

Vergleich von SQL-Anfragen: Theorie und Implementierung in Java

Robert Hartmann

Martin-Luther-Universität Halle-Wittenberg
Naturwissenschaftliche Fakultät III
Institut für Informatik

26. September 2013

- 1 Einleitung
 - Motivation
 - Überblick des neuen Ansatzes
- 2 Standardisierung
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen
- 5 Implementierung
- 6 Live Präsentation

- 1 Einleitung
 - Motivation
 - Überblick des neuen Ansatzes
- 2 Standardisierung
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen
- 5 Implementierung
- 6 Live Präsentation

Motivation

- Vergleich von SQL-Anfragen üblicherweise in der Lehre
- Übungsaufgaben notwendig für praktisches Verständnis von SQL
- Bestehend aus Sachaufgabe und Datenbankschema
- Lösung des Lernenden formuliert als SQL-Statement

Motivation

- Vergleich von SQL-Anfragen üblicherweise in der Lehre
- Übungsaufgaben notwendig für praktisches Verständnis von SQL
- Bestehend aus Sachaufgabe und Datenbankschema
- Lösung des Lernenden formuliert als SQL-Statement

semantische Äquivalenz

Zwei SQL-Anfragen sind semantisch äquivalent, wenn beide Anfragen auf allen Datenbankzuständen einer Datenbank stets die selben Tupel zurückliefern.

- Allgemein: Nicht entscheidbar
- Korrektur der Aufgabe auf zwei Arten: manuell oder automatisch

Manueller Vergleich

Vorteile

- Korrektur zuverlässig
- Syntaktische Varianten der Lösung werden erkannt
- Lernender erhält (potentiell) detailliertes Feedback

Nachteile

- Korrektur langsam
- Wenig Geld und Kürzungen in Lehre → Wenig Zeit zur Verfügung
- Unter Zeitdruck: Mehr Fehler, wenig detailliertes Feedback

Fazit: Manuelles Vergleichen funktioniert nur, wenn genug Mitarbeiter/ Hilfskräfte zur Verfügung stehen.

Automatischer Vergleich

Automatischer Vergleich zweier SQL-Anfragen wird üblicherweise durch den Vergleich der Ergebnistupel realisiert.

Vorteile

- Korrektur in Echtzeit
- Keine Mitarbeiter oder Hilfskräfte benötigt

Nachteile

- Für jedes Datenbankschema sind Daten notwendig
- Feedback für Lernenden oft unzureichend um Fehler zu identifizieren
- *false positive* leicht zu erstellen, System kann ausgehebelt werden
- *false positive* auch unabsichtlich problematisch: Lernender bemerkt Fehler nicht

Fazit: Automatisches Vergleichen durch bloßes Vergleichen von Ergebnistupeln nicht zuverlässig

Neuer Ansatz

- automatischer Vergleich mit Datenbankschema und Musterlösung
- sichere Rückmeldung benötigt Daten, Hauptteil aber ohne möglich

Neuer Ansatz

- automatischer Vergleich mit Datenbankschema und Musterlösung
- sichere Rückmeldung benötigt Daten, Hauptteil aber ohne möglich

Erster Schritt: **Standardisierung** (Prüfung: hinreichende Bedingung)

Ziel: Semantisch äquivalente Anfragen sollen nach der Standardisierung auch syntaktisch gleich sein

Neuer Ansatz

- automatischer Vergleich mit Datenbankschema und Musterlösung
- sichere Rückmeldung benötigt Daten, Hauptteil aber ohne möglich

Erster Schritt: **Standardisierung** (Prüfung: hinreichende Bedingung)

Ziel: Semantisch äquivalente Anfragen sollen nach der Standardisierung auch syntaktisch gleich sein

Zweiter Schritt: **Vergleich von Ergebnismengen** (Prüfung: notwendige Bedingung)

Ziel: Nachweis der Ungleichheit beider Anfragen

Neuer Ansatz

- automatischer Vergleich mit Datenbankschema und Musterlösung
- sichere Rückmeldung benötigt Daten, Hauptteil aber ohne möglich

Erster Schritt: **Standardisierung** (Prüfung: hinreichende Bedingung)

Ziel: Semantisch äquivalente Anfragen sollen nach der Standardisierung auch syntaktisch gleich sein

Zweiter Schritt: **Vergleich von Ergebnismengen** (Prüfung: notwendige Bedingung)

Ziel: Nachweis der Ungleichheit beider Anfragen

Dritter Schritt: **Struktureller Vergleich der Anfragen**

Ziel: genauere Lokalisierung von Fehlern des Lernenden

- 1 Einleitung
- 2 Standardisierung**
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen
- 5 Implementierung
- 6 Live Präsentation

Überblick

- Standardisierung bildet ersten Schritt
- Entfernen von syntaktischen Details und Einlesen von Anfrage in Datenstruktur durch Parser
- Behandeln von einzelnen Teilen der SQL-Anfrage
(SELECT, FROM, WHERE, GROUP BY, ORDER BY)
- Abarbeitung nicht streng hintereinander, da einige Teile abhängig sind
- Zusammensetzen der behandelten Teile zur standardisierten SQL-Anfrage

FROM-Teil

- Lexikographisches Sortieren der Tabellen
- Einführung künstlicher Tupelvariablen (TV) mit fortlaufender Nummerierung (a_1, a_2, \dots)
- TV auf gleicher Ebene erhalten gleichen Startwert für Iteration

FROM-Teil

- Lexikographisches Sortieren der Tabellen
- Einführung künstlicher Tupelvariablen (TV) mit fortlaufender Nummerierung (a1,a2,...)
- TV auf gleicher Ebene erhalten gleichen Startwert für Iteration

```
SELECT ename FROM emp
WHERE sal > (SELECT AVG(sal) FROM emp)
AND empno > (SELECT AVG(empno) FROM emp)
```

FROM-Teil

- Lexikographisches Sortieren der Tabellen
- Einführung künstlicher Tupelvariablen (TV) mit fortlaufender Nummerierung (a1,a2,...)
- TV auf gleicher Ebene erhalten gleichen Startwert für Iteration

```
SELECT ename FROM emp
WHERE sal > (SELECT AVG(sal) FROM emp)
AND empno > (SELECT AVG(empno) FROM emp)
```

```
SELECT ename FROM emp a1
WHERE sal > (SELECT AVG(sal) FROM emp a2)
AND empno > (SELECT AVG(empno) FROM emp a2)
```


SELECT-Teil

- Ersetzen von Wildcard (*) zu konkreten Spalten (vor Sortieren in `FROM`)
- Einführen der künstlichen TV als Aliase
- Wenn Spaltenreihenfolge unwichtig: lexikographisches Sortieren der Spalten

SELECT-Teil

- Ersetzen von Wildcard (*) zu konkreten Spalten (vor Sortieren in FROM)
- Einführen der künstlichen TV als Aliase
- Wenn Spaltenreihenfolge unwichtig: lexikographisches Sortieren der Spalten

```
SELECT ename FROM emp a1
WHERE sal > (SELECT AVG(sal) FROM emp a2)
AND empno > (SELECT AVG(empno) FROM emp a2)
```

SELECT-Teil

- Ersetzen von Wildcard (*) zu konkreten Spalten (vor Sortieren in FROM)
- Einführen der künstlichen TV als Aliase
- Wenn Spaltenreihenfolge unwichtig: lexikographisches Sortieren der Spalten

```
SELECT ename FROM emp a1
WHERE sal > (SELECT AVG(sal) FROM emp a2)
AND empno > (SELECT AVG(empno) FROM emp a2)
```

```
SELECT a1.ename FROM emp a1
WHERE a1.sal > (SELECT AVG(a2.sal) FROM emp a2)
AND a1.empno > (SELECT AVG(a2.empno) FROM emp a2)
```

WHERE-Teil - syntaktische Varianten

- Umwandeln des WHERE-Ausdrucks in KNF (konjunktive Normalform)
- Erhöhung der Lesbarkeit, Eliminierung von unnötig tiefen Teilbäumen

Entfernung von syntaktischen Varianten:

WHERE-Teil - syntaktische Varianten

- Umwandeln des WHERE-Ausdrucks in KNF (konjunktive Normalform)
- Erhöhung der Lesbarkeit, Eliminierung von unnötig tiefen Teilbäumen

Entfernung von syntaktischen Varianten:

- `a BETWEEN l AND u` zu `a >= l AND a <= u`
- `a >= ALL(c1, c2, c3)` zu
`a >= c1 AND a >= c2 AND a >= c3`
- `a >= ANY(c1, c2, c3)` zu
`a >= c1 OR a >= c2 OR a >= c3`
- `EXISTS (SELECT expr FROM ...)` zu
`EXISTS (SELECT 1 FROM ...)`

WHERE-Teil - Arten von Operatoren

- WHERE-Ausdruck besteht im wesentlichen aus Operatoren, Konstanten und Spaltennamen
- Ordnung wird benötigt, damit semantisch äquivalente Anfragen auch syntaktisch gleich sind

WHERE-Teil - Arten von Operatoren

- WHERE-Ausdruck besteht im wesentlichen aus Operatoren, Konstanten und Spaltennamen
- Ordnung wird benötigt, damit semantisch äquivalente Anfragen auch syntaktisch gleich sind

Arten von Operatoren:

- Operatoren:
IS NULL, IS NOT NULL, EXISTS, ANY/SOME, ALL, ...
→ keine syntaktischen Varianten möglich
- nicht-kommutative Operatoren: \leq , \geq , $<$, $>$, $-$, $/$
→ Hinzufügen aller Schreibweisen
- kommutative Operatoren: $=$, $<>$, $+$, $*$, *AND*, *OR*
→ Alle Permutationen der Operanden sind gültig → Festlegung einer Ordnung

Hinzufügen aller Schreibweisen

- Mengen M_i enthalten Muster, die äquivalente Schreibweisen beinhalten
- Passt atomare Formel a auf $m \in M_i \rightarrow$ Hinzufügen aller $m' \in M_i$ mit $m \neq m'$
- Vorgang wiederholen bis keine neuen Formeln hinzugefügt wurden
- Beispiel: `salary >= minsal`
- $M_6 : \{ A \geq B, B \leq A \}$
- $\{ \}$

Hinzufügen aller Schreibweisen

- Mengen M_i enthalten Muster, die äquivalente Schreibweisen beinhalten
- Passt atomare Formel a auf $m \in M_i \rightarrow$ Hinzufügen aller $m' \in M_i$ mit $m \neq m'$
- Vorgang wiederholen bis keine neuen Formeln hinzugefügt wurden
- Beispiel: `salary >= minsal`
- $M_6 : \{ A \geq B, B \leq A \}$
- $\{ salary \geq minsal, \}$

Hinzufügen aller Schreibweisen

- Mengen M_i enthalten Muster, die äquivalente Schreibweisen beinhalten
- Passt atomare Formel a auf $m \in M_i \rightarrow$ Hinzufügen aller $m' \in M_i$ mit $m \neq m'$
- Vorgang wiederholen bis keine neuen Formeln hinzugefügt wurden
- Beispiel: `salary >= minsal`
- $M_6 : \{ A \geq B, B \leq A \}$
- $\{ salary \geq minsal, minsal \leq salary, \}$

Ordnung bei kommutativen Op

- Baum $T(x)$ mit Wurzel x . $children(x) = (c_1, c_2, \dots, c_n)$
 - Reihenfolge von $children(x)$ beliebig veränderbar
- Eindeutige Repräsentation erfordert Ordnung von $children(x)$

Ordnung bei kommutativen Op

- Baum $T(x)$ mit Wurzel x . $children(x) = (c_1, c_2, \dots, c_n)$
- Reihenfolge von $children(x)$ beliebig veränderbar
- Eindeutige Repräsentation erfordert Ordnung von $children(x)$
 - $order : children \rightarrow \mathbb{N}$
 - Nach Sortieren gilt: $order(c_1) \leq order(c_2) \leq \dots order(c_n)$

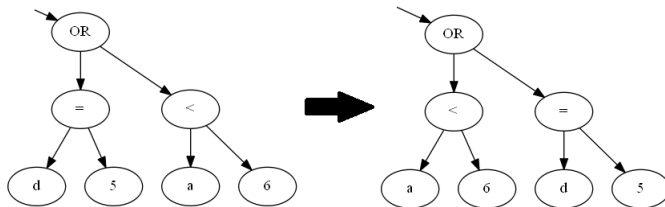
r	Spalte	Konstante	OR	\leq	\geq	$<$	$>$	$=$	$<>$	$+$...
$order(r)$	1	2	3	4	5	6	7	8	9	10	11

- Vorgehen: BOTTOM-UP durch Rekursion

WHERE-Teil - Sortierung (2)

Beispiel:

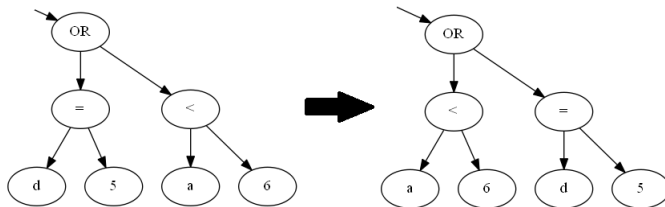
$d = 5 \text{ OR } a < 6 \rightarrow a < 6 \text{ OR } d = 5$



WHERE-Teil - Sortierung (2)

Beispiel:

$d = 5 \text{ OR } a < 6 \rightarrow a < 6 \text{ OR } d = 5$

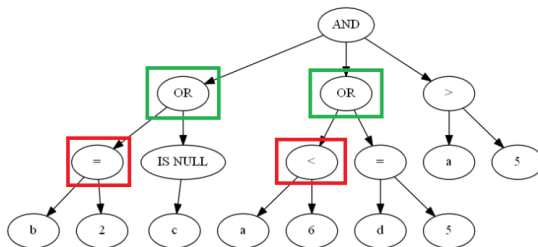


Problem: Was passiert, wenn zwei Kinder den gleichen Operator bezeichnen oder beide Kinder Spaltennamen bzw. Konstanten sind?

Beide Kindknoten bezeichnen gleichen Operator

Beispiel:

$b = 2$ OR c IS NULL and $a < 6$ OR $d = 5$ AND $a > 5$



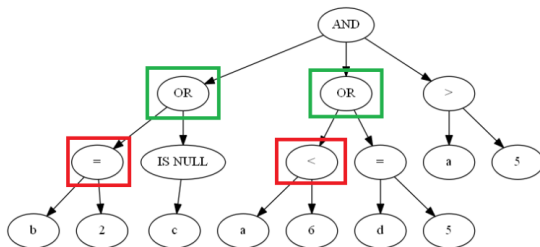
→ Tauschen, da $order(=) = 8 > 6 = order(<)$

Beide Kindknoten bezeichnen gleichen Operator

- Situation: $order(c_i) = order(c_j)$ und c_i, c_j sind Operatoren ($j = i + 1$)
- Schrittweise Tiefensuche auf $T(c_i), T(c_j)$ Aktuelle Knoten in DFS: v_i, v_j
- Wenn $v_i \neq v_j$ dann muss gelten: $order(v_i) < order(v_j)$, sonst Tauschen von c_i, c_j

Beispiel:

$b = 2$ OR c IS NULL and $a < 6$ OR $d = 5$ AND $a > 5$



→ Tauschen, da $order(=) = 8 > 6 = order(<)$

GROUP BY und ORDER BY

GROUP BY:

- Einführen der künstlichen TV
- Lexik. Sortierung der GROUP BY-Items
- HAVING wie WHERE behandelt

ORDER BY:

- Ersetzen von Selektionsnummern
- Einführen der künstlichen TV

Verbunde und Unterabfragen

- Äußere Verbunde ersetzt durch inneren Verbund verknüpft mit `UNION ALL`
- Natürliche Verbunde unter `FROM` als innere Verbunde unter `WHERE`
- Entfernen von Schlüsselwort `CROSS JOIN`
- Innere Verbunde unter `FROM` werden unter `WHERE` formuliert
- Umwandeln sämtlicher Unterabfragen zu `EXISTS`-Unterabfragen

- 1 Einleitung
- 2 Standardisierung
- 3 Vergleich mit realen Daten**
- 4 Hinweismeldungen
- 5 Implementierung
- 6 Live Präsentation

Idee

- iterativer Vergleich beider Ergebnismengen nicht ohne weiteres möglich
- Problem: Optimierer des DBMS übernimmt Sortierung, wenn nicht explizit angegeben
- selbst mit expliziter Angabe: Sortierung nicht notwendigerweise eindeutig

- Beispiel:

```
SELECT surname, firstname  
FROM people ORDER BY surname
```

Probleme:

Idee

- iterativer Vergleich beider Ergebnismengen nicht ohne weiteres möglich
- Problem: Optimierer des DBMS übernimmt Sortierung, wenn nicht explizit angegeben
- selbst mit expliziter Angabe: Sortierung nicht notwendigerweise eindeutig
- Beispiel:

```
SELECT surname, firstname  
FROM people ORDER BY surname
```

Probleme:

- Falsche Lösung wird zufällig korrekt sortiert → falsche Übereinstimmung
- Korrekte Lösung wird falsch sortiert → keine Übereinstimmung

Fallunterscheidung (1)

- ML enthält kein ORDER BY → ORDER BY von LL streichen
- künstliche Sortierung nach Auswahlspalten, die unbenutzt sind
- iteratives Vorgehen

Beispiel:

```
SELECT firstname, surname FROM people  
ORDER BY surname
```

Fallunterscheidung (1)

- ML enthält kein ORDER BY → ORDER BY von LL streichen
- künstliche Sortierung nach Auswahlspalten, die unbenutzt sind
- iteratives Vorgehen

Beispiel:

```
SELECT firstname, surname FROM people  
ORDER BY surname
```

zu:

```
SELECT firstname, surname FROM people  
ORDER BY surname, 1, 3
```

→ SQL-Anfrage nun auf jedem DBMS immer eindeutige Ausgabereihenfolge

- 1 Einleitung
- 2 Standardisierung
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen**
- 5 Implementierung
- 6 Live Präsentation

Feedback

- Vergleich einzelner Teile der Anfrage (`SELECT`, `FROM`, ...)
- Hinweis an Nutzer, welche Teile identisch mit ML sind (WO ist der Fehler?)
- Vergleich von Anzahl verschiedener Komponenten (Tabellen, Formeln, Verbunde, Unterabfragen)
- konkreter Hinweis, WAS an der eigenen Lösung fehlt/überflüssig ist
- Vergleich mit Realdaten (Schritt 2) unterstützt verschiedene DBMS → verschiedene Fehlermeldung der DBMS beim Parsen mit DBMS-Parser
- Kompatibilitätstest

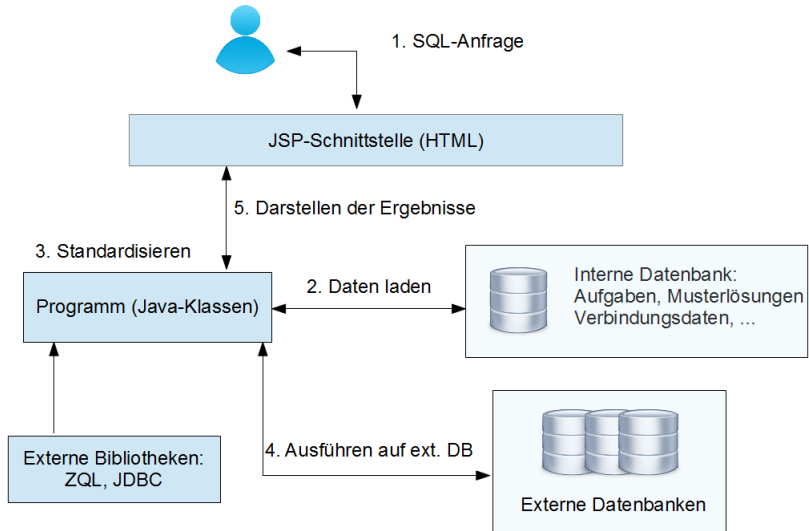
- 1 Einleitung
- 2 Standardisierung
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen
- 5 Implementierung**
- 6 Live Präsentation

Technische Details

Ergebnisse der Arbeit sollen in Form einer Lernplattform umgesetzt werden.

- Javaklassen per JSP als HTML-Seiten nutzbar
- plattformunabhängig
- automatisches Build- und Deployskript per *ant*
- Parser: ZQL (Open Source)
- mögliche DBMS für Schritt 2: alle mit JDBC-Connector

Aufbau des Programms



Einschränkungen in der Praxis / Probleme

- Nur Tabellen in `FROM` zugelassen (keine Verbunde, Unterabfragen)
 - keine Implementierung von (bereits ausgearbeiteten) Verbundskonzepten
- Parser versteht keine `CREATE TABLE`-Anweisungen
 - Kenntnis über Spalten (Name, Datentyp, Eigenschaften) notwendig
 - rudimentäres Parsen von `CREATE TABLE`-Anweisungen implementiert
- JDBC-Connector für DBMS zum Teil sehr unterschiedlich
 - Anbinden von neuen DBMS-Typen muss getestet werden

Fortsetzung

- Erweitern des Parsers
- Umsetzen der Verbundkonzepte
- Erkennen von unnötigem `DISTINCT`
- Ausweitung auf Behandlung aller SQL-Ausdrücke (`UPDATE`, `DELETE`)
- Ausarbeitung und Implementierung weiterer Konzepte (zB: unnötige Verbunde, ...)
- transitiv-implizierte Formeln: $a = b \text{ AND } a = 5 \rightarrow b = 5$
- Beschränkung der Domänen: $a > 2 \text{ AND } a > 0 \rightarrow a > 2$

- 1 Einleitung
- 2 Standardisierung
- 3 Vergleich mit realen Daten
- 4 Hinweismeldungen
- 5 Implementierung
- 6 Live Präsentation**