

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

- 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
- Allgemein: Nicht entscheidbar
- Korrektur der Aufgabe auf zwei Arten: manuell oder automatisch
- Beide Ansätze haben signifikante Nachteile

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 (1)

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

Erster Schritt: **Standardisierung**

- Entfernung von syntaktischen Details
- Vereinheitlichen von syntaktischen Varianten
- Sortieren des SQL-Ausdrucks

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

Neuer Ansatz (2)

Sind beide Anfragen nach der Standardisierung nicht syntaktisch gleich, so wird versucht die Ungleichheit beider Anfragen zu zeigen.

Zweiter Schritt: **Vergleich von Ergebnismengen**

- Wird benutzt um **Ungleichheit** zu zeigen, Unterschied zum üblichen Vorgehen
- *false positive* dadurch nicht möglich
- Versagt Schritt zwei so ist der Status der Lösung: unbekannt (Diese sollten zum großen Teil korrekt sein)

Gründe für *unbekannte* Lösungen:

- Programmfehler
- unnötige Bedingungen in WHERE-Klausel
- Verwendung von syntaktisch unterschiedlichen Konzepten

Neuer Ansatz (3)

Aus besagten Schwächen leiten sich weitere Anforderungen ab:

- Ausdrücke auswerten
- Standardisierungskonzept für Verbunde
- Standardisierungskonzept für Unterabfragen
- Anzahl von *unbekannten* Lösungen minimieren durch Zulassen mehrerer Musterlösungen

Dritter Schritt: **Struktureller Vergleich der Anfragen**

- Vergleich der einzelnen Teile der SQL-Anfragen → genaue Lokalisierung des Fehlers
- Vergleich der Anzahl verschiedener Komponenten (Tabellen, Formeln, Verbunde, Unterabfragen)

- 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
- Sortierung: (1) Tabellen, (2) Unterabfragen, (3) Verbunde
- 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 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 - Überblick

- Umwandeln des `WHERE`-Ausdrucks in KNF (konjunktive Normalform)
- Erhöhung der Lesbarkeit, Eliminierung von unnötig tiefen Teilbäumen
 - (Entfernen von unnötigen Klammern)
 - Auflösen von `NOT`
 - Anwendung des Distributivgesetzes

WHERE-Teil - syntaktische Varianten

Entfernung von syntaktischen Varianten:

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

WHERE-Teil - Operatorenvielfalt (1)

Problem: Ausdrücke können unterschiedlich aufgeschrieben werden. Es ist nicht klar, welche Schreibweise der Lernende verwenden wird.

Beispiel: $a > 3$ äquivalent zu $3 < a$ und mit Zusatzwissen auch zu $a \geq 2$ und $2 \leq a$

Ansatz 1: Hinzufügen aller äquivalenten Ausdrücke

Ansatz 2: Zulassen einer Repräsentation und Verbiehen der restlichen Schreibweisen

Wir verfolgen den implementierten Ansatz 1.

WHERE-Teil - Operatorenvielfalt (2)

- 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 >= 5` mit `salary NUMERIC(4,2)`
- A, B : Konstanten oder Attribute, X : numerische Konstante
- $M_6 : \{ A \geq B, B \leq A \}$
 $M_7 : \{ A > X, A \geq X + adjust(A) \}$
- $adjust(A) = 1 / (10^{places(A)})$
- $\{ salary \geq 5, \quad \}$

WHERE-Teil - Operatorenvielfalt (2)

- 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 >= 5` mit `salary NUMERIC(4,2)`
- A, B : Konstanten oder Attribute, X : numerische Konstante
- $M_6 : \{ A \geq B, B \leq A \}$
 $M_7 : \{ A > X, A \geq X + \text{adjust}(A) \}$
- $\text{adjust}(A) = 1 / (10^{\text{places}(A)})$
- $\{ \text{salary} \geq 5, 5 \leq \text{salary}, \}$

WHERE-Teil - Operatorenvielfalt (2)

- 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 >= 5` mit `salary NUMERIC(4,2)`
- A, B : Konstanten oder Attribute, X : numerische Konstante
- $M_6 : \{ A \geq B, B \leq A \}$
 $M_7 : \{ A > X, A \geq X + adjust(A) \}$
- $adjust(A) = 1 / (10^{places(A)})$
- $\{ salary \geq 5, 5 \leq salary, salary \geq 4.99 + 0.01, \}$

WHERE-Teil - Operatorenvielfalt (2)

- 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 >= 5` mit `salary NUMERIC(4,2)`
- A, B : Konstanten oder Attribute, X : numerische Konstante
- $M_6 : \{ A \geq B, B \leq A \}$
 $M_7 : \{ A > X, A \geq X + adjust(A) \}$
- $adjust(A) = 1 / (10^{places(A)})$
- $\{ salary \geq 5, 5 \leq salary, salary > 4.99, \}$

WHERE-Teil - Operatorenvielfalt (2)

- 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 >= 5` mit `salary NUMERIC(4,2)`
- A, B : Konstanten oder Attribute, X : numerische Konstante
- $M_6 : \{ A \geq B, B \leq A \}$
 $M_7 : \{ A > X, A \geq X + adjust(A) \}$
- $adjust(A) = 1 / (10^{places(A)})$
- $\{ salary \geq 5, 5 \leq salary, salary > 4.99, 4.99 < salary \}$

WHERE-Teil - Sortierung (1)

- Baum $T(x)$ mit Wurzel x . $children(x) = (c_1, c_2, \dots, c_n)$
 - Eindeutige Repräsentation erfordert Ordnung von $children(x)$
 - Ist x nicht-kommutativer Operator \rightarrow Hinzufügen aller Schreibweisen
 - Sonst: Ordnung von $children(x)$ beliebig veränderbar
- \rightarrow Ordnung muss definiert werden!
- $order : children \rightarrow \mathbb{N}$
 - Nach Sortieren gilt: $order(c_1) \leq order(c_2) \leq \dots order(c_n)$

| | | | | | | | | | |
|------------|----------|-----------|----|--------|--------|-----|-----|-----|-----|
| r | Variable | Konstante | OR | \leq | \geq | $<$ | $>$ | $=$ | $+$ |
| $order(r)$ | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

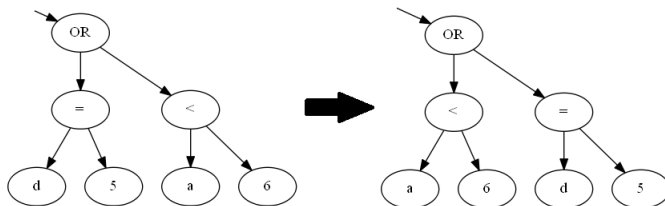
| | | | | | | |
|------------|----|---------|-------------|--------|-----|-----|
| r | — | IS NULL | IS NOT NULL | EXISTS | ANY | ALL |
| $order(r)$ | 10 | 11 | 12 | 13 | 14 | 15 |

- Vorgehen: BOTTOM-UP durch Rekursion

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 / beides Variablen / beides Konstanten sind?

WHERE-Teil - Sortierung (3)

Es sei $T(x)$ ein Parserbaum mit $children(x) = (c_1, c_2, \dots, c_n)$. Haben wir $order(c_i) = order(c_{i+1})$ benötigen wir weiteres Kriterium für Bestimmung eindeutiger Ordnung. Wir benutzen schrittweise Tiefensuche, da alle Bäume unter c_i und c_j bereits sortiert sind.

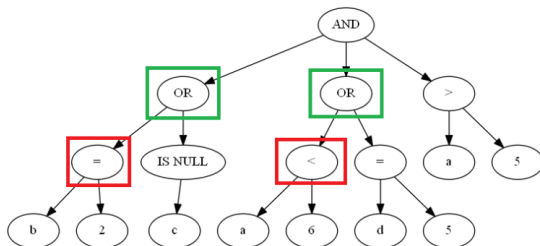
- Es sei $order(c_i) = order(c_j)$ mit $j = i + 1$ und c_i, c_j sind beides keine Konstanten oder Variablen
- $DFS_i(x)$ bezeichnet aktuellen Knoten bei Tiefensuche auf Wurzelknoten x im Schritt i .
- for k=1 to n
 - if ($DFS_k(c_j) == \text{NULL}$) AND ($DFS_k(c_i) \neq \text{NULL}$)
 - swap(c_j, c_i);
 - else if $order(DFS_k(c_j)) < order(DFS_k(c_i))$
 - swap(c_j, c_i);
 - return;
 - endif
- endfor

WHERE-Teil - Sortierung (4)

Gilt $order(c_i) = order(c_j)$ mit $j = i + 1$ und c_i, c_j sind beides Konstanten oder Variablen, dann entscheidet lexikographische Sortierung.

Beispiel:

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



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

Letzter Schritt der Sortierung: Entfernung von Duplikaten.

GROUP BY und ORDER BY

GROUP BY:

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

ORDER BY:

- Ersetzen von Selektionsnummern
- Einführen der künstlichen TV
- ASC bereits durch Parser eingeführt

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`
- Innere Verbunde unter `FROM` werden unter `WHERE` formuliert
- Selbstverbunde: Erstellen von gültigen Permutationen und Hinzufügen zur standardisierten Lösung (i)
- Umwandeln sämtlicher Unterabfragen zu `EXISTS`-Unterabfragen (i)

Ausblick

Folie verschieben!

- Erkennen von unnötigem `DISTINCT`
- transitiv-implizierte Formeln: $a = b \text{ AND } a = 5 \rightarrow b = 5$
- Beschränkung der Domänen: $a > 2 \text{ AND } a > 0 \rightarrow a > 2$
- Behandlung von komplexeren arithmetischen Ausdrücken
- Erweiterung des Parsers um Unterabfragen und Verbund unter `FROM`

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

Allgemeines

- Schritt nur notwendig, wenn Standardisierung fehlschlägt
- Prüfen der notwendigen Bedingung einer Äquivalenz zweier Anfragen
- Soll Ungleichheit nachweisen
- *false positive* nicht möglich
- reale Daten notwendig
- Vorgehen: Ausführen von Anfragen, Vergleich der Ergebnisse
- Unterstützte DBMS: abhängig von JDBC-Connector

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
```

- Fallunterscheidung: Enthält Musterlösung (ML)/ Lösung des Lernenden (LL) ORDER BY?

Fallunterscheidung (1)

ML enthält kein ORDER BY, LL enthält ORDER BY

- Sortierung offensichtlich egal → zunächst Entfernung von ORDER BY von LL

ML enthält ein ORDER BY, LL enthält kein ORDER BY

- Sortierung von Bedeutung
- Lernende ist Sortierung offensichtlich egal
- Ziel: Nachweisen der Ungleichheit → keine Anpassungen
- Problem: zufällige Übereinstimmung

ML enthält kein ORDER BY, LL enthält ORDER BY

- keine Anpassung
- Problem: zufällige Übereinstimmung

Fallunterscheidung (2)

ML enthält kein ORDER BY, LL enthält kein ORDER BY

- Sortierung unwichtig, aber Optimierer bestimmt Ausgabe → kann Falschmeldungen erzeugen
- Lösung: Sortieren der Lösungen nach den Ausgabespalten (iterativ)
- Beispiel: `SELECT surname, firstname FROM people` zu:

```
SELECT surname, firstname  
FROM people ORDER BY 1,2
```

Bei Fall 2 und 3 muss nachträglich nach allen unbenutzten Ausgabespalten sortiert werden. Selbst dann ist keine Eindeutigkeit gewährleistet. → offenes Problem.

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

Vergleich der Anfragen

- 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
- automatisches Build- und Deployskript per *ant*
- Parser: ZQL (Open Source)
- DBMS für Schritt 2: alle mit JDBC-Connector

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) unabdingbar
 - 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 Verbundskonzepte
 - komplexe arithmetische Ausdrücke
 - Erkennen von unnötigem `DISTINCT`
 - Ausweitung auf Behandlung aller SQL-Ausdrücke (`UPDATE`, `DELETE`)
 - Ausarbeitung und Implementierung weiterer Konzepte (zB: unnötige Verbunde, ...)
 - Ausbau der Webseite: Gruppierung von Aufgaben, Einfügen von Kategorien, ...

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