankschema und SQL-Anfrage

(b) Revers Relationaler Algebraausdruck

(c) Ein- und Ausgabe der Operatoren

Abbildung 3: RQP Beispiel für OLAP-Anwendungen

Theoretisch ist nicht entscheidbar, ob für jede mögliche SQL-Anfrage und deren Ergebnis euch eine Datenbank existiert [Bin08]. In der Praxis kann RQP aber für die meisten SQL-Anfragen und deren Ergebnis eine entsprechende Datenbankinstanz erzeugen. Wie schon zu Beginn dieses Abschnittes erwähnt, haben wir in [BKL07] gezeigt, dass sich für alle Anfragen des TPC-H Benchmarks [TPC] eine geeignete Testdatenbank generieren lässt. Bei Anfragen mit komplexen Aggregationsfunktionen ist der Aufwand zur Prozessierung entsprechend hoch. In den Experimenten in [BKL07] variierte die Bandbreite zur Generierung der Testdatenbank auf einem Linux AMD Opteron 2.2 GHz Server mit 4 GB Hauptspeicher von 600GB pro Stunde im besten Fall bis zu nur 100MB pro Stunde im schlechtesten Fall.

3 Multi Reverse Query Processing

Im Gegensatz zu OLAP-Anwendungen, die in der Regel große korrelierte Datenbestände von der Datenbank lesen, implementieren OLTP-Anwendungen häufig prozessorientierte Anwendungsfälle, die in einzelnen Schritten kleine Datenmengen von der Datenbank lesen bzw. ändern. Ein Beispiel hierfür ist eine Bibliotheksanwendung. Ein möglicher Anwendungsfall einer Bibliotheksanwendung ist, dass der Benutzer mit Hilfe einer ISBN ein Buch sucht, das er gerne ausleihen möchte. Der Prozess für diesen Anwendungsfall könnte wie folgt sein.

1. Der Benutzer gibt die ISBN des Buches ein.
2. Das System zeigt Details des Buches (Titel, Autor) an.
   - Ausnahme 1: Alle Bücher mit der ISBN sind bereit verliehen. Abbruch.
   - Ausnahme 2: Kein Buch mit dieser ISBN steht im Freihandbereich . Abbruch.
3. Der Benutzer gibt seine Benutzerdaten ein (Benutzername, Passwort).

4. Das System überprüft die Benutzerdaten und schließt die Reservierung ab.
    - Ausnahme 3: Die Anmeldedaten sind nicht korrekt. Abbruch.
    - Ausnahme 4: Der Benutzer hat zu hohe Mahngebühren. Abbruch.

Um die Funktionalität eines solchen Anwendungsfalles zu testen, ist es notwendig, das tatsächliche Verhalten der Anwendung mit dem erwarteten Verhalten zu vergleichen [Bin99]. Folglich sind unterschiedliche Testfälle notwendig, die das Verhalten der verschiedenen Ausführungspfade überprüfen. Um die einzelnen Testfälle ausführen zu können, sind hierzu ein oder mehrere Testdatenbanken notwendig. Um beispielsweise die einzelnen Ausführungspfade des zuvor gezeigten Anwendungsfalles zu überprüfen, ist eine Testdatenbank notwendig, die Bücher mit unterschiedlichen Eigenschaften enthält (z.B. alle Bücher mit einer bestimmten ISBN sind bereits verliehen bzw. kein Buch mit einer bestimmten ISBN steht im Freihandbereich).

Um eine Testdatenbank für ein oder mehrere Testfälle einer OLTP-Anwendung zu spezifizieren, sind im Unterschied zu RQP häufig mehr als eine SQL-Anfrage und ein Ergebnis notwendig. Der Grund hierfür ist, dass OLTP-Anwendungen in der Regel verschiedene Datensätze einer Datenbank lesen bzw. ändern, die nicht notwendigerweise korreliert sind und sich dadurch nur umständlich in einer Sicht (SQL-Anfrage und Ergebnis) beschreiben lassen. Daher ist die Grundidee von Multi Reverse Query Processing (MRQP) die Testdatenbank mit Hilfe von mehreren Sichten zu spezifizieren. Die Definition der einzelnen Sichten kann entweder manuell erfolgen, indem ein Tester SQL als deklarative Sprache zur Spezifikation der Testdatenbank verwendet, oder sie kann automatisch mit Hilfe von Codeanalysen erfolgen.

Ein Beispiel der Spezifikation einer Testdatenbank für einen Testfall, der im oben gezeigten Anwendungsfall die erfolgreiche Buchreservierung überprüfen soll (d.h. keine Ausnahme tritt auf), ist in Abbildung 4a) zu sehen. Um einen solchen Testfall ausführen zu können, muss die Testdatenbank mindestens ein Buch mit einer bestimmten ISBN enthalten, das im Freihandbereich steht und das nicht ausgeliehen ist. Darüber hinaus darf der Benutzer, der das Buch ausleihen möchte, keine Mahngebühren haben. Eine entsprechende Testdatenbank wird im Beispiel mit Hilfe von zwei Sichten ($Q1/R1$ und $Q2/R2$) beschrieben, die die für den Testfall relevanten Daten definieren (z.B. die Werte für die Attribute b_isbn und $b\_closedstack$ der Tabelle $book$, aber nicht für irrelevante Attribute wie b_title). Das Datenbankschema der Anwendung ist in Abbildung 4b) zu sehen und eine mögliche von MRQP erzeugte Testdatenbank in Abbildung 4c).

RQP unterstützt (wie schon erwähnt) als Eingabe nicht mehrere SQL-Anfragen und deren Ergebnisse. MRQP ist wie in [Bin08] beschrieben allerdings nicht für alle möglichen SQL-Anfragen effizient lösbar. Das Ziel von MRQP ist es daher RQP als Basis zur Generierung einer Testdatenbank zu verwenden. Um dies zu ermöglichen wird in [BKL08] die Menge der unterstützten SQL-Anfragen entsprechend eingeschränkt. Mit dieser eingeschränkten Menge von SQL-Anfragen lassen sich dennoch alle möglichen Testdatenbanken deklarativ beschreiben.

```
Q1: SELECT b_closedstack,
    b_uid
    FROM book
    WHERE b_isbn= '12345'
R1: {<false, NULL>}
```

```
Q2: SELECT u_password,
    u_charges
    FROM user
    WHERE u_name='test'
R2: {<test, 0.0>}
```

(a) Spezifikation der Testdatenbank D

```
CREATE TABLE user (
u_id INTEGER PRIMARY KEY,
u_name VARCHAR(20) UNIQUE,
u_password VARCHAR(20),
u_charges FLOAT NOT NULL
  CHECK(u_charges>=0));
```

```
CREATE TABLE book (
b_id INTEGER PRIMARY KEY,
b_title VARCHAR (20) NOT NULL,
b_closedstack BOOLEAN NOT NULL,
b_isbn VARCHAR(20) UNIQUE,
b_uid INTEGER FOREIGN KEY
 REFERENCES user(u_id));
```

(b) Datenbankschema S

u_id	u_name	u_password	u_charges
1	test	test	0.0

user

b_id	b_title	b_closedstack	b_isbn	b_uid
1	TitleA	false	12345	NULL

book

(c) Instanz der Testdatenbank D

Abbildung 4: MRQP Beispiel für OLTP-Anwendungen

4 Symbolic Query Processing

Symbolic Query Processing (SQP) [BKLO07] ist eine Kombination aus der traditionellen Anfragebearbeitung in einem DBMS auf der einen Seite und der symbolischen Ausführung [Kin76] auf der anderen Seite. Bei SQP werden die Daten in einer Datenbank durch Symbole anstelle von konkreten Werten repräsentiert und die Anfrageausführung verändert die symbolischen Daten anstelle der konkreten Werte. Die Hauptanwendung von SQP ist die Erzeugung von Testdatenbanken für den Test einzelner Komponenten eines DBMS.

Soll bei einem DBMS eine neue Komponente oder eine neue Technik in das Produkt integriert werden, dann ist es notwendig, diese sehr sorgfältig zu testen, um zu überprüfen, dass sich das DBMS unter der erwarteten Last wie gewünscht verhält. Wenn beispielsweise ein neuer Algorithmus für einen Verbund-Operator in ein DBMS integriert wird, ist es notwendig, sein Verhalten (z.B. Speicheraufwand, Performanz) zu uberprüfen. Auf Basis bestimmten Anfrage, die den gewünschten Operator nutzt, ist es dann sinnvoll verschiedene Testfälle abzuleiten, die sich in den Eigenschaften der Zwischenergebnisse unterscheiden (z.B. Kardinalität, Verteilung).

Für den Test einzelner DBMS-Komponenten macht es Sinn, einen Testfall durch einen Anfrageplan Q und einen Satz von Bedingungen C zu definieren, welche die Eigenschaften der Zwischenergebnisse (z.B. Kardinalität, Verteilung) beschreiben. Abbildung 5b) zeigt einen Testfall T_1, der auf der Anfrage Q aus Abbildung 5a) basiert. Der Testfall T_1 definiert die Eigenschaften der Zwischenergebnisse, wenn die Anfrage Q auf einer Testdatenbank (mit zwei Tabellen R und S) ausgeführt wird (wobei R 2000 Datensätze enthalten soll und S 4000 Datensätze): Der Testfall definiert z.B., dass das Zwischenergebnis der Selektion $\sigma_{R.a<:p_1}$ (a ist eine Spalte der Tabelle R und $:p_1$ ist ein Parameter der Anfrage) dabei zehn Datensätze und das Endergebnis des Verbundes 40 Datensätze enthalten soll.

Testfall T_1 dient z.B. dazu, das Verhalten des neuen Verbund-Algorithmus für den Fall zu überprüfen, dass die beiden Eingabetabellen sich in ihrer Größe unterscheiden und das Endergebnis des Verbundes gegenüber der Eingabe recht klein ist. Ein weiteres Beispiel ist

Testfall T_2 in Abbildung 5c). Dieser Testfall überprüft das Verhalten des neuen Verbund-Algorithmus, wenn beide Eingaben wie auch das Endergebnis recht groß sind.

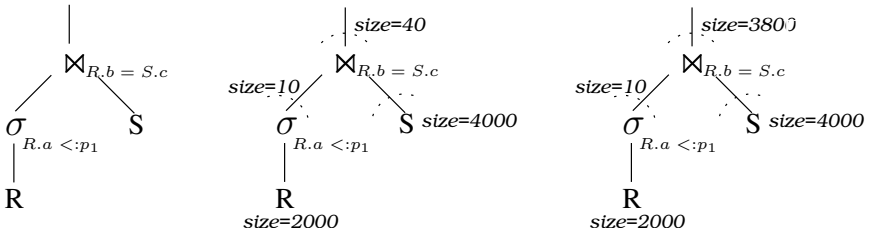
Das Erstellen einer geeigneten Testdatenbank für einen solchen Testfall ist in der Praxis häufig ein manueller Prozess und sehr zeitintensiv, da aktuell keine geeigneten Werkzeuge existieren, die zur Erzeugung einer Testdatenbank die Bedingungen auf den Zwischenergebnisse beachten. Um beispielweise eine Testdatenbank für Testfall T_1 aus Abbildung 5b) zu erzeugen, müsste ein Tester zuerst mit Hilfe eines existierenden Datenbankgenerators (z.B. mit Hilfe des IBM DB2 Test Database Generator, [CDF+04], [BC05] oder [HTW06]) eine Testdatenbank D erzeugen und danach *manuell* die Tabellen R und S so anpassen, dass die Ausführung der Anfrage Q die gewünschten Zwischenergebnisse liefert (z.B. 10 Datensätze für die Selektion $\sigma_{R.a<:p_1}$).

SQP kann dazu verwendet werden, diesen Prozess zu automatisieren. Auf der Basis von SQP wurde ein entsprechender Prototyp QAGen entwickelt [BKLO07] (QAGen steht für "Query-Aware Test Database Generator"). QAGen nimmt das Datenbankschema S und einen Testfall T (Anfrage und Bedingungen) als Eingabe und generiert eine Datenbank D, welche die definierten Bedingungen C erfüllt, wenn die Anfrage Q auf D ausgeführt wird.

Um eine Testdatenbank für den Testfall T_1 in Abbildung5b) zu generieren, erzeugt QAGen zuerst die beiden Tabellen R und S (wobei die Tabelle R aus 2000 *symbolischen Datensätzen* besteht und Tabelle S aus 4000 *symbolischen Datensätzen*). Ein symbolischer Datensatz ist ein Tupel, das nicht aus konkreten Werten, sondern aus einzelnen Symbolen besteht (siehe [BKLO07] für weitere Details). Danach wird die Anfrage Q mit Hilfe des *symbolischen Anfrageprozessors* in QAGen verarbeitet. Der symbolische Anfrageprozessor funktioniert ähnlich wie ein Anfrageprozessor eines DBMS, d.h. jeder Operator ist als Iterator implementiert, wobei der Datenfluss von den Blättern des Anfrageplans hin zur Wurzel geht [Gra93]. Während der Anfrageprozessierung manipulieren die einzelnen Operatoren die symbolischen Daten dahingehend, dass die Bedingungen der einzelnen Operatoren entsprechend der Semantik des jeweiligen Operators erfüllt sind (z.B. ein Selektionsoperator fügt den symbolischen Datensätzen, welche den Operator passieren, das Selektionsprädikat hinzu und den anderen Datensätzen das negierte Selektionsprädikat).

In Abbildung 5b) fügt der Selektionsoperator beispielweise die Bedingung $R.a <: p_1$ den ersten zehn Datensätzen der Eingabe hinzu (die dann auch im Zwischenergebnis des Operators enthalten sind). Den restlichen Datensätzen wird die Bedingung $R.a \geq: p_1$ hinzugefügt. Nachdem der *symbolische Anfrageprozessor* den Anfrageplan verarbeitet hat, sind alle Bedingungen des Testfalls in den symbolischen Daten enthalten. Als letzter Schritt kann QAGen dann einen Modellprüfer verwenden, um aus den symbolischen Daten mit den entsprechenden Bedingungen eine konkrete Testdatenbank zu erzeugen, die diese Bedingungen erfüllt.

In [BKLO07] wurde gezeigt, dass QAGen mit Hilfe von SQP Testdatenbanken für eine Reihe komplexer Testfälle erzeugen kann. In den Experimenten wurden für die meisten Anfragen des TPC-H Testfälle definiert und entsprechende Testdatenbanken erzeugt. Die Bandbreite der Testdatenbankgenerierung mit QAGen auf einem Linux AMD Opteron 2.2 GHz Servers mit 4 GB Hauptspeicher variierte von 1.5GB pro Stunde im besten Fall bis 50GB pro Stunde im schlechtesten Fall.



(a) Anfrage Q (b) Testfall T_1 (c) Testfall T_2

Abbildung 5: SQP Beispiel für den Test von DBMS Komponenten

5 Zusammenfassung

Die Automation einzelner Testaktivitäten (Testfallerzeugung, Generierung der Testdatenbank, Testfallausführung) ist ein wirkungsvoller Ansatz, um auf der einen Seite die Testkosten zu reduzieren und auf der anderen Seite die Testabdeckung zu erhöhen. Allerdings sind existierende Testwerkzeuge nicht unbedingt (optimal) für den Test von Datenbankanwendungen bzw. Datenbankmanagementsystemen geeignet. Speziell für die Generierung von Testdatenbanken existieren meist Werkzeuge, die Zufallsdaten erzeugen, welche sich nur bedingt für die Ausführung bestimmter Testfälle eignen.

In diesem Artikel haben wir verschiedene neue Ansätze beschrieben, die zur Generierung von relevanten Testdatenbanken für den funktionalen Test von OLTP- und OLAP-Anwendungen bzw. auch für das Testen einzelner Komponenten eines Datenbankmanagementsystems geeignet sind. Die Idee der Ansa”tze ist es, Bedinguengen an die Testdatenbank deklarativ zu beschreiben, sodass bestimmte Testfälle ausführbar sind.

Jedoch haben diese Ansätze bestimmte Einschränkungen und Nachteile, auf die wir in Zukunft gerne eingehen möchten:

- Um die Benutzbarkeit der Ansätze zu steigern, wäre eine vereinfachte (grafische) Spezifikation der Testdatenbank sinnvoll, die den Tester nicht zwingt, SQL-Anfragen manuell zu schreiben. Diese Werkzeuge könnten in ihrer Funktionsweise an Werkzeugen zur Definition von Sichten in einer Datenbank angelehnt sein.
- Ein weiterer Nachteil der Ansätze ist, dass häufig zu viele Testdatenbanken (z.B. eine Datenbank pro Testfall) generiert werden. Sinnvoll wäre es, einzelne Testdatenbanken miteinander zu verschmelzen. Dies ist bei allen gezeigten Ansätzen möglich, da die einzelnen Testdatenbanken deklarativ beschrieben werden und daher die Minimierung der generierten Datenbanken ein mögliches Optimierungsziel ist.
- Ein weiterer interessanter und wichtiger Punkt ist die Anpassung einer Testdatenbank, wenn sich die zugrunde liegende Spezifikation leicht ändert. Hierbei möchte man das erneute Generieren der kompletten Testdatenbank vermeiden und nur möglichst einen geringen Teil der Daten anpassen.

Literatur

[BC05] Nicolas Bruno und Surajit Chaudhuri. Flexible Database Generators. In *VLDB*, Seiten 1097–1107, 2005.

[Bin99] Robert V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Bin08] Carsten Binnig. *Generating Meaningful Test Databases*. Dissertation, University of Heidelberg, 2008.

[BKL07] Carsten Binnig, Donald Kossmann und Eric Lo. Reverse Query Processing. In *ICDE*, Seiten 506–515, 2007.

[BKL08] Carsten Binnig, Donald Kossmann und Eric Lo. Multi-RQP: generating test databases for the functional testing of OLTP applications. In *DBTest*, Seiten 5–10, 2008.

[BKLO07] Carsten Binnig, Donald Kossmann, Eric Lo und Tamer Özsu. QAGen: Generating Query-Aware Test Databases. In *SIGMOD*, Seiten 341–352, 2007.

[BKLSB08] Carsten Binnig, Donald Kossmann, Eric Lo und Angel Saenz-Badillos. Automatic Result Verification for the Functional Testing of a Query Language. In *ICDE*, 2008.

[CDF+04] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos und Elaine J. Weyuker. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability*, 2004.

[GMUW01] Hector Garcia-Molina, Jeffrey D. Ullman und Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, 2001.

[Gra93] Goetz Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[HKL07] Florian Haftmann, Donald Kossmann und Eric Lo. A framework for efficient regression tests on database applications. *VLDB J.*, 16(1):145–164, 2007.

[HTW06] Kenneth Houkjær, Kristian Torp und Rico Wind. Simple and Realistic Data Generation. In *VLDB*, Seiten 1243–1246, 2006.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[RTI02] RTI. The Economic Impacts of Inadequate Infrastructure for Software Testing. May 2002.

[TPC] TPC-H Benchmark. http://www.tpc.org/tpch.

Carsten Binnig studierte von 1998-2001 Informatik an der Berufsakademie Karlsruhe. Danach arbeitete er von 2001-2004 am Forschungszentrum Karlsruhe an innovativen Lösungen für Wissensmanagementsysteme. Während dieser Zeit legte Herr Binnig noch einen Master-Abschluss in Informatik an der Fachhochschule Karlsruhe ab. Anschließend (2004-2007) promovierte er am Lehrstuhl für Datenbanken und am Lehrstuhl für Software-Engineering an der Universität Heidelberg, wo er sich mit neuen Ideen zur Generierung von Testdatenbanken befasste. Diese Anätze integrierte Herr Binnig während eines Aufenthalts bei Microsoft (Redmond, USA) in die SQL-Server Testlandschaft. Aktuell arbeitet Herr Binnig als Senior Researcher an der ETH Zürich (Systems Group) an effizienten Algorithmen zur Analyse großer Datenbestände u.a. zusammen mit der SAP AG (Walldorf, Deutschland).