# Ranking von Produktempfehlungen mit präferenz-annotiertem SQL

Matthias Beck, Sven Radde und Burkhard Freitag

{Matthias.Beck, Sven.Radde, Burkhard.Freitag}@uni-passau.de

**Abstract:** Web-basierte, datenbankgestützte Beratungssysteme finden durch zentrale Wartbarkeit bei sich permanent ändernden Produktpaletten aktuell starke Verbreitung. Kernpunkt für eine optimale Produktempfehlung ist dabei die Berücksichtigung der Präferenzen des Kunden, welche auf eine möglichst *einfache und nachvollziehbare* Art und Weise spezifiziert werden sollten. Daher wird ein Ansatz präsentiert, der erlaubt, die vom Benutzer ohnehin anzugebenden Selektionsbedingungen zusätzlich mit Gewichten zu *annotieren* und damit die Sortierung der Empfehlungen zu beeinflussen. Dies wird durch eine erweiterte SQL-Syntax ermöglicht, über die theoretisch fundiert ein *Ranking* auf der Ergebnismenge definiert wird.

## 1 Einleitung

Der Bedarf an Beratungssystemen steigt durch die zunehmende Kontextualisierung ständig. Eines der prominentesten Beispiele sind die personalisierten Buchempfehlungen von Amazon [LSY03]. Wichtig ist, die Präferenzen des Nutzers zu berücksichtigen und für ein Ranking der Treffermenge zu verwenden. Es ist daher wünschenswert, dem Nutzer die Möglichkeit zu geben, seine Wünsche und Präferenzen auf einfache und verständliche Weise zu spezifizieren.

Der Beitrag dieser Arbeit ist eine einheitliche Methode zur Annotation von SQL-Anfragen mit Gewichten, die zusätzlich die Angabe von *Softconstraints* erlaubt. Basierend auf den angegebenen Gewichten wird ein Ranking der Resultatrelation definiert. Dabei bleibt im Gegensatz zu anderen Verfahren (s. Abschnitt 7) die *Antwortsemantik* der Anfrage erhalten. Dies und die Tatsache, dass nur die SQL-Anfrage annotiert wird, macht die Benutzung beliebiger Datenbanksysteme möglich. Das schließt aber nicht aus, durch Anpassung des Anfrageoptimierers die Ranking-Information bereits bei der Anfrageauswertung nutzbringend einzusetzen.
Die weiteren Teile des Artikels sind folgendermaßen strukturiert. Abschnitt 2 beschreibt kurz den Anwendungsfall. In Abschnitt 3 wird auf die Gewichtsannotationen in SQL eingegangen, die dann in der formalen Definition des Rankings (Abschnitt 4) verwendet werden. Bevor der Artikel mit einer Diskussion (Abschnitt 7) endet, werden in Abschnitt 5 Softconstraints u. ä. beschrieben.

## 2 Anwendungsfall

Der heutige Markt für Mobilfunkprodukte ist im wesentlichen durch kurze Produktzyklen und die hohe Innovationsfolge der technischen Entwicklung gekennzeichnet. Dabei wird besonders deutlich, dass Kunden nur durch qualifizierte Beratung eine optimale Kaufentscheidung treffen können, was einen dauerhaft hohen Schulungsaufwand für das Personal bedeutet. Datenbankgestützte Beratungssysteme können aufgrund ihrer zentralen Wartbarkeit zeitnah auf Änderungen der Produktlandschaft reagieren. Sie haben somit den Vorteil, das Beratungspersonal stets mit aktuellen Daten und Informationen zum Produktkatalog versorgen zu können. Dadurch unterstützen sie den Beratungsverlauf, was den Schulungsbedarf des Personals bei gleichbleibend hoher Beratungsqualität reduziert.

Darüber hinaus stellen die heute im Web verfügbaren "Beratungssysteme" meist eher einfache "Konfiguratoren" dar, welche vom Kunden ein relativ großes Domänenwissen fordern, um gute Resultate zu liefern. Hochwertige Onlineberatung kann hier einen großen Beitrag zur Kundenzufriedenheit leisten und auch dazu dienen, neue Käuferschichten zu erschließen, welche von den bisherigen Möglichkeiten des Online-Shoppings nicht angesprochen werden.

Hierbei steht zunächst die Ermittlung der Kundenpräferenzen im Mittelpunkt. Für den Anwendungsfall ergibt sich, dass diese sich sehr gut auf die Zugehörigkeit des Produkts zu *Klassen* abbilden lassen und *Gewichtungen* enthalten, wie in der folgenden beispielhaften Kundenäußerung: *„Ich hätte gerne ein Handy hauptsächlich mit Multimedia- aber auch mit Businessfunktionen."* Für die Produktempfehlung sollten also Geräte, welche sowohl der Klasse „Multimediahandy" als auch der Klasse „Businesshandy" angehören, bevorzugt dargestellt werden. Reine Multimedia- oder Businesshandys sind ebenfalls akzeptabel (wobei reine Multimediahandys wiederum bevorzugt werden), während Mobiltelefone völlig ohne entsprechende Funktionen dem Kunden nicht angeboten werden sollen.

Die Sortierung der Empfehlungen kann verfeinert werden, sobald detailliertere Informationen über die Präferenzen des Kunden verfügbar sind. Beispielsweise ergeben sich innerhalb der Klasse „Multimediahandy" sicherlich Qualitätsunterschiede z.B. hinsichtlich der Auflösung der Digitalkamera oder bei den vom Medienplayer abspielbaren Formaten, so dass die Resultatmenge innerhalb dieser Klasse weiter sortiert werden kann. Die Präferenzen des Benutzers bestimmen wieder die Gewichtung zwischen diesen beiden Faktoren (also ob ggf. eine bessere Digitalkamera oder ein besserer Medienplayer bevorzugt wird). Zu beachten ist, dass diese feinere Sortierung innerhalb der vorher definierten gröberen Klassen erfolgt.

Neben der Aufgabe, diese Präferenzen vom Kunden zu erfragen, muss einem Beratungssystem also eine geeignete Technik zur Verfügung stehen, an den Produktkatalog Anfragen entsprechend den oben beispielhaft erläuterten Anforderungen zu stellen. Aufgrund der weiten Verbreitung relationaler Datenbanken bietet es sich an, hierfür herkömmliche SQL-Anfragen in geeigneter Weise zu annotieren, so dass die Präferenzen des Kunden als Gewichte in die Auswertung einfließen können.

# 3 Gewichtsannotiertes SQL

Das gewichtsannotierte SQL soll zwei Anforderungen erfüllen. Erstens ist es notwendig, die Zugehörigkeit zu unterschiedlichen *Klassen* unterschiedlich gewichten zu können, und zweitens sollen diese Gewichte ein Ranking und damit eine Sortierung der Resultatmenge ermöglichen. Dies erhält dem Kunden seine freie Wahlmöglichkeit, während die empfohlenen Produkte bevorzugt dargestellt werden können (in der Form von „Top-10-Empfehlungen“ o.ä.).

In einem Beispielszenario soll nun die Zugehörigkeit zur Klasse *MultimediaHandy* mit einem Gewicht von 2 und zur Klasse *BusinessHandy* mit einem Gewicht von 1 annotiert werden. Nimmt man der Einfachheit halber an, dass diese beiden Klassen jeweils durch ein boolesches Attribut in einer Tabelle *handy* repräsentiert werden, so kann die Anfrage folgendermaßen formuliert werden:

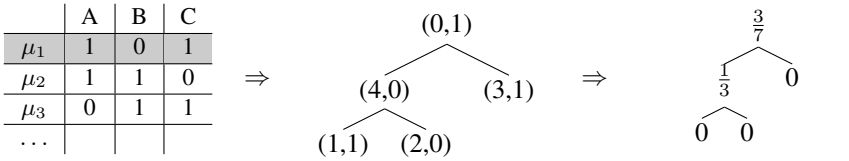
**Beispiel 1**

```
SELECT * FROM handy
WHERE (MultimediaHandy = 1) [2] OR (Businesshandy = 1) [1]
```

Die intuitive Bedeutung dieser Anfrage ist dabei wie folgt: In der sortierten Resultatmenge sollen die Mobiltelefone, die zu beiden Klassen gehören, an der Spitze stehen, gefolgt von den reinen Multimedia-Handys, zuletzt die reinen Business-Handys. Besteht die Selektionsbedingung wie im Beispiel aus einem n-stelligen *OR*, so steigt die Anzahl der Klassen-Kombination und damit, bei geeignet gewählten Gewichten, die Anzahl der Werte, die der Rang eines Tupels annehmen kann, exponentiell in der Anzahl der Operanden $n$. Fügt man in die *OR*-Bedingung aus Beispiel 1 nur einen weiteren Operanden ein, so erhöht sich die Anzahl der Klassen bereits auf acht. Somit lässt sich bereits mit einer relativ kleinen Anzahl von gewichteten Ausdrücken ein differenziertes Ranking erreichen. Dabei müssen für jedes Tupel aber nur die spezifizierten $n$ Bedingungen ausgewertet werden.

Die Realisierung einer solchen Sortierung muss folgende Anforderungen erfüllen: Ein Gewicht 0 darf keinen Einfluss auf die Sortierung nehmen. Je höher andererseits die Gewichte der Klassen sind, denen ein Element der Resultatmenge angehört, umso besser soll der Rang des Elementes sein.

Allgemein gilt die Form $(expression)[weight]$ für einen gewichtsannotierten Ausdruck. $expression$ steht dabei für einen beliebigen booleschen Ausdruck, der in der *Where*-Klausel einer Anfrage gestattet ist. Erlaubt sind insbesondere Ausdrücke der Form *(table.column* $<=$ *const)*, geschachtelte Ausdrücke wie *(E and F)* und Ausdrücke, die Subqueries enthalten, wie *(exists select . . .)*. Ebenfalls möglich ist eine Schachtelung von Gewichten: *(((A) [1] and (B) [4]) [2] or (C) [3])*. Falls für einen Ausdruck kein Gewicht angegeben ist, wird als Standard ein Gewicht von 0 verwendet.
Eine natürliche Einschränkung ist, dass gewichtsannotierte Ausdrücke nur als (direkte) *Bestandteile* einer Disjunktion oder Konjunktion möglich sind. Ein allein stehender Ausdruck der Form *(A or B) [1]* ist daher nicht sinnvoll, im Gegensatz zu *((A) [1] or (B) [2])*. Diese Restriktion bedeutet keine Einschränkung der Ausdrucksmächtigkeit, denn nur innerhalb von Disjunktionen bzw. Konjunktionen gibt es mehrere Klassen, die relativ zueinander gewichtet werden können. Dies ist beim ersten Ausdruck nicht der Fall. Wäre




(a) Tupel (b) Rank. Map. Tree f. $\mu_1$ (c) Ranking Tree f. $\mu_1$

Abbildung 1: Ablauf des Rankings für die Query *(A [1] and B [2]) [4] or C [3]*

dieser Ausdruck das Selektionskriterium einer Anfrage, dann würden *alle* Elemente des Resultats mit 1 gewichtet werden, da sie alle die Bedingung *(A or B)* erfüllen.

# 4 Ranking

Die in Abschnitt 3 beschriebenen Gewichtsangaben sind die Grundlage des Rankings der Resultatrelation. Dazu muss für jedes Tupel der Ergebnisrelation die Erfüllung bzw. Nicht-Erfüllung aller gewichtsannotierten Teilausdrücke der *Where*-Klausel geprüft werden. Formal beschrieben wird dieser Vorgang durch das Rank Mapping in Abschnitt 4.1. Für jedes Tupel der Ergebnismenge erhält man dadurch einen Rank Mapping Tree. Die Information des Rank Mapping Trees wird mit Hilfe einer Aggregationsfunktion zum Ranking Tree verdichtet. Auf den Ranking Trees wird eine Ordnung definiert (s. 4.2). Basierend auf dieser Ordnung wird das abschließende Ranking der Ergebnisrelation durchgeführt.

## 4.1 Rank Mapping

Für eine gegebene gewichtsannotierte Query $Q$ wird zunächst die Ergebnisrelation[1] $R$ berechnet. Anschließend wird auf jedes Tupel $\mu \in R$ das Rank Mapping angewendet. Ergebnis dieses Mappings ist dann ein *Rank Mapping Tree* für jedes Ergebnistupel.

**Definition 1 (Rank Mapping)** *Sei $DB$ eine Datenbankinstanz, $Q$ eine Selektionsbedingung mit Gewichtsannotationen und $\mu$ ein Resultattupel bzgl. $Q$. Sei weiter $\diamond \in \{\texttt{AND}, \texttt{OR}\}$. Dann definieren wir das Rank Mapping $r_{DB}^{start}(Q, \mu)$ durch*

$$r_{DB}^{start}(Q, \mu) := (0, 1, r_{DB}(Q, \mu)) \quad (1)$$

$$r_{DB}(C_1^{r_1} \diamond \ldots \diamond C_n^{r_n}, \mu) := ((r_1, d_1, r_{DB}(C_1, \mu)), \ldots, (r_n, d_n, r_{DB}(C_n, \mu))) \quad (2)$$

$$\textit{wobei} \qquad d_i = \begin{cases} 1, & \textit{falls } C_i[\mu] \textit{ wahr in } DB \\ 0, & \textit{sonst} \end{cases}$$

[1] Zur Berechnung der Gewichte ist die Ergebnisrelation *vor* einer Projektion auf die Elemente der Select-Liste erforderlich.

$$r^{\neg}_{DB}(C_1^{r_1} \diamond \ldots \diamond C_n^{r_n}, \mu) := ((r_1, d_1, r^{\neg}_{DB}(C_1, \mu)), \ldots, (r_n, d_n, r^{\neg}_{DB}(C_n, \mu))) \quad (3)$$

$$\textit{wobei} \qquad d_i = \begin{cases} 1, & \textit{falls } (not\ C_i)[\mu] \textit{ wahr in DB} \\ 0, & \textit{sonst} \end{cases}$$

$$r_{DB}(\texttt{NOT}\ C, \mu) := r^{\neg}_{DB}(C, \mu) \quad (4)$$

$$r^{\neg}_{DB}(\texttt{NOT}\ C, \mu) := r_{DB}(C, \mu) \quad (5)$$

$$r_{DB}(A, \mu) := (), \textit{ falls } A \textit{ elementar} \quad (6)$$

$$r^{\neg}_{DB}(A, \mu) := (), \textit{ falls } A \textit{ elementar} \quad (7)$$

In den Gleichungen 2 und 3 wird für jeden Teilausdruck einer Konjunktion oder Disjunktion geprüft, ob er für das aktuell betrachtete Tupel $\mu$ erfüllt ist. Dieses Ergebnis (0 oder 1) wird zusammen mit der Gewichtung dieses Teilausdruckes gespeichert. Zusammen bilden diese beiden Werte $(r, b)$ einen Knoten des entstehenden Rank Mapping Trees. Anschließend wird das Mapping rekursiv für die einzelnen Teilausdrücke durchgeführt – es werden die Kinder von $(r, b)$ im Rank Mapping Tree berechnet. Negation wird in den Gleichungen 4 und 5 behandelt. Die Negation wird hier durch Verwendung der $r^{\neg}$-Funktion ‚gespeichert' und dann bei der Auswertung wieder angewendet (s. Gleichung 3). Für elementare Ausdrücke wird in den Gleichungen 6 und 7 ein Standardwert definiert. Zur Definition des Wurzelknotens dient die $r^{start}$-Funktion (1).
Durch die Duplikateliminierung bei einer möglichen anschließenden Projektion auf die Attribute der *Select*-Liste kann der Fall auftreten, dass zu einem Resultattupel $\mu$ mehrere Ranking Trees $T_\mu$ entstehen. In einem solchen Fall wird ein ‚guter' Ranking Tree $t \in T_\mu$ gewählt, d. h. es muss gelten: $\forall t' \in T_\mu : t \not< t'$.
Abbildung 1a-b zeigt an einem Beispiel den Aufbau des Rank Mapping Trees. Dieser speichert die grundlegenden Informationen des Rankings. Der Wurzelknoten *(0,1)* ist vordefiniert. Alle weiteren Knoten ergeben sich durch Auswertung von Teilbedingungen der annotierten Query *(A [1] and B [2]) [4] or C [3]*. So ergibt sich der Knoten *(3,1)* durch Auswertung des Ausdrucks *C* bezüglich des betrachteten Tupels $\mu_1$. Die erste Komponente ist das Gewicht des Teilausdrucks (*3*). Die zweite Komponente (*1*) zeigt, dass für $\mu_1$ der Teilausdruck $C$ erfüllt ist. Zur besseren Übersicht enthält der Baum in Abbildung 1b keine (leeren) Blätter (vgl. Gleichungen 6 und 7 in Definition 1).

Für eine sinnvolle Auswertung ist es erforderlich, diese Informationen unter Verwendung einer *Aggregationsfunktion* zu verdichten. Resultat dieser Aggregation ist dann der *Ranking Tree.*

**Definition 2 (Level Aggregation Function)** *Eine Funktion* $\bigcup_{i \in \mathbb{N}_0} (\mathbb{N} \times \{0, 1\})^i \to \mathbb{R}$ *ist eine Level Aggregation Function.*

**Definition 3 (Weighted Average Level Aggregation Function)**

$$agg_{wa}((n_1, b_1), \ldots, (n_m, b_m)) := \begin{cases} 0, & \textit{falls } \forall 1 \leq i \leq m : n_i = 0 \\ \frac{\sum_{i=1}^{m} n_i b_i}{\sum_{i=1}^{m} n_i}, & \textit{sonst} \end{cases}$$

Nach den bisher gesammelten Erfahrungen ist der gewichtete Durchschnitt als Level Aggregation Function für die vorgestellten Anforderungen gut geeignet. Alternativen wären

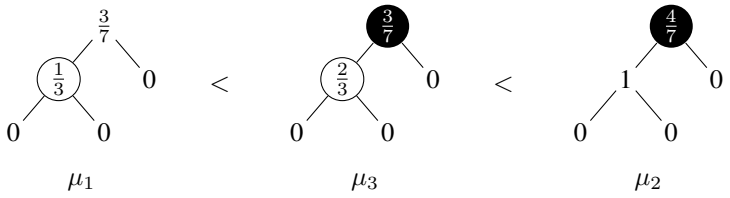



Abbildung 2: Ordnung auf Ranking Trees

etwa Maximum, gewichtete Summe etc. Der Ranking Tree entsteht nun durch rekursive Anwendung einer Level Aggregation Function auf jeden Knoten des ursprünglichen Rank Mapping Trees.

**Definition 4 (Tree-based Level Aggregation)** *Sei $agg$ eine Level Aggregation Function, $t$ ein Rank Mapping Tree. Dann wird die Tree-based Level Aggregation $tla$ induktiv folgendermaßen definiert:*

1. *Falls $t = (r, b, ((r_1, b_1, sub_1), \ldots, (r_n, b_n, sub_n)))$ mit $n > 0$,*
   *so sei $tla(t) := (agg((r_1, b_1), \ldots, (r_n, b_n)), tla(sub_1), \ldots, tla(sub_n))$,*
2. *sonst $tla(t) := (0)$*

In Abbildung 1b-c wird exemplarisch der Übergang vom Rank Mapping Tree zum Ranking Tree unter Verwendung des gewichteten Durchschnitts gezeigt. Der Wurzelknoten $\frac{3}{7}$ beispielsweise ergibt sich durch Berechnung des gewichteten Durchschnitts aus den Tupeln *(4,0)* und *(3,1)*, d. h. $\frac{4 \cdot 0 + 3 \cdot 1}{4+3}$.

### 4.2 Ranking Tree

Jedem Tupel der Ergebnisrelation wird durch das Rank Mapping ein Ranking Tree zugeordnet.

**Definition 5 (Ranking Tree)**

1. $(0)$ *ist ein Ranking Tree (Blattknoten).*
2. *Sei $r \in \mathbb{R}$ und seien $t_1, \ldots, t_n$ Ranking Trees mit $n \geq 1$. Dann ist $(r, t_1, \ldots, t_n)$ ein Ranking Tree.*

Um das Ziel eines Rankings der Ergebnisrelation auf Grundlage der erhaltenen Bäume zu erreichen, ist es erforderlich, eine Ordnung auf Ranking Trees zu definieren. Die Definition der Ordnung erfolgt auf Grundlage der Ebenen des Baumes. Ein solches Level wird dabei wie folgt definiert:

**Definition 6 (Ranking Tree Level)** *Sei $a = (r, a_1, \ldots, a_n)$ ein Ranking Tree. Dann definieren wir induktiv die Funktion $level$. Das resultierende $n$-Tupel $(r_1, \ldots, r_n)$ ist ein Ranking Tree Level.*

$$
\begin{aligned}
level_0(a) &:= (r) \\
level_m(a) &:= \begin{cases} level_{m-1}(a_1) \circ \cdots \circ level_{m-1}(a_n), & \textit{falls } n > 0 \\ (), & \textit{sonst} \end{cases}
\end{aligned}
$$

Abbildung 2 veranschaulicht den Grundgedanken der Ordnung. Die drei abgebildeten Ranking Trees entstehen aus den Tupeln $\mu_1$ bis $\mu_3$ von Abbildung 1 bei Verwendung des gewichteten Durchschnitts. Wenn die Wurzeln der Bäume bereits eine Unterscheidung ermöglichen, müssen die Kindknoten nicht weiter betrachtet werden (2b-c). Bei gleicher Wurzel entscheidet die Ebene der Kindknoten (2a-b). Sollte auch diese Ebene übereinstimmen, wird die nächste Ebene einbezogen usw.

**Definition 7 (Strict Order on Ranking Trees)** *Es seien zwei Ranking Tree Level $l = (l_1, \ldots, l_n)$ und $k = (k_1, \ldots, k_n)$ gegeben. Dann gilt $l < k$ g.d.w. $\forall 1 \leq i \leq n : l_i < k_i$. Seien jetzt $a$ und $b$ Ranking Trees mit gleicher Struktur. Dann gilt $a < b$ g.d.w.*

$$\exists m : level_m(a) < level_m(b) \quad \textit{und} \quad \forall 0 \leq i < m : level_i(a) = level_i(b)$$

Die entstehende Ordnung ist partiell. Sie lässt sich jedoch zu einer totalen Ordnung vervollständigen, indem unvergleichbare Elemente auf konsistente Weise in die Ordnung integriert werden. Dies ist im einfachsten Fall z. B. unter Rückgriff auf den Tupelidentifier o. ä. möglich.
Die Ranking Trees haben eine weitere nützliche Eigenschaft, bieten sie doch durch ihre Struktur direkt eine Erklärungskomponente. Jede Gewichtung, die der Nutzer vergeben hat, findet sich im Ranking Tree repräsentiert und kann so mit minimalem Nachbearbeitungsaufwand direkt verwendet werden, um dem Nutzer Hinweise zugeben, wie einerseits das aktuelle Ranking zustande kommt und wie er andererseits durch Veränderung der Gewichte ein verändertes Ranking bewirken kann.

## 5 Erweiterte Möglichkeiten

Oft sind bestimmte Eigenschaften eines Mobiltelefons nicht zwingend erforderlich, wären aber wünschenswert. Die Erfüllung einer solchen Bedingung, eines *Softconstraints*, sollte sich in einem höheren Rang in der Ergebnisrelation widerspiegeln.

Die bereits bekannte Anfrage aus Beispiel 1 kann erweitert werden, so dass Sonderangebote einen höheren Rang erhalten. Genauer ausgedrückt, sollen innerhalb der Klassen *reines Multimediahandy*, *Multimedia- und Businesshandy* sowie *reines Businesshandy* immer die Sonderangebote zuoberst erscheinen. *Innerhalb* der ursprünglichen Gewichtung lässt sich dadurch das Ranking weiter verfeinern. Die Bedingung *Sonderangebot* ist dabei ein Softconstraint, da weiterhin auch Handys, die kein Sonderangebot sind, im Ergebnis erscheinen sollen. Ausdrücken lässt sich diese Anforderung wie folgt:

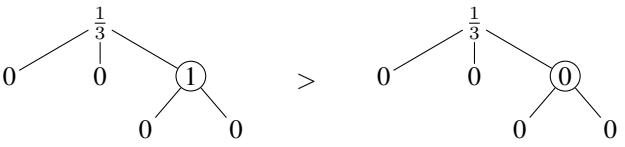



Abbildung 3: Ranking Trees mit Softconstraint

```
SELECT * FROM handy
WHERE (MultimediaHandy = 1) [2] OR (BusinessHandy = 1) [1]
      OR ((Sonderangebot = 1) [1] AND (0=1))
```

Abbildung 3 zeigt die Ranking Trees für zwei reine Businesshandys, von denen *nur* das linke ein Sonderangebot ist. Durch die unerfüllbare Teilbedingung *(0=1)* wird der gesamte Teil *(Sonderangebot = 1) AND (0=1)* unerfüllbar. Die eigentliche Selektionsbedingung bleibt also im Vergleich zur ursprünglichen Anfrage aus Beispiel 1 unverändert. Andererseits unterscheiden sich, wie Abbildung 3 zeigt, die Ranking Trees, je nachdem ob ein Sonderangebot vorliegt oder nicht. Diese Information kommt dann beim Ranking der Resultatmenge zum Einsatz. Die etwas umständliche Formulierung der Form *(... or (Softconstr [n] and false))* lässt sich durch Einführung eines neuen Schlüsselwortes *SOFT* leicht zu *(... or (SOFT Softconstr [n]))* umformen, um die Lesbarkeit zu erhöhen.
Das Beispiel zeigt Softconstraints auf der *OR-Ebene*. Analog dazu gibt es auch die Möglichkeit, Softconstraints auf der *AND-Ebene* auszudrücken. Das entsprechende Muster ist *(*$Bed_1[r_1]$ *and . . . and* $Bed_n[r_n]$ *and (Softconstr [r] or true))*.

Der vorgestellte Ranking-Mechanismus erlaubt, verschiedene Eigenschaften mit verschiedenen Gewichten zu versehen. Dies bietet die Möglichkeit, das Ranking sehr fein an die Anforderungen des Nutzers anzupassen. In einigen Fällen ist aber eine leicht veränderte Sichtweise vorzuziehen: Von einer Menge gegebener Eigenschaften sollen *möglichst viele* erfüllt sein. Ein Beispiel dafür sind etwa Verbindungsmöglichkeiten eines Handys, wie Bluetooth, Infrarot, USB, WLAN etc. Dies lässt sich sehr einfach durch Gleichgewichtung der Eigenschaften erreichen:

```
SELECT ... FROM ...
WHERE ((Bluetooth=1) [1] OR (Infrarot=1) [1]
      OR (USB=1) [1] OR ...)
```

## 6 Implementierung

Die Realisierung der Auswertung von gewichtsannotierten SQL-Statements ist zweigeteilt. In einem ersten Schritt wird das annotierte SQL-Statement geparst und analysiert. Dann wird aus dem annotierten SQL-Statement ein leicht verändertes Standard-SQL-Statement erzeugt. Dies wird an die Datenbank übergeben und ausgewertet. Die Resultate werden eingelesen, die Ranking Trees werden berechnet und die Resultatrelation sortiert.

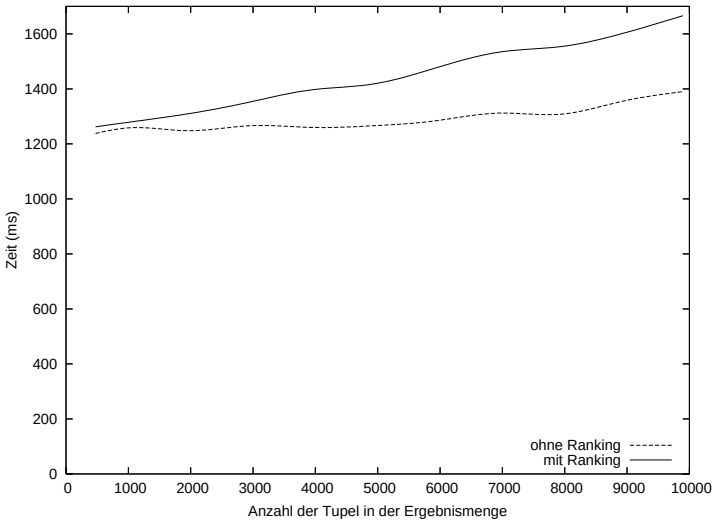



Abbildung 4: Auswertungszeit in Abhängigkeit von der Ergebnisgröße

Die Laufzeit des Verfahrens liegt für die im Anwendungsfall zu erwartenden Problemgrößen von maximal wenigen tausend Tupel in einem guten Bereich. So benötigt die Anfrage aus Beispiel 1 bei einer Ergebnisgröße von 1000 Tupeln 1278 ms, während die gleiche Anfrage als reines SQL ohne Ranking 1258 ms benötigt.[2] Selbst bei einer Ergebnisgröße von 10.000 Tupeln ist der durch das Ranking hinzukommende Overhead im Vergleich zur Gesamtlaufzeit gering (mit Ranking 1666 ms, ohne Ranking 1390 ms).

### 6.1 SQL

Für den Aufbau der Ranking-Trees muss für jedes Tupel der Ergebnisrelation die Information vorliegen, welche Bedingungen und Teilbedingungen der *Where*-Klausel für dieses Tupel erfüllt sind. Übertragen auf Beispiel 1 muss für jedes Resultattupel bekannt sein, ob es die Bedingungen *(MultimediaHandy = 1)* und/oder *(BusinessHandy = 1)* erfüllt. Dies lässt sich durch *Case*-Klauseln in der *Select*-Liste erreichen. Folgendes Statement wird auf diese Weise generiert:

```
SELECT h.*,
CASE WHEN (MultimediaHandy=1) THEN 1 ELSE 0 END AS RANK_EXPR_0,
CASE WHEN (BusinessHandy=1) THEN 1 ELSE 0 END AS RANK_EXPR_1
FROM Handy h WHERE (MultimediaHandy = 1) AND (BusinessHandy = 1)
```

[2] Gemessen auf einem Pentium mit 1,5 GHz und 1 GB RAM mit einer PostgreSQL 8.0 Datenbank.

In diesem einfachen Fall könnten die Werte *MultimediaHandy* und *BusinessHandy* auch direkt selektiert werden, da ihr Wertebereich ebenfalls $\{0, 1\}$ ist. Dies ist für allgemeine Bedingungen nicht möglich, so dass dann *Case*-Klauseln zwingend erforderlich sind.

### 6.2 JDBC-Treiber

Als Schicht zwischen dem (Java-)Anwendungsprogramm und der Datenbank wurde ein eigener JDBC-Treiber entwickelt, der sich für die Kommunikation mit der Datenbank auf einen datenbankspezifischen Standard-SQL-Treiber abstützt. Diese Schicht enthält Funktionalität zum Parsen des annotierten SQL-Statements, zum Erzeugen des Standard-SQL-Statements und zum abschließenden Aufbau der Ranking-Strukturen und Sortieren der Resultatrelation.

```
// Laden des Standard-Datenbank-Treibers
Class.forName("com.ibm.db2.jcc.DB2Driver");
// Laden des Ranking-Datenbank-Treibers
Class.forName("de.uni_passau.im.pref.sql.ranking.jdbc.RankedDriver");

DriverManager.getConnection("jdbc:rank:db2:...", user, pwd);
...
```

Zuerst werden sowohl der Standard-Datenbank-Treiber, in diesem Fall der DB2-Treiber, als auch der *RankedDriver* beim Java-Drivermanager registriert. Der RankedDriver erklärt sich dabei für Datenbank-URLs der Form *jdbc:rank:...* zuständig. Soll der Drivermanager nun eine Verbindung zu einer solchen rank-URL aufbauen, wird der RankedDriver angesprochen. Dieser baut dann wieder über den Drivermanager eine Standard-Verbindung zur DB2-Datenbank auf, unter Benutzung des DB2-Treibers. Das weitere Verfahren, insbesondere Erstellung eines Statements, Abrufen von Resultaten usw. erfolgt dann wie bei JDBC allgemein üblich. Somit ist die gesamte Funktionalität im RankedDriver gekapselt und für den Benutzer nach dem Laden des Treibers vollkommen transparent.

## 7 Zusammenfassung und Diskussion

Zur Realisierung von Beratungssystemen weit verbreitet ist der Einsatz der *Collaborative-Filtering*-Methode [CSP03, MS02], der u. a. in [LSY03] genutzt wird. Unser Ansatz konzentriert sich auf die explizite Ermittlung der Kundenpräferenzen, anstatt diese implizit über die schrittweise Bewertung von Beispielprodukten aus dem Katalog zu lernen. Durchaus möglich wäre es aber, die Gewichte aus bisher getroffenen (Kauf-)Entscheidungen des Nutzers zu lernen.

Viele Ansätze beschäftigen sich mit dem Zusammenspiel von Datenbanken, Ranking und Präferenzen. Schon Ende der 80er Jahre beschrieben Lacroix und Lavency [LL87] eine Erweiterung des Domänenkalküls um Präferenzen.

Kießling und andere haben mit PreferenceSQL [KK02, Kie02, Kie05, HK05] eine Methode zur Repräsentation von Präferenzen in SQL entwickelt, die für gegebene Präferenzen *ausschließlich* die besten Treffer ermittelt („Best Matches Only"-Semantik). In der zugrunde liegenden Preference Algebra wird dies durch einen den sog. Preference-Query-Operator $\sigma[P](R)$ (für Präferenz $P$, Relation $R$) erreicht, dessen deklarative Semantik fordert, dass sich genau die Tupel für die Ergebnismenge qualifizieren, die ein sog. Perfect Match bzgl. der Präferenz $P$ darstellen. Die Standard-Antwortsemantik von PreferenceSQL und unserem Ansatz unterscheidet sich hier deutlich. Denn anders als bei PreferenceSQL ändert unser Beitrag die Mengen-Semantik von SQL nicht, sondern ordnet lediglich die Ergebnisrelation um. Der Kunde kann somit aus der gesamten Treffermenge wählen und wird durch evtl. nicht explizit formulierte Präferenzen nicht behindert. Die Grundlage von PreferenceSQL bilden strikte partielle Ordnungen (z. B. ein schwarzes Auto ist »besser« als ein weißes Auto). Vorteilhaft an PreferenceSQL ist die reichhaltige formale Preference Algebra. Somit können Optimierungen – ähnlich wie bei Standard-SQL – mittels geeigneter Umformungsregeln schon auf der syntaktischen Ebene durchgeführt werden, wie beispielsweise in [HK05] beschrieben. Neben diesen Präferenzen gibt es auch die Möglichkeit, mit numerischen Gewichten zu arbeiten. Der Unterschied zu unserem Ansatz liegt darin, dass in PreferenceSQL bestimmten Attributwerten Gewichte zugeordnet werden. Beispielsweise könnte der Farbe »weiß« ein Gewicht $w$ zugeordnet werden. Im Gegensatz dazu wird bei unserem Ansatz die Zugehörigkeit zu einer Klasse, d. h. einer beliebig komplexen booleschen Bedingung über Attributwerten, gewichtet. Gewichte für derartige Bedingungen ließen sich in PreferenceSQL nur schwer nachbilden, insbesondere bei komplexeren Bedingungen, die beispielsweise Unterabfragen enthalten. Ein weiterer Unterschied ist, dass in PreferenceSQL Präferenzen ausschließlich über Softconstraints definiert sein dürfen. Unser Ansatz erlaubt dagegen die Spezifikation von Gewichten sowohl für Hard- als auch für Softconstraints. Um Präferenzen in PreferenceSQL spezifizieren zu können, wurde die Syntax des SQL-Select-Statements stark erweitert um die *Preferring*-, *Grouping*- und *But-only*-Klauseln. Im Unterschied dazu reicht bei unserem Ansatz eine kleine Erweiterung der *Where*-Klausel um numerische Gewichte bereits aus. Ein Recommender-System auf Grundlage von PreferenceSQL wird in [SEK06] beschrieben.

Chomicki präsentiert in [Cho03] ein Framework zur Angabe von Präferenzen in logischen Ausdrücken und deren Einbettung in die relationale Algebra. Auch in diesem Ansatz werden Präferenzen zur Einschränkung der Ergebnismenge verwendet. Ein weiteres Framework zur Kombination von Präferenzen und Datenbanken beschreiben Agrawal und Wimmers in [AW00]. Die Präferenzen werden in diesem Ansatz als komplexe Präferenz-Funktionen modelliert. Ein Ranking der Resultatrelation erreichen Li und andere [LCIS05] mit *RankSQL*. Sie erweitern die relationale Algebra um einen Ranking Operator zur sog. Rank-relational Algebra. Daher lässt sich der Ansatz nicht auf Standarddatenbanken anwenden, sondern muss in den Datenbankkern integriert werden. Außerdem basiert das Ranking nicht auf Präferenzen, sondern auf komplexen Ranking-Prädikaten. Daneben gibt es einige weitere Ansätze, die sich mit der Umsetzung von top-k Anfragen innerhalb der Datenbank beschäftigen [CK97, IA05, ISA+04, IAE04]. Top-k Anfragen außerhalb des Datenbankkerns werden u. a. in [CG99, HKP01, YYY+03] untersucht.

Eine Übertragung des PageRank-Algorithmus (Google) auf relationale Datenbanken präsentieren Hwan und andere in [HHP06].

Wir haben einen Ansatz für das gewichtsbasierte Ranking von Datenbankanfragen vorgestellt. Das Ranking basiert ausschließlich auf der Anfrage, die mit Gewichten annotiert wird. Einer der Vorteile liegt darin, dass der Nutzer genau das Ranking erhält, das er selbst angegeben hat. Daher ist das Ranking personalisiert und selbsterklärend. Die Mengen-Semantik der Anfrage bleibt unverändert, so dass diese Methode mit beliebigen Standard-Datenbanken kombiniert werden kann. Der Ansatz ist nicht auf Datenbanken beschränkt. In [BF06] wurde eine ähnliche Methode für das Ranking von Anfragen in Beschreibungslogiken vorgestellt. Dies macht insbesondere das Ranking von Anfragen im *Semantic Web* möglich.
Eine Implementierung des Ansatzes mit einem guten Laufzeitverhalten wurde vorgestellt. Die Bereitstellung des JDBC-Treibers ermöglicht die transparente Integration in Java-Applikationen. Weitere Optimierungen unseres Algorithmus sind geplant. So kann durch weitere Aggregation vom Ranking Tree zu einer Vektor-basierten Darstellung übergegangen werden. Weiter ist parallel zu dem vorgestellten Ansatz mit seinem Vorteil der breiten Einsatzmöglichkeiten auch eine datenbankkern-nähere Implementierung angedacht. Insbesondere kann die Ranking-Information durch Anpassung des Optimierers bereits bei der Anfrageauswertung verwendet werden. Dadurch wird die Realisierung eines *Top-k-Operators* möglich. Noch offen ist die Frage einer geeigneten Bedienoberfläche für die Eingabe der Präferenzen durch den Nutzer.

## 8 Danksagung

Wir danken den anonymen Gutachtern für ihre hilfreichen Kommentare.

## References

[AW00] Rakesh Agrawal und Edward L. Wimmers. A Framework for Expressing and Combining Preferences. In *SIGMOD '00: Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Seiten 297–306, New York, NY, USA, 2000. ACM Press.

[BF06] Matthias Beck und Burkhard Freitag. Semantic Matchmaking using Ranked Instance Retrieval. In *SMR '06: Proceedings of the 1st International Workshop on Semantic Matchmaking and Resource Retrieval, Co-located with VLDB*, Jgg. 178 von *CEUR Workshop Proceedings*. CEUR-WS.org, 2006.

[CG99] Surajit Chaudhuri und Luis Gravano. Evaluating Top-*k* Selection Queries. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, Seiten 397–410, San Francisco, CA, USA, 1999. Morgan Kaufmann.

[Cho03] Jan Chomicki. Preference Formulas in Relational Queries. *ACM Transactions on Database Systems*, 28(4):427–466, 2003.

[CK97] Michael J. Carey und Donald Kossmann. On Saying "Enough Already!" in SQL. In *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Seiten 219–230, New York, NY, USA, 1997. ACM Press.

[CSP03] Giuseppe Carenini, Jocelyin Smith und David Poole. Towards more Conversational and Collaborative Recommender Systems. In *IUI' 03: Proceedings of the International Conference on Intelligent User Interfaces*, Seiten 12–18, New York, NY, USA, 2003. ACM Press.

[HHP06] Heasoo Hwang, Vagelis Hristidis und Yannis Papakonstantinou. ObjectRank: a System for Authority-based Search on Databases. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, Seiten 796–798, New York, NY, USA, 2006. ACM Press.

[HK05] Bernd Hafenrichter und Werner Kießling. Optimization of Relational Preference Queries. In *ADC '05: Proceedings of the Sixteenth Australasian Database Conference*, Jgg. 39 von *CRPIT*, Seiten 175–184, Sydney, Australia, 2005. Australian Computer Society.

[HKP01] Vagelis Hristidis, Nick Koudas und Yannis Papakonstantinou. PREFER: a System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Seiten 259–270, New York, NY, USA, 2001. ACM Press.

[IA05] Ihab F. Ilyas und Walid G. Aref. Rank-Aware Query Processing and Optimization. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, Seite 1144. IEEE Computer Society, 2005.

[IAE04] Ihab F. Ilyas, Walid G. Aref und Ahmed K. Elmagarmid. Supporting Top-k Join Queries in Relational Databases. *VLDB Journal*, 13(3):207–221, 2004.

[ISA+04] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter und Ahmed K. Elmagarmid. Rank-aware Query Optimization. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, Seiten 203–214, New York, NY, USA, 2004. ACM Press.

[Kie02] Werner Kießling. Foundations of Preferences in Database Systems. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, Seiten 311–322, San Francisco, CA, USA, 2002. Morgan Kaufmann.

[Kie05] Werner Kießling. Preference Queries with SV-Semantics. In *COMAD '05: Advances in Data Management 2005, Proceedings of the Eleventh International Conference on Management of Data*, Seiten 15–26. Computer Society of India, 2005.

[KK02] Werner Kießling und Gerhard Köstler. Preference SQL - Design, Implementation, Experiences. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, Seiten 990–1001, San Francisco, CA, USA, 2002. Morgan Kaufmann.

[LCIS05] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas und Sumin Song. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, Seiten 131–142, New York, NY, USA, 2005. ACM Press.

[LL87] M. Lacroix und Pierre Lavency. Preferences; Putting More Knowledge into Queries. In *VLDB '87: Proceedings of the 13th International Conference on Very Large Data Bases*, Seiten 217–225, San Francisco, CA, USA, 1987. Morgan Kaufmann.

[LSY03] Greg Linden, Brent Smith und Jeremy York. Amazon.com Recommendations: Item-to-Item Collaborative Filtering. *IEEE Internet Computing*, 7(1):76–80, 2003.

[MS02] Lorraine McGinty und Barry Smyth. Evaluating Preference-Based Feedback in Recommender Systems. In *AICS*, Jgg. 2464 von *Lecture Notes in Computer Science*, Seiten 209–214. Springer, 2002.

[SEK06] Benjamin Satzger, Markus Endres und Werner Kießling. A Preference-Based Recommender System. In *EC-WEB '06: Proceedings of the 7th International Conference on E-Commerce and Web Technologies*, Jgg. 4082 von *Lecture Notes in Computer Science*, Seiten 31–40. Springer, 2006.

[YYY+03] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia und Yuguo Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE '03: Proceedings of the 19th International Conference on Data Engineering*, Seiten 189–200. IEEE Computer Society, 2003.