Preference SQL - eine Einführung

Timotheus Preisinger, Johannes von Stetten

26. Februar 2007

Inhaltsverzeichnis

1. Preference SQL - hybride Version

Die bisherige Version von Preference SQL wird „hybrider Ansatz“ genannt. Dieses Konzept modelliert Preference SQL als eigenständige Schicht, die an eine beliebige Datenbank mittels JDBC angebunden werden kann.

1.1. Installation PSQL (hybride Version)

Zur Installation der hybriden Version von Preference SQL in eine eigene Anwendung müssen lediglich die beiden JAR-Dateien `psql-base.jar`, `psql-hybrid.jar` und `xxl.jar` [1] in den `classpath` aufgenommen werden. Damit steht die PSQL-Schicht der Anwendung zur Verfügung.

1.2. Ausführung von PSQL-Anfragen

Für die Ausführung einer PSQL-Query gibt es zwei unterschiedliche Methoden, je nachdem, wie viel Kontrolle der Programmierer auf die Ausführung ausüben will. Zum Abarbeiten einer Query sind im Überblick folgende Punkte zu erledigen:

- Laden des JDBC-Treibers der zu verwendenden Datenbank und Herstellen einer Connection
- Laden der SQLEngine mit Angabe der Repository-Datei
- ggf. Laden der SCORE- und RANK-Funktionen
- Instanzieren von TreeBuilder und Parser
- Parsen des Statements
- ggf. Optimieren des Statements
- Auswerten des Statements

Mehrere davon können automatisch von PSQL durchgeführt werden. Das Laden des JDBC-Treibers und die Erstellung einer Verbindung zur Datenbank kann auch erst an späterer Stelle geschehen. Es ist in jedem Fall vom Programmierer durchzuführen.

Für eine MySQL-Datenbank kann dies wie folgt ablaufen:

Listing 1: Listing PSQL hybrid - 3

```
42      Class.forName("com.mysql.jdbc.Driver").newInstance();
43
44      Connection sqlCon = DriverManager.getConnection(
45          "jdbc:mysql://localhost/test?user=aa&password=bb");
```

[1] z. B. auf der CD im Verzeichnis `jars` zu finden

1.3. Einfache Ausführung

Das JDBC-Verbindungsobjekt wird an den Konstruktor der Klasse `preference.sql.PSQLEngine` übergeben.

Listing 2: Initialisierung der PSQL-Engine

```
1 preference.sql.PSQLEngine engine =
2     new preference.sql.PSQLEngine(con);
```

Queries werden dann mit Hilfe der Methode `executeQuery(String psql)` ausgeführt. Als optionaler zweiter Parameter ist die Angabe von Fremd- und Primärschlüsselbeziehungen in einem XML-Format möglich. Eine Beschreibung des XML-Formats dieser Informationen ist in Kapitel 1.7.1 zu finden.

Die XML-Datei kann als

- `org.w3c.dom.Document` oder als
- `java.io.File`

übergeben werden.

Ein PSQL-Statement kann also wie folgt ausgeführt werden:

Listing 3: Ausführung einer Query mit Optimierung

```
1 xxl.core.cursors.MetaDataCursor result =
2     engine.executeQuery(String query,
3         org.w3c.dom.Document optimizerConstraints);
```

Informationen zur Weiterverarbeitung dieses `MetaDataCursor`-Objekts befinden sich in Kapitel 1.5.

1.4. Standard-Ausführung mit erweiterter Kontrolle

Im Anschluss wird ein Code-Beispiel vorgestellt, anhand dessen die Implementierung der nötigen Schritte aufgezeigt wird. In Kapitel 1.6 wird dann darauf eingegangen, wie diese Methode in eigenen Java-Code eingebunden werden kann.

Ein komplettes Listing ist im Anhang A.1 als Listing 11 zu finden. Die hier angegebenen Zeilennummern beziehen sich direkt auf die Zeilennummern des Listings 11.

1.4.1. Laden der SQLEngine mit Angabe Respository-File

Als erster Schritt ist zunächst die SQLEngine zu laden:

Listing 4: Listing PSQL hybrid - 1

```
28    SQLEngine sqlEng = new SQLEngine();
```

1.4.2. ggf. Laden der SCORE- und RANK-Funktionen

Innerhalb einer Präferenzen können SCORE- und RANK-Funktionen verwendet werden (siehe Kapitel 3.1 und 3.3). Dazu müssen diese Funktionen als Klassen implementiert vorliegen (Beispiel siehe Anhang A.3).

Diese Klassen mit SCORE oder RANK-Funktionen können entweder im folgenden registriert werden, oder zur Laufzeit dynamisch ermittelt werden. Der Vorteil des Registrierens ist, dass den registrierten Funktionen einen beliebigen Namen haben können, unter welchem sie dann verwendbar sind, wohingegen beim Lookup nur der Klassenname angegeben werden kann.

Im folgenden Listing werden jeweils eine dieser Funktionen registiert.

Listing 5: Listing PSQL hybrid - 2

```
35    sqlEng.registerScoreFunction(new SimpleScoreFunc());
36    sqlEng.registerCombiningFunction(new SimpleRankFunc());
```

Werden die Klassen nicht registriert, so wird mittels Reflection versucht, eine geeignete Klasse zu ermitteln. Wird eine Präferenz mit SCORE oder RANK gefunden, wird zunächst überprüft, oder der übergebene Funktionsname eine registrierte Funktion ist. Ist dies nicht der Fall, so wird getestet, ob der String einen Klassennamen darstellt (z. B. `preference.myFunctions.PreisFunction`). Zuletzt wird noch geprüft, ob eine Klasse mit dem angegebenen Namen im Package `preference.extensions` zu finden ist.

1.4.3. Instanziieren von TreeBuilder und Parser

Anschließend sind die nötigen Klassen für das Parsen des Statements zu instanziieren. Dies geschieht mittels folgendem Code:

Listing 6: Listing PSQL hybrid - 4

```
50    PreferenceSQLParser prefParser = new PreferenceSQLParser();
```

```
51
52      SQLTreeBuilder sqlBuilder = new SQLTreeBuilder(sqlEng,
53          sqlCon);
54
55      prefParser.setSQLListener(sqlBuilder);
56
57      // re-init and reset the parser
58      prefParser.ReInit(new StringReader(query));
59      prefParser.reset();
```

Der `SQLTreeBuilder` wird dem Parser übergeben und setzt die geparsten Ausdrücke zu einem SQL-Statement zusammen. Der eigentliche Parser wird in Zeile 50 als `prefParser` instanziiert. In Zeile 58 wird die eigentlich Query dem Parser übergeben.

Zu beachten ist, dass der Parser mehrfach verwendet werden kann, d.h. es können hintereinander beliebig viele Statements geparst werden. Einzige Voraussetzung ist, dass vor jedem Durchlauf die Parser-Methoden `ReInit()` und `reset()` aufgerufen werden.

1.4.4. Parsen des Statements

Nun wird der Parser gestartet und die Query geparst, sowie aus dem Ergebnis ein `QueryTree` aufgebaut:

Listing 7: Listing PSQL hybrid - 5

```
64      prefParser.SQLSelect();
65
66      RelAOp opRoot = sqlBuilder.getQueryTree();
```

1.4.5. ggf. Optimieren des Statements

Falls der Optimierer verwendet werden soll, wird dieser nun gestartet. Die zu verwendeten Optimierungs-Einstellungen werden dabei entweder als bereits geparstes XML-DOM-`Document` übergeben oder als Dateiname.

Für die Übergabe eines bereits geparstens Dokuments bietet die Klasse `OptimizerConfig` im Paket `preference.sql.optimize` einige Hilfsmethoden:

- `getDefaultConfig()`: Eine Standard-Konfiguration des Optimierers wird zurückgeliefert.

- `getOptimizerConfig(String fileName)`: Liest die Konfiguration aus einer Datei. Wird die Datei nicht gefunden (oder treten Parser-Probleme auf), wird eine `PreferenceException` geworfen.

- `getOptimizerConfig(org.xml.sax.InputSource src)`: Liest die Konfiguration aus einer beliebigen SAX-Quelle. Auch hier wird bei Problemen eine `PreferenceException` geworfen.

- `getOptimizerConfig()`: Versucht die Konfiguration aus einer Datei zu lesen, deren Name als Umgebungsvariable `psql.optimizer.configfile` angegeben ist. Schlägt das fehl (z. B. weil die Umgebungsvariable nicht definiert ist), wird die Konfiguration aus der Datei `psql-optimizer.xml` gelesen. Treten auch dabei Probleme auf, wird die Standard-Konfiguration verwendet.
  Die Methode wirft keine Exception.

Beim direkten Aufruf des Optimierers mit dem Dateinamen wie auch beim Umweg über `OptimizerConfig` bezieht sich der Dateiname auf die XML-Datei, in der die anzuwendenen Regeln spezifiziert sind. Auf der CD liegen im Verzeichnis `./optimizer/` einige Regeldateien bereit.

Listing 8: Listing PSQL hybrid - 6

```
69      if (useOptimizer)
70        opRoot = runOptimizer(<DATEINAME>, sqlCon, opRoot);
```

Die Methode `runOptimizer()` lautet dabei wie folgt:

Listing 9: Listing PSQL hybrid - runOptimizer

```
106   private static RelAOpI runOptimizer(String optimizer,
107       Connection sqlCon, RelAOpI opRoot)
108       throws PreferenceException {
109     PrefSQLOptimicer opPref = new PrefSQLOptimicer();
110     /* entweder ... */
111     opPref.initOptimicer(optimizer);    /* ... oder ... */
112     opPref.initOptimicer(OptimizerConfig.getDefaultConfig(
113         optimizer));
114     opRoot = opPref.optimizeQuery(opRoot,
115         "../PreferenceSQL/optimizer/constraint.xml", sqlCon);
116     return opRoot;
117   }
```

Hier wird zunächst der Optimierer mit dem übergebenen Dateinamen der Optimizer-XML-Datei initialisiert (siehe Kapitel 1.7). Anschließend wird der Optimierer gestartet, wobei der zu optimierende Baum übergeben wird (`opRoot`), sowie das zu verwendende Constraint-File (siehe Kapitel 1.7).

1.5. Auswerten des Statements

Das eigentliche Auswerten der Query findet während der Instantiierung der Klasse `MetaDataCursor` in Zeile 72 sowie des nachfolgenden Durchschalten des Ergebnis-Cursors ab Zeile 76 statt:

Listing 10: Listing PSQL hybrid - 7

```
72    MetaDataCursor cursor = opRoot.getQuery(null);
73
74    ArrayList results = new ArrayList();
75
76    while (cursor.hasNext()) {
77      Tuple tup = (Tuple) cursor.next();
78      results.add(tup);
79    }
80    cursor.close();
81
82    return results;
```

Jeder Aufruf der Methode `next()` des `MetaDataCursor`-Objekts liefert genau ein Ergebnis der Query als Objekt vom Typ `Tuple` zurück. Dieser Typ besitzt die folgenden Zugriffsmethoden, die in ähnlicher Form auch in der Klasse `java.sql.ResultSet` vorhanden sind:

- `java.sql.ResultSetMetaData getMetaData()`: gibt ein Meta-Daten-Objekt zurück
- `int getColumnCount()`: gibt die Anzahl der Spalten des Tupels zurück
- `boolean getBoolean(int column)`: entspricht `getBoolean` in `ResultSet`.
- `byte getByte(int column)`: entspricht `getByte` in `ResultSet`.
- `java.sql.Date getDate(int column)`: entspricht `getDate` in `ResultSet`.
- `double getDouble(int column)`: entspricht `getDouble` in `ResultSet`.
- `float getFloat(int column)`: entspricht `getFloat` in `ResultSet`.
- `int getInt(int column)`: entspricht `getInt` in `ResultSet`.
- `Object getObject(int column)`: entspricht `getObject` in `ResultSet`.
- `short getShort(int column)`: entspricht `getShort` in `ResultSet`.

- `String getString(int column)`: entspricht `getString` in `ResultSet`.
- `java.sql.Time getTime(int column)`: entspricht `getTime` in `ResultSet`.
- `java.sql.Timestamp getTimestamp(int column)`: entspricht `getTimestamp` in `ResultSet`.
- `boolean isNull(int column)`: vergleicht den Inhalt der Spalte mit `null`.
- `Object[] toArray()`: kopiert die Spaltenwerte in ein Array

Die Spaltenwerte lassen sich ebenfalls mit ihren Namen abrufen. Dabei müssen jedoch voll qualifizierte Namen verwendet werden, d.h. der Tabellenname gefolgt vom Spaltenname, wobei ein Punkt die Bezeichner trennt. Der voll qualifizierende Name einer Spalte `AGE` in einer Tabelle `CARS` ist `CARS.AGE`.

Das in Anhang A.1 dargestellte Minimalbeispiel 11 stellt beispielhaft dar, wie aus dem zurückgelieferten `Tuple` (genauer gesagt einer Unterklasse davon, `ArrayTuple`) die einzelnen Ergebniswerte extrahiert werden können. Dazu wird die Methode `getUsedCarsFromResults()` verwendet, die aus dem `ResultSet` Objekte des Types `UsedCars` erzeugt.

1.6. Verwendung in eigenem Java-Code

Zur Verwendung in einem eigenen Java-Programm kann also beispielsweise die in Listing 11 beschriebene Methode `ArrayList executeQuery(String, String, boolean)` implementiert werden.

Hierbei sind lediglich noch Anpassungen bezüglich Repository-File (Kapitel 1.4.1), verwendeter SCORE- und RANK-Funktionen (Kapitel 1.4.2) sowie JDBC-Daten der zu verwendenden Datenbank (Kapitel 1.2) vorzunehmen.

Anschließend kann der Methode die betreffende Query als `String` übergeben werden, und als Ergebnis eine `ArrayList` von `ArrayTuple`-Objekten entgegengenommen werden. Diese können dann durch Code analog zu dem in Listing 11 präsentierten Beispiel ausgewertet beziehungsweise in die benötigten Datentypen konvertiert werden.

Selbstverständlich kann der hier skizzierte Code auch direkt in der eigenen Anwendung implementiert werden, solange die grundlegende Struktur und Reihenfolge beibehalten wird.

1.7. Besonderheiten in der hybriden Version

1.7.1. Metadaten

In der hybriden Version kann die Preference Engine, bedingt durch das Design „on top of the database“, nicht per JDBC auf alle Metadaten der unterliegenden Datenbank zugreifen.

Daher müssen *Fremdschlüssel*- und *Primärschlüssel*-Beziehungen der Präferenz-Schicht gesondert bekannt gemacht werden. Dies geschieht mit Hilfe eines XML-Files („Constraint-File“), in dem diese dort in folgendem Format vermerkt werden (komplettes Listing siehe Anhang A.4):

```
<DatabaseMetaData>
  <ForeignKey>
    <reference table = "TABLE1"    column = "COL1"
            reftable = "TABLE2" refcolumn = "REFCOL"/>
    ...
  </ForeignKey>

  <PrimaryKey>
    <key table = "TABLE1" column = "KEYCOL"/>
    ...
  </PrimaryKey>
</DatabaseMetaData>
```

In diesem Beispiel existiert eine Fremdschlüssel-Beziehung von der Spalte `TABLE1.COL1` auf die Spalte `TABLE2.REFCOL`, und im `TABLE1` ist die Spalte `KEYCOL` der Primärschlüssel.

Dieses Constraint-File muss nun an den Optimizer bei Beginn der Optimierung übergeben werden (siehe Listing Zeile 115).

1.7.2. GROUP BY

In PSQL-Hybrid ist `GROUP BY` ein Präferenz-Keyword. Das bedeutet, dass das gleichnamige SQL-Keyword in PSQL-Hybrid nicht verwendet werden kann!

`GROUP BY` in Preference SQL kennzeichnet eine gruppierte Präferenz. Hierbei wird die Tupelmenge nach dem betreffenden Attribut gruppiert und anschließend eigentliche Präferenz auf jeder einzelnen Gruppe evaluiert.

Beispiel:

```
SELECT * FROM used_cars PREFERRING price LOWEST GROUP BY color
```

Diese Präferenz würde die verfügbaren Autos zunächst nach Farbe gruppieren, und auf jede dieser Gruppen die Präferenz `price LOWEST` auswerten. Als Ergebnis würde also für jede Farbe das billigste Auto zurückgeliefert.

1.7.3. Debuglevel

Zur Steuerung des Detailgrads der Systemausgaben wurde ein zentraler Mechanismus implementiert, mit dessen Hilfe sich der Detailgrad in bestimmten Schichten von „sehr detailliert“(Level 4) bis „überhaupt keine Ausgaben“ (Level 0) regeln lässt. Je höher der eingestellte Level, desto höher ist der Detailgrad der ausgegebenen Meldungen.

Dieser sog. Debuglevel lässt sich entweder statisch in der Klasse Klasse `DebugLevel` im Package `preference.engine` setzen, oder mittels einer Konfigurationsdatei einstellen (siehe Anhang A.6).

Diese Konfigurationsdatei muss beim Start des Programms geladen werden, z. B. mittels folgendem Code:

```
1 DebugLevel.init("./psql.conf");
```

Bei Verwendung der Methode ohne Parameter wird standartmäßig die Konfigurationsdatei in „`./psql.conf`“ ausgehend vom Systempfad gesucht. Wird die Datei nicht gefunden, so wird versucht, die System-Property „`debuglevel`“ zu verwenden. Ist diese ebenfalls nicht vorhanden, wird der Standardwert, Level 0, verwendet.

2. Preference SQL - in McKoi integrierte Version

Für die Verwendung der neuen, in McKoi integrierten PSQL-Version sind folgende Schritte nötig:

- Installation PSQL-McKoi
- Einrichten Datenbank
- Füllen Datenbank

2.1. Installation PSQL-McKoi

Zur Installation der McKoi-Version von Preference SQL sind folgende Schritte nötig:

Zum Betrieb eines McKoiDB-Servers, bzw. zum Erstellen und Einrichten der Datenbank: Zunächst sind die JAR-Dateien `mckoidb.jar`, `psql-hybrid.jar`, `psql-base.jar` und `xxl.jar` in den `classpath` aufzunehmen.

Für das Anlegen einer neuen Datenbank wird eine Konfigurationsdatei benötigt, wie sie zum Beispiel in der Datei `db.conf` vorstellt ist (siehe Anhang A.5).

Falls durch die Anwendung nur ein bereits laufender McKoiDB-Server angesprochen werden soll, ist lediglich die JAR-Datei `mkjdbc.jar` nötig.

Nach erfolgter Installation sind keine weiteren Schritte durch den Benutzer erforderlich. Die McKoi-Datenbank kann dann auf herkömmliche Weise als JDBC-Datenbank in die Anwendung eingebunden und angesprochen werden.

2.2. Anlegen einer neuen Datenbank

Zum Erstellen einer neuen Datenbank wird eine so genannte Konfigurationsdatei benötigt. Eine solche ist beispielsweise in der Datei `./configs/db.conf` zu finden. Anhand dieser wird mittels der Klasse `com.mckoi.runtime.McKoiDBMain` eine neue Datenbank angelegt (der Speicherort dieser Datenbank ist in der Konfigurationsdatei vermerkt). Es wurde eine Beispielsklasse `com.mckoi.preferences.samples.CreateDefaultDB` implementiert, die das Anlegen einer neuen Datenbank zeigt.

```
java com.mckoi.runtime.McKoiDBMain -create "user" "pass"
                                   -conf ./configs/db.conf
  bzw.
java -jar mckoidb.jar  -create ... -conf ...
```

2.3. Starten und Verwenden der Datenbank

Die McKoi-Datenbank kann entweder im *Server-Mode* oder im *Embedded-Mode* laufen.

2.3.1. Server-Mode

Im Server-Mode wird die Datenbank als eigener Prozess gestartet, und kann beliebig viele `Connections` gleichzeitig verarbeiten. In diesem Fall wird die Datenbank mittels der Klasse `com.mckoi.runtime.McKoiDBMain` gestartet.

Als Argument muss hierzu die Konfigurationsdatei (siehe Kapitel 2.2) angegeben werden, zum Beispiel `-conf ./configs/db.conf`:

```
java com.mckoi.runtime.McKoiDBMain -conf ./configs/db.conf
  bzw.
java -jar mckoidb.jar -conf ./configs/db.conf
```

Angesprochen wird sie dann beispielsweise durch den im Anhang A.2 in Listing 12 vorgestellten Code. Der hier vorgestellte Code ist in der Klasse `ConnectToServerDemo` im Package `com.mckoi.preferences.samples` zu finden.

Hierbei wird in der Anwendung lediglich der JDBC-Treiber `mkjdbc.jar` benötigt.

2.3.2. Embedded-Mode

Im Embedded-Mode hingegegen muss die Datenbank nicht als eigener Prozess gestartet werden, sie kann direkt in die Anwendung integriert werden. Sie wird also direkt über die Anwendung gestartet, wobei im JDBC-Treiber die Configurationsdatei der Datenbank angegeben werden muss.

Dies ist besonders praktisch, da in diesem Fall die Datenbank nicht installiert werden muss, sondern lediglich die JAR-Files in das eigene Projekt eingebunden werden müssen.

Beispielsweise zeigt die Klasse `com.mckoi.preferences.samples.EmbeddedDatabaseDemo` eine solche Verwendung. In diesem Fall kann die Datenbank jedoch immer nur eine `Connection` annehmen!

Im Gegensatz zum Server-Mode ist hierbei lediglich die URL der JDBC-Verbindung anzupassen, wie im Anhang A.2 in Listing 13 dargestellt.

Im Embedded-Mode werden die folgenden JAR-Files benötigt:
`mckoidb.jar`, `psql-hybrid.jar` und `psql-base.jar`.

2.4. Beispieldaten

Die Klasse `com.mckoi.preferences.samples.SetupEmbeddedDB` demonstriert das Einrichten der Beispieldatenbank mit einigen Tabellen und Daten. Hier werden die nötigen Daten für die JUnit-Tests angelegt, sowie Tabellen mit Zufallsdaten (Tabellen `myTable` und `used_cars`).

2.5. Besonderheiten in der McKoi-Version

2.5.1. Metadaten

Im Gegensatz zur hybriden Version müssen hier die *Fremdschlüssel-* und *Primärschlüssel*-Beziehungen nicht gesondert angegeben werden. Die in McKoi integrierte Version ermittelt diese direkt aus der McKoi-Datenbank.

2.5.2. GROUP BY

Das Präferenz-Keyword `GROUP BY` (siehe auch Kapitel 1.7) wurde für PSQL-McKoi umbenannt, so dass das gleichnamige SQL-Keyword wieder verwendet werden konnte.

In PSQL-McKoi lautet das Keyword für die Präferenz-Gruppierung nun `GROUPING`. Das Keyword für die SQL-Gruppierung lautet wie in Standart-SQL `GROUP BY`.

Beispiel für GROUPING:

```
SELECT * FROM used_cars PREFERRING price LOWEST GROUPING color
```

(Erklärung analog zu Kapitel 1.7)

Beispiel für GROUP BY:

```
SELECT name, SUM(umsatz)
  FROM kunde
  GROUP BY name
```

Hier wird pro Kunde eine Summe der Umsätze zurückgegeben, indem die Daten der Tabelle `kunde` zunächst nach `name` gruppiert werden. Ohne die Gruppierung würden sämtliche Umsätze der Tabelle `kunde` aufsummiert, ohne Rücksicht auf den jeweiligen Namen.

Beide Begriffe kennzeichnen also eine Gruppierung, wobei `GROUPING` innerhalb der Präferenz gruppiert, und `GROUP BY` außerhalb. Dabei können auch problemlos beide Keywords gleichzeitig verwendet werden.

2.5.3. Debuglevel

Wie schon in Kapitel 1.7.3 vorgestellt, lässt sich der Detailgrad der Systemausgaben mit dem Debuglevel steuern.

In McKoi wird automatisch beim Initialisieren der Präferenz-Engine versucht, eine Konfigurationsdatei zu laden. Diese wird standardmäßig im Pfad „`../PreferenceSQL/psql.conf`“ gesucht[2].

Soll eine Konfigurationsdatei aus einem anderen Pfad verwendet werden, so muss analog der in Kapitel 1.7.3 dargestellten Vorgehensweise verfahren werden.

[2] In der Methode `com.mckoi.preferences.Preferences.initPreferenceParser()` lässt sich dieser Pfad anpassen.

3. Präferenz-Konstruktoren

In den verschiedenen Versionen von Preference SQL können zahlreiche vordefinierte Präferenz-Konstruktoren verwendet werden, mit denen sich die meisten Präferenzen darstellen lassen.

Im Folgenden wird noch einmal die verschiedenen möglichen Präferenztypen vorgestellt, wie sie in der hybriden Version von Preference SQL sowie in der McKoi-Version verwendet werden können.

Allgemein ist eine simple Query wie folgt aufgebaut:

```
SELECT <columns> FROM <tables>
       WHERE <hardconstraint>
       PREFERRING <preference>
```

Die Präferenzkonstrukte lassen sich in Preference SQL mit folgender Syntax verwenden. Dabei wird zunächst die herkömmliche Syntax (ohne d-Parameter und SV-Relation etc.) vorgestellt, anschließend eventuelle Alternativen. Zum Schluss folgen jeweils noch einige Beispiele (In diesen Beispielen wird auch gleich die Syntax für die Konstrukte d-Parameter und SV-Relation dargestellt, für eine Erklärung dieser Konstrukte siehe Kapitel 3.4).

Die in der Syntaxbeschreibung vorkommenden Platzhalter haben folgende Bedeutung: `<column>` steht für eine Spaltenbezeichnung, also das Attribut der Präferenz, `<number>` für einen beliebigen numerischen Wert. `<string_literal>` kennzeichnet einen in (einfachen oder doppelten) Anführungszeichen eingeschlossenen String und `<preference>` eine Teil-Präferenz (welche sowohl eine Basispräferenz als auch eine komplexe Präferenz sein kann).

Ein Ausdruck, der in eckigen Klammern mit anschließendem Fragezeichen eingefasst ist, z. B. `[<ausdruck>]?`, ist optional, kann also auch weggelassen werden. Dagegen kommt ein Ausdruck in Klammern mit anschließendem Stern, z. B. `[<ausdruck>]*`, keinmal oder beliebig oft vor. Das Zeichen `|` kennzeichnet eine Alternative, es ist also entweder die linke oder die rechte Seite möglich.

3.1. Numerische Basispräferenzen

BETWEEN(A, [low,up])
Die BETWEEN-Präferenz gibt an, dass der gesuchte Wert des Attributs möglichst in einem bestimmten Bereich liegen soll.

Wird mindestens ein Tupel innerhalb dieses Bereiches gefunden, so sind diese Tupel besser als die restlichen. Werden keine solchen Tupel gefunden, so sind diejenigen Werte besser, die möglichst nahe an dem Bereich $[low, up]$ liegen. Haben zwei Tupel den gleichen Abstand, sind jedoch nicht identisch, so sind sie unvergleichbar.

```
PSQL-Syntax : <column> BETWEEN <number> AND <number>
Alternativen: <column> UP TO <number>
              <column> MORE THAN <number>

Beispiele   : alter BETWEEN 10 AND 15
              preis BETWEEN 5000 AND 6000 , 100 REGULAR
```

Die Alternative MORE THAN bedeutet, dass ein Wert zwischen der angegebenen Zahl und $+\infty$ gesucht wird, bei UP TO zwischen $-\infty$ und der angegebenen Zahl.

AROUND(A, z)
Die AROUND-Präferenz ist ein Spezialfall der BETWEEN-Präferenz. Mit ihr definiert man den Wunsch, dass der gesuchte Attributwert möglichst nahe an dem angegeben Wert liegen soll.

Werden ein oder mehrere Tupel gefunden, die in A genau den Wert z haben (optimale Tupel), so sind diese Tupel besser als alle anderen. Falls kein solches Tupel gefunden wird, so ist ein Tupel mit geringerem Abstand von A zu z besser als ein Tupel mit höherem Abstand. Haben zwei Tupel den gleichen Abstand, sind jedoch nicht identisch, so sind sie unvergleichbar.

```
PSQL-Syntax : <column> AROUND <number>
Alternativen: -

Beispiele   : leistung AROUND 90
              verbrauch AROUND 6.5 , 0.5 REGULAR
```

HIGHEST(A)
Die extremale Präferenze HIGHEST gibt an, dass man den Wert des Attributs A maximieren möchte.

Dementsprechend ist bei HIGHEST ein größerer Wert immer besser als ein kleinerer.

```
PSQL-Syntax : <column> HIGHEST
Alternativen: -

Beispiele   : leistung HIGHEST
              preis HIGHEST 20000 , 100 REGULAR
```

LOWEST(A)
Die extremale Präferenze LOWEST gibt an, dass man den Wert des Attributs A minimieren möchte.

Dementsprechend ist bei LOWEST ein kleinerer Wert immer besser als ein größerer.

```
PSQL-Syntax : <column> LOWEST
Alternativen: -

Beispiele   : preis LOWEST
              verbrauch LOWEST 3.0 , 0.5 REGULAR
```

SCORE(A, f)
In der SCORE-Präferenz werden Tupel anhand des Ergebniswerts einer so genannten Score-Funktion $f : dom(A) \rightarrow \mathbb{R}$ verglichen.

Je größer der Wert der Funktion („score“) für ein Attribut ist, desto besser ist das betreffende Tupel bezüglich dieser Präferenz.

```
PSQL-Syntax : <column> SCORE <string_literal>
Alternativen: -

Beispiele   : alter SCORE 'alterFunction'
              preis SCORE 'preisFunction' , 5 REGULAR
```

Das Registrieren von SCORE-Funktionen wird in Kapitel 1.4.2 erläutert.

3.2. Kategorielle Basispräferenzen

POS(A, POS-set)
Die POS-Präferenz, oder auch Positivpräferenz, drückt die Bevorzugung von bestimmten Werten gegenüber den restlichen Werten in $dom(A)$ aus.

Die verschiedenen Werte innerhalb der Menge POS-set sind zueinander unvergleichbar und jeweils besser als jeder Wert außerhalb von POS-set. Diese Werte außerhalb sind wiederum jeweils unvergleichbar.

```
PSQL-Syntax : <column> IN ( <string_literal>
                            [, <string_literal>]* )
Alternativen: <column> = <string_literal>

Beispiele   : farbe IN ('gelb','grün','blau')
```

```
                farbe IN ('weiß','schwarz') REGULAR
                modell = 'Kombi'
                modell = 'Van' REGULAR
```

POS/POS(A, POS_1-set; POS_2-set)
Durch die POS/POS-Präferenz kann im Vergleich zur Positivpräferenz zusätzlich noch eine alternative zweite Menge mit ebenfalls bevorzugten Werten angegeben werden.

Die Werte aus dem POS_2-set sind besser als der Rest aus $dom(A)$, aber schlechter als die Werte des POS_1-set. Innerhalb einer Menge sind die Werte wie schon in der POS Präferenz unvergleichbar.

```
PSQL-Syntax : <column> IN ( <string_literal>
                            [, <string_literal>]* )
                      ELSE ( <string_literal>
                            [, <string_literal>]* )
Alternativen: -

Beispiele   : modell IN ('Kombi','Limousine')
                  ELSE ('SUV', 'Van')
              farbe IN ('blau', 'gelb')
                  ELSE ('rot', 'grün') REGULAR
```

NEG(A, NEG-set)
Mit der Negativpräferenz (NEG-Präferenz) wird die Abneigung gegenüber einer bestimmten Menge von Werten ausgedrückt.

Die Negativpräferenz ist das Gegenteil der Positivpräferenz. Hier sind alle Werte aus der Menge NEG-set schlechter als alle restlichen Werte aus $dom(A)$. Die Werte innerhalb NEG-set sind wiederum jeweils unvergleichbar zueinander, ebenso die restlichen Werte.

```
PSQL-Syntax : <column> NOT IN ( <string_literal>
                              [, <string_literal>]* )
Alternativen: <column> != <string_literal>

Beispiele   : modell NOT IN ('Kleinwagen','LKW')
              modell NOT IN ('PickUp','Mini') REGULAR
              farbe != 'rosa'
              farbe != 'pink' REGULAR
```

POS/NEG(A, POS-set; NEG-set)
Als vierte Variante existiert die POS/NEG-Präferenz. Sie ist eine Kombination der Positiv- und der Negativpräferenz.

Hier sind die Werte im POS-set bevorzugt, und gleichzeitig ist der Rest aus $dom(A)$ besser als alle Werte aus dem NEG-set. Die Werte der einzelnen Mengen sind wiederum untereinander unvergleichbar.

```
PSQL-Syntax : <column> IN ( <string_literal>
                            [, <string_literal>]* )
                  NOT IN ( <string_literal>
                            [, <string_literal>]* )
Alternativen: -

Beispiele   : farbe IN ('schwarz', 'rot', 'gold')
                  NOT IN ('pink', 'rosa', 'lila')
```

LAYERED$_m$(A,L)
Die LAYERED-Präferenz ist eine Verallgemeinerung der Positiv- und Negativ-Präferenzen. Hier werden beliebige Schichten („layers“) mit Werten aus $dom(A)$ definiert. Jeder Attributwert tritt dabei nur in genau einer Schicht auf.

Aus der Sortierung der Schichten ergibt sich die Bewertung der Tupel. Ein Wert aus einer höheren Schicht ist immer besser als ein Wert aus einer niedrigeren Schicht (wobei die höchste Schicht die zuerst angegebene ist).

```
PSQL-Syntax : <column> LAYERED (
                  ( <string_literal> [, <string_literal>]* )
                    | OTHERS
                [, ( <string_literal> [, <string_literal>]* )
                    | , OTHERS ]*  )
Alternativen: -

Beispiele   : farbe LAYERED ( ('grün','blau','gelb'),
                  ('weiß','schwarz','grün'), OTHERS,
                  ('lila','rosa')  )
```

EXP(A, E-graph)
In einer expliziten Präferenz werden einzelne Werte aus $dom(A)$ direkt zueinander in Beziehung gesetzt.

Alle anderen, nicht explizit festgelegten Werte sind zueinander unvergleichbar. Als Ausnahme hierzu sind auch Wertepaare miteinander vergleichbar, die nicht explizit angegeben wurden, falls diese durch die Transitivität zueinander in Relation stehen.

```
PSQL-Syntax : <column> EXPLICIT (
                    <string_literal> < <string_literal>
                  [, <string_literal> < <string_literal> ]* )
```

```
Alternativen: -

Beispiele   : modell EXPLICIT ('LKW' < 'Van',
                  'Van' < 'Kombi', 'Van' < 'Limousine',
                  'Kombi' < 'Limousine')
```

ANTICHAIN(A, $\oslash$)
Die ANTICHAIN-Präferenz (auch: $A^{\leftrightarrow}$) ist ein Spezialfall einer Präferenz und das Gegenteil der oben erwähnten „chain". In einer „antichain" sind sämtliche Werte aus $dom(A)$ zueinander unvergleichbar oder identisch.

Hierdurch kann zum Beispiel in Kombination mit einer komplexen Präferenz eine Gruppierung nach den einzelnen Werten aus A erreicht werden.

```
PSQL-Syntax : <column> ANTICHAIN
Alternativen: -

Beispiele   : preis ANTICHAIN
              alter ANTICHAIN REGULAR
```

3.3. Komplexe Präferenzen

PARETO: $P_1 \otimes P_2$
Unter einer PARETO-Präferenz versteht man die gleichwertige Verknüpfung von verschiedenen Bedingungen. Ein Tupel t_1 dominiert hierbei ein Tupel t_2, wenn t_1 mindestens in einer enthaltenen Präferenz besser ist, und in den anderen enthaltenen Präferenzen mindestens gleich gut ist. Wenn weder t_1 noch t_2 besser als das jeweils andere ist, sind die beiden unvergleichbar.

```
PSQL-Syntax : <preference> AND <preference>
Alternativen: -

Beispiele   : preis around 5000 AND farbe = 'blau'
```

PRIORISIERUNG: $P_1 \& P_2$
In einer PRIORISIERUNG werden die unterschiedlichen enthaltenen Präferenzen verschieden gewichtet. P_1 ist hier wichtiger als P_2. Die Präferenz P_2 wird nur beachtet, wenn die Werte in P_1 gleich oder ersetzbar sind. In der Priorisierung ist die Reihenfolge also von Bedeutung.

Ein Tupel t_1 ist in einer Priorisierung P besser als ein Tupel t_2, wenn t_1 in P_1 besser als t_2 ist, oder in P_1 gleich oder ersetzbar und in P_2 besser.

```
PSQL-Syntax : <preference> PRIOR TO <preference>
Alternativen: -

Beispiele   : modell = 'Kombi' PRIOR TO alter lowest
```

RANK: rank$_F$(P_1, P_2
In einer RANK-Präferenz werden die Score-Werte zweier oder mehrerer SCORE-Präferenzen kombiniert, indem aus diesen Werten mittels einer Funktion $F : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ ein Rank-Wert ermittelt wird.

```
PSQL-Syntax : ( <f_preference> [, <f_preference> ]* )
                RANK <string_literal>
Alternativen: -

Beispiele   : ( alter SCORE 'alter_func' ,
                preis SCORE 'preis_func' ,
              ) RANK 'alter_preis_func'

              ( leistung SCORE 'f1' ,
                ( alter SCORE 'f2' , preis SCORE 'f3')
                  RANK 'r1' ,
                id SCORE 'f4'
              ) RANK 'r2'
```

Dabei kann `<f_preference>` sowohl eine SCORE-Präferenz, als auch eine RANK-Präferenz sein, ansonsten sind jedoch keine anderen Präferenztypen erlaubt.

Das Registrieren von RANK-Funktionen wird in Kapitel 1.4.2 erläutert.

3.4. weitere Konstruktoren

Die einzelnen Basispräferenzen können zusätzlich noch um die Konstrukte d-Parameter und SV-Relation erweitert werden, welche hier ansatzweise beschrieben sind. Für eine genauere Erklärung sei auf die betreffende Literatur verwiesen.

3.4.1. d-Parameter

Durch den d-Parameter wird für numerische Präferenzen der Ergebnisraum partitioniert. Innerhalb dieser Partitionen werden dann alle Werte als gleich gut bezüglich der Präferenz angesehen. Ist beispielsweise der Preis eines Produkts 1000 €, so kann eine Abweichung von ± 100 € für den Benutzer unerheblich sein. Das optimale Ergebnis bezüglich der Präferenz ist also der Wert 1000 €. Die erste Partition wäre danach 900 - 999 € sowie 1001 - 1100 €, innerhalb derer sämtliche Werte als gleich gut angesehen würden. Die nächste Partition wäre dann analog 800 - 899 € sowie 1101 - 1200 € und so weiter.

Der d-Parameter kann bei sämtlichen numerischen Basispräferenzen verwendet werden. Er wird der Präferenz nach einem Komma angehängt.

Zum Beispiel eine Partionierung in 100er-Schritten bei BETWEEN:

```
preis BETWEEN 5000 AND 6000, 100
```

Als Spezialfall muss bei AROUND noch ein Sumpremum bzw. Infimum angegeben werden. Supremum von A. Dieser höchste bzw. niedrigste auftretende Wert wird benötigt, um einen Ausgangspunkt für die Partitionierung des Wertebereichs zu erhalten.

Folgende Präferenz partitioniert in 100er-Schritte, wobei 20000 der höchste Wert ist, den `preis` einnehmen kann.

```
preis HIGHEST 20000 , 100
```

3.4.2. SV-Relation

Mit der SV-Relation lässt sich die Ersetzbarkeit von atomaren Werten in Basispräferenzen definieren. Im Normalfall sind alle Attributwerte, die bezüglich einer Präferenz als gleich gut gelten, *unvergleichbar* zueinander. Durch Definition einer entsprechenden SV-Relation lassen sich solche unvergleichbaren Werte als ersetzbar („substitutable“) definieren. Die restlichen Werte sind dann weiterhin unvergleichbar (auch: „alternativ“). Es ist also der natürliche Zustand möglich, dass einige Tupel ersetzbar werden, während andere unvergleichbar bleiben.

In PSQL existieren momentan drei Arten von SV-Relationen: Die *triviale*, in der nur identische Werte ersetzbar sind, die *reguläre*, in der sämtliche unvergleichbaren Werte ersetzbar sind, sowie die *explizite*, in der die ersetzbaren Werte selber definiert werden können.

Ist bei einer Präferenz keine SV-Relation angegeben, so gilt die triviale SV-Relation.

Die reguläre SV-Relation wird durch das Schlüsselwort `REGULAR` angezeigt. Im Beispiel sind die Autos mit den Farben rot und blau untereinander ersetzbar.

```
farbe IN ('rot', 'blau') REGULAR
```

Die explizite SV-Relation verwendet das Keyword `SV`, gefolgt von der Definition der ersetzbaren Werte:

```
farbe IN ('rot', 'blau', 'gruen', 'gelb')
         SV ( ('blau','gelb'), ('gruen','rot') )
```

In diesem Beispiel sind die Farben blau und gelb untereinander ersetzbar, ebenso die Farben grün und rot. Nicht ersetzbar sind jedoch z. B. die Farben blau und rot, usw.

3.4.3. GROUPING

Wie in Kapitel 1.7 und 2.5 bereits vorgestellt, lassen sich Präferenzen gruppiert nach einem bestimmten Attribut auswerten. Die geschieht mittels des Keywords GROUPING, gefolgt von dem Attribut, nach dem Gruppiert werden soll.

Beispiel:

```
SELECT * FROM used_cars PREFERRING price LOWEST GROUPING color
```

Diese Präferenz würde die verfügbaren Autos zunächst nach Farbe gruppieren, und auf jede dieser Gruppen die Präferenz `price LOWEST` auswerten. Als Ergebnis würde also für jede Farbe das billigste Auto zurückgeliefert.

3.5. Beispiel-Präferenz

Als Beispiel betrachten wir nocheinmal folgende komplexe Präferenz:

```
SELECT * FROM used_cars
  PREFERRING color IN ('red', 'green')
    PRIOR TO (price lowest AND age around 10)
```

Hierbei werden zunächst Autos gesucht, die rot oder grün sind (`color IN ('red', 'green')`).

name	color	price	age
Auto 01	green	4999	14
Auto 02	blue	9999	11
Auto 03	red	14999	7
Auto 04	white	19999	5
Auto 05	red	1000	14
Auto 06	yellow	2000	12
Auto 07	gray	5100	10
Auto 08	blue	9000	8
Auto 09	red	13000	5
Auto 10	green	5900	9

Tabelle 1: Beispiel-Datenbank Gebrauchtwagen

Die Pareto-Präferenz `price lowest AND age around 10` ist mittels eine Priorisierung mit dem ersten Teil verbunden, ist also insgesamt weniger wichtig als die am Anfang stehende POS-Präferenz.

Demnach werden hier nur noch die roten und grünen Autos berücksichtigt, da diese bzgl. der POS-Präferenz optimal sind.

Als Beispiel betrachten wir die Auswertung der oben angegebenen Query auf die Tabelle 1, welche gebrauchte Autos enthält.

Dabei würden die folgenden Autos als Ergebnis resultieren:

name	color	price	age
Auto 01	green	4999	14
Auto 03	red	14999	7
Auto 05	red	1000	14
Auto 10	green	5900	9

Wie zu sehen ist, sind alle Autos rot oder grün, weil dies in der Priorisierung am höchsten gewichtet ist. Die Bedingungen der Pareto-Präferenz werden dagegen von manchen Tupeln besser, von manchen schlechter erfüllt. So ist das Auto 03 zwar näher am gewünschten Alter als das andere rote Auto 05, hat jedoch einen deutlich höheren Preis. Damit sind die beiden unvergleichbar, und treten somit beide im Ergebnis auf. Ähnliches ist bei den beiden grünen Autos zu beobachten.

Die Autos 01 und 05 beispielsweise sind beide im Ergebnis, da sie verschiedene Farben haben, und somit bezüglich der POS-Präferenz unvergleichbar sind. Durch das Erweitern

der POS-Präferenz um eine geeignete SV-Relation, z.B. `REGULAR`, würden die Farben rot und grün vergleichbar werden, und somit die Autos 01 und 03 aus dem Ergebnis entfallen:

Anfrage mit SV:

```
SELECT * FROM used_cars
  PREFERRING color IN ('red', 'green') REGULAR
    PRIOR TO (price lowest AND age around 10)
```

Ergebnis mit SV:

name	**color**	**price**	**age**
Auto 01	green	4999	14
Auto 03	red	14999	7
Auto 05	red	1000	14
Auto 10	green	5900	9

Zu beachten ist die korrekte Klammerung bei Verwendung komplexer Präferenzen: Eine Priorisierung *bindet stärker* als eine Pareto-Präferenz. Daher ergibt sich folgende implizite Klammerung:

```
1 AND 2 PRIOR TO 3    ⇒    1 AND (2 PRIOR TO 3)
```

Soll also, wie im obigen Beispiel, die Pareto-Präferenz stärker binden, sind entsprechende Klammern zu setzen.

A. Anhang

A.1. Listings für PSQL hybrid

Komplettes Listing für die Verwendung von Preference SQL hybride Version in einem Java-Programm. Die folgende Klasse Minimalbeispiel.java baut auf der Klasse `preference.sql.samples.PrefSQLExampleNeu` auf.

Listing 11: Minimalbeispiel zur Verwendung von PSQL hybrid in Java

```
1  package preference.sql.samples;
2
3  import java.io.StringReader;
4  import java.sql.Connection;
5  import java.sql.DriverManager;
6  import java.sql.SQLException;
7  import java.util.ArrayList;
8  import java.util.Iterator;
9
10 import preference.exception.PreferenceException;
11 import preference.sql.PrefSQLOptimicer;
12 import preference.sql.SQLEngine;
13 import preference.sql.parser.ParseException;
14 import preference.sql.parser.PreferenceSQLParser;
15 import preference.sql.parser.SQLTreeBuilder;
16 import preference.sql.rela.RelAOpI;
17 import xxl.core.cursors.MetaDataCursor;
18 import xxl.core.relational.ArrayTuple;
19
20 public class Minimalbeispiel {
21
22   private static ArrayList executeQuery(String query,
23       String optimizer, boolean useOptimizer)
24       throws PreferenceException, InstantiationException,
25       IllegalAccessException, ClassNotFoundException,
26       SQLException, ParseException {
27
28     SQLEngine sqlEng = new SQLEngine();
29
30     /*
31      * Anmelden einer Score-Funktion an die Preference-Engine. Ab
32      * diesem Zeitpunkt ist die Verwendung der Score-Funktion in
33      * einem Präferenzausdruck möglich.
34      */
35     sqlEng.registerScoreFunction(new SimpleScoreFunc());
36     sqlEng.registerCombiningFunction(new SimpleRankFunc());
37
38     /*
```

```
39      * Datenbankverbindung aufbauen. In diesem Beispiel wird auf
40      * eine mysql-DB connected
41      */
42     Class.forName("com.mysql.jdbc.Driver").newInstance();
43
44     Connection sqlCon = DriverManager.getConnection(
45         "jdbc:mysql://localhost/test?user=aa&password=bb");
46
47     /*
48      * Preference-SQL-Statement-Parsen und Query-Tree aufbauen
49      */
50     PreferenceSQLParser prefParser = new PreferenceSQLParser();
51
52     SQLTreeBuilder sqlBuilder = new SQLTreeBuilder(sqlEng,
53         sqlCon);
54
55     prefParser.setSQLListener(sqlBuilder);
56
57     // re-init and reset the parser
58     prefParser.ReInit(new StringReader(query));
59     prefParser.reset();
60
61     /*
62      * Parsen des Statements
63      */
64     prefParser.SQLSelect();
65
66     RelAOpI opRoot = sqlBuilder.getQueryTree();
67
68     // run optimizer if necessary
69     if (useOptimizer)
70       opRoot = runOptimizer(optimizer, sqlCon, opRoot);
71
72     MetaDataCursor cursor = opRoot.getQuery(null);
73
74     ArrayList results = new ArrayList();
75
76     while (cursor.hasNext()) {
77       ArrayTuple tup = (ArrayTuple) cursor.next();
78       results.add(tup);
79     }
80     cursor.close();
81
82     return results;
83   }
84
85   public static ArrayList getUsedCarsFromResults(
86       ArrayList results, String prefix) {
87     ArrayList cars = new ArrayList();
```

```
88
89      Iterator it = results.iterator();
90
91      while (it.hasNext()) {
92        ArrayTuple tup = (ArrayTuple) it.next();
93
94        UsedCar car = new UsedCar();
95        car.age = tup.getFloat(prefix + ".age");
96        car.color = tup.getString(prefix + ".color");
97        car.id = tup.getFloat(prefix + ".id");
98        car.name = tup.getString(prefix + ".name");
99        car.price = tup.getFloat(prefix + ".price");
100
101       cars.add(car);
102     }
103     return cars;
104   }
105
106   private static RelAOpI runOptimizer(String optimizer,
107       Connection sqlCon, RelAOpI opRoot)
108       throws PreferenceException {
109     PrefSQLOptimicer opPref = new PrefSQLOptimicer();
110     /* entweder ... */
111     opPref.initOptimicer(optimizer);    /* ... oder ... */
112     opPref.initOptimicer(OptimizerConfig.getDefaultConfig(
113         optimizer));
114     opRoot = opPref.optimizeQuery(opRoot,
115         "../PreferenceSQL/optimizer/constraint.xml", sqlCon);
116     return opRoot;
117   }
118
119   public static void main(String[] args)
120       throws InstantiationException, IllegalAccessException,
121       ClassNotFoundException, SQLException, ParseException,
122       PreferenceException {
123
124     ArrayList resTuple = executeQuery(
125         "SELECT * FROM used_cars
126           PREFERRING color IN ('red') ELSE ('green')",
127         "optimizer.xml", true);
128
129     ArrayList cars = getUsedCarsFromResults(resTuple,
130         "used_cars_1");
131   }
132 }
```

A.2. Listings für PSQL McKoi

Listing 12: Minimalbeispiel zu PSQL McKoi im Server-Mode

```
1  package com.mckoi.preferences.samples;
2
3  import java.sql.*;
4
5  public class ConnectToServerDemo {
6
7    public static void main(String[] args) {
8
9      // Register the Mckoi JDBC Driver
10     try {
11       Class.forName("com.mckoi.JDBCDriver").newInstance();
12     } catch (Exception e) {
13       System.out.println("Unable to register the JDBC Driver.\n"
14           + "Make sure the JDBC driver is in the\n"
15           + "classpath.\n");
16       System.exit(1);
17     }
18
19     /*
20      * This URL specifies we are connecting with a database
21      * server on localhost.
22      */
23     String url = "jdbc:mckoi://localhost/";
24
25     // The username / password to connect under.
26     String username = "aa";
27     String password = "bb";
28
29     // Make a connection with the database.
30     Connection connection;
31     try {
32       connection = DriverManager.getConnection(url, username,
33           password);
34     } catch (SQLException e) {
35       System.out.println("Unable to make a connection to " +
36           "the database.\n The reason: " + e.getMessage());
37       System.exit(1);
38       return;
39     }
40
41     try {
42       Statement stmt = connection.createStatement();
43
44       stmt.execute("SELECT * FROM used_cars
45           PREFERRING color IN ('red') ELSE ('green')");
46
```

```
47        ResultSet rs = stmt.getResultSet();
48
49        while (rs.next()) {
50          String s = rs.getString("name");
51          System.out.print("name: " + s);
52          int i = rs.getInt("price");
53          System.out.println(", price: " + i);
54        }
55
56        // Close the connection when finished
57        connection.close();
58
59      } catch (SQLException e) {
60        System.out.println("An error occured\n"
61            + "The SQLException message is: " + e.getMessage());
62        return;
63      }
64
65    }
66  }
```

Hierzu muss zusätzlich die McKoi-Datenbank angelegt und im Server-Mode gestartet sein (siehe Kapitel 2.2 und 2.3).

Um die McKoi-Datenbank anstatt im Server-Mode im Embedded-Mode auszuführen, genügt es, in der Zeile 23 die URL entsprechend anzupassen. Für den Embedded-Mode ist folgende URL zu verwenden:

Listing 13: Anpassung des Minimalbeispiel für Embedded-Mode

```
19      /* This URL specifies we are connecting with a local database
20       * within the file system. './db.conf' is the path of the
21       * configuration file of the database to embed.
22       */
23      String url = "jdbc:mckoi:local://./configs/db.conf";
```

Im Embedded-Mode muss die Datenbank lediglich angelegt (Kapitel 2.2), nicht jedoch als Server gestartet sein.

A.3. Listing SCORE- und RANK-Funktionen

Ein Beispiel für eine Klasse, die eine SCORE-Funktion implementiert. Hier bekommen Fahrzeuge mit der Kategorie *Roadster* den Score 3, *Van's* bekommen den Score 5, usw.:

Listing 14: Beispiel einer SCORE-Funktion

```
1  package preference.sql.samples;
2
3  import preference.algebra.BasePreference;
4  import preference.algebra.ScoreFunctionBase;
5  import preference.exception.PreferenceException;
6
7  public class SimpleScoreFunc extends ScoreFunctionBase {
8
9      public SimpleScoreFunc() {
10     }
11
12     public int getDomainType() {
13         return BasePreference.domain_DOUBLE;
14     }
15
16     /**
17      * Returns the name of the function
18      */
19     public String getName() {
20         return "MyScore";
21     }
22
23     /**
24      * Computes the score value for the given object
25      */
26     public double score(Object objPref, Object objContext)
27             throws PreferenceException {
28         String kategorie =
29                 m_attribSelector.evaluateString(objPref,
30                         objContext);
31         if (kategorie.equals("ROADSTER"))
32             return 3;
33         else if (kategorie.equals("VAN"))
34             return 5;
35         else if (kategorie.equals("KOMBI"))
36             return 6;
37         else
38             return 7;
39     }
40 }
```

Ein Beispiel für eine Klasse, die eine RANK-Funktion implementiert. In dieser RANK-Funktion werden die Werte der einzelnen, enthaltenen SCORE-Funktionen miteinander multipliziert:

Listing 15: Beispiel einer RANK-Funktion

```
1  package preference.sql.samples;
2
3  import preference.algebra.CombiningFunction;
4  import preference.exception.PreferenceException;
5
6  public class SimpleRankFunc extends CombiningFunction {
7
8      public SimpleRankFunc() {
9      }
10
11
12     public String getName() {
13         return "AddRank";
14     }
15
16     public double score(Object objPref, Object objContext)
17             throws PreferenceException {
18
19         double dScoreVal = 1;
20
21         for (int i = 0; i < getCount(); i++) {
22             dScoreVal = dScoreVal
23                     * getScoreFunction(i).score(objPref,
24                             objContext);
25         }
26
27         return dScoreVal;
28     }
29 }
```

A.4. Constraint-Datei

Die Constraint-Datei für den Optimizer in PSQL-Hybrid.

Listing 16: Contraint-Datei, constraint.xml

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<DatabaseMetaData>

  <ForeignKey>

    <reference table = "USED_CARS_1"  column = "REF"
               reftable = "CATEGORY"    refcolumn = "REF"/>
    <reference table = "USED_CARS_5"  column = "REF"
               reftable = "CATEGORY"    refcolumn = "REF"/>

    <reference table = "USED_CARS_1"  column = "SID"
               reftable = "SELLER"    refcolumn = "SID"/>
    <reference table = "USED_CARS_5"  column = "SID"
               reftable = "SELLER"    refcolumn = "SID"/>

  </ForeignKey>

  <PrimaryKey>

    <key table = "TEST1"      column = "ID"/>
    <key table = "TEST2"      column = "ID"/>

    <key table = "SELLER"     column = "SID"/>
    <key table = "CATEGORY"   column = "REF"/>

    <key table = "USED_CARS_1"    column = "CARID"/>
    <key table = "USED_CARS_5"    column = "CARID"/>
    <key table = "USED_CARS_10"   column = "CARID"/>
    <key table = "USED_CARS_50"   column = "CARID"/>
    <key table = "USED_CARS_100"    column = "CARID"/>

    <key table = "A"      column = "X"/>
    <key table = "B"      column = "X"/>

  </PrimaryKey>

</DatabaseMetaData>
```

A.5. Konfigurationsdatei für McKoi-Datenbank

Konfigurationsdatei zum Erstellen einer neuen McKoi-Datenbank.

Listing 17: Konfigurationsdatei für McKoi-Datenbank, db.conf

```
1  ############################################################
2  #
3  # Configuration options for the Mckoi SQL Database.
4  #
5  # NOTE: Lines starting with '#' are comments.
6  #
7  ############################################################
8
9  #
10 # database_path - The path where the database data files
11 #   are located.
12 #   See the 'root_path' configuration property for the
13 #   details of how the engine resolves this to an
14 #   absolute path in your file system.
15
16 database_path=../data
17
18 #
19 # log_path - The path the log files are written.
20 #   See the 'root_path' configuration property for the
21 #   details of how the engine resolves this to an
22 #   absolute path in your file system.
23 #   The log path must point to a writable directory.  If
24 #   no log files are to be kept, then comment out (or
25 #   remove) the 'log_path' variable.
26
27 log_path=../log
28
29 #
30 # root_path - If this is set to 'jvm' then the root
31 #   path of all database files is the root path of the
32 #   JVM (Java virtual machine) running the database
33 #   engine.  If this property is set to 'configuration'
34 #   or if it is not present then the root path is the
35 #   path of this configuration file.
36 #   This property is useful if you are deploying a
37 #   database and need this configuration file to be the
38 #   root of the directory tree of the database files.
39
40 root_path=configuration
41 #root_path=jvm
42
43 #
44 # jdbc_server_port - The TCP/IP port on this host where
```

```
45 #   the database server is mounted.  The default port
46 #   of the Mckoi SQL Database server is '9157'
47
48 jdbc_server_port=9157
49
50 #
51 # ignore_case_for_identifiers - If enabled all
52 #   identifiers are compared case insensitive.  If
53 #   disabled (the default) the case of the identifier
54 #   is important.
55 #   For example, if a table called 'MyTable' contains
56 #   a column called 'my_column' and this property is
57 #   enabled, the identifier 'MYTAble.MY_COlumN' will
58 #   correctly reference the column of the table.  If
59 #   this property is disabled a not found error is
60 #   generated.
61 #   This property is intended for compatibility with
62 #   other database managements systems where the case
63 #   of identifiers is not important.
64
65 ignore_case_for_identifiers=disabled
66
67 #
68 # socket_polling_frequency - Mckoi SQL maintains a pool
69 #   of connections on the server to manage dispatching
70 #   of commands to worker threads.  All connections on
71 #   the jdbc port are polled frequently, and ping
72 #   requests are sent to determine if the TCP
73 #   connection has closed or not.  This value determines
74 #   how frequently the connections are polled via the
75 #   'available' method.
76 #   The value is the number of milliseconds between each
77 #   poll of the 'available' method of the connections
78 #   input socket stream.  Different Java implementations
79 #   will undoubtedly require this value to be tweaked.
80 #   A value of '3' works great on the Sun NT Java 1.2.2
81 #   and 1.3 Java runtimes.
82 #
83 #   NOTE: This 'socket polling' module is a horrible hack
84 #   and will be removed at some point when the threading
85 #   performance improves or there is an API for non-
86 #   blocking IO.  I'll probably write an alternative
87 #   version for use with the improved Java version.
88
89 socket_polling_frequency=3
90
91
92
93
```

```
94 # ----- PLUG-INS -----
95
96 #
97 # database_services - The services (as a Java class) that
98 #   are registered at database boot time.  Each service
99 #   class is separated by a semi-colon (;) character.
100 #   A database service must extend
101 #   com.mckoi.database.ServerService
102 #
103 #database_services=mypackage.MyService
104
105 #
106 # function_factories - Registers one or more FunctionFactory
107 #   classes with the database at boot time.  A
108 #   FunctionFactory allows user-defined functions to be
109 #   incorporated into the SQL language.  Each factory class
110 #   is separated by a semi-colon (;) character.
111 #
112 #function_factories=mypackage.MyFunctionFactory
113
114 #
115 # The Java regular expression library to use.  Currently
116 # the engine supports the Apache Jakarta regular expression
117 # library, and the GNU LGPL regular expression library.
118 # These two regular expression libraries can be found at the
119 # following web sites:
120 #
121 # GNU Regexp: http://www.cacas.org/~wes/java/
122 # Apache Regexp: http://jakarta.apache.org/regexp/
123 #
124 # The libraries provide similar functionality, however they
125 # are released under a different license.  The GNU library
126 # is released under the LGPL and is compatible with GPL
127 # distributions of the database.  The Apache Jakarta library
128 # is released under the Apache Software License and must not
129 # be linked into GPL distributions.
130 #
131 # Use 'regex_library=gnu.regexp' to use the GNU library, or
132 # 'regex_library=org.apache.regexp' to use the Apache
133 # library.
134 #
135 # NOTE: To use either library, you must include the
136 #  respective .jar package in the Java classpath.
137
138 regex_library=gnu.regexp
139
140
141
142
```

```
143 # ----- PERFORMANCE -----
144
145 #
146 # data_cache_size - The maximum amount of memory (in bytes)
147 #   to allow the memory cache to grow to.  If this is set
148 #   to a value < 4096 then the internal cache is disabled.
149 #   It is recommended that a database server should provide
150 #   a cache of 4 Megabytes (4194304).  A stand alone
151 #   database need not have such a large cache.
152
153 data_cache_size=4194304
154
155 #
156 # max_cache_entry_size - The maximum size of an element
157 #   in the data cache.  This is available for tuning
158 #   reasons and the value here is dependant on the type
159 #   of data being stored.  If your data has more larger
160 #   fields that would benefit from being stored in the
161 #   cache then increase this value from its default of
162 #   8192 (8k).
163
164 max_cache_entry_size=8192
165
166 #
167 # lookup_comparison_list - When this is set to 'enabled'
168 #   the database attempts to optimize sorting by generating
169 #   an internal lookup table that enables the database to
170 #   quickly calculate the order of a column without having
171 #   to look at the data directly.  The column lookup
172 #   tables are only generated under certain query
173 #   conditions.  Set this to 'disabled' if the memory
174 #   resources are slim.
175
176 lookup_comparison_list=enabled
177
178 #
179 # lookup_comparison_cache_size - The maximum amount of
180 #   memory (in bytes) to allow for column lookup tables.
181 #   If the maximum amount of memory is reached, the lookup
182 #   table is either cached to disk so that is may be
183 #   reloaded later if necessary, or removed from memory
184 #   entirely.  The decision is based on how long ago the
185 #   table was last used.
186 #
187 #   This property only makes sense if the
188 #   'lookup_comparison_list' property is enabled.
189 #
190 # NOTE: This property does nothing yet...
191
```

```
192 lookup_comparison_cache_size=2097152
193
194 #
195 # index_cache_size - The maximum amount of memory (in
196 #   bytes) to allow for the storage of column indices.
197 #   If the number of column indices in memory reaches
198 #   this memory threshold then the index blocks are
199 #   cached to disk.
200 #
201 # ISSUE: This is really an implementation of internal
202 #   memory page caching but in Java.  Is it necessary?
203 #   Why not let the OS handle it with its page file?
204 #
205 # NOTE: This property does nothing yet...
206
207 index_cache_size=2097152
208
209 #
210 # max_worker_threads - The maximum number of worker
211 #   threads that can be spawned to handle incoming
212 #   requests.  The higher this number, the more
213 #   'multi-threaded' the database becomes.  The
214 #   default setting is '4'.
215
216 maximum_worker_threads=4
217
218 #
219 # soft_index_storage - If this is set to 'enabled', the
220 #   database engine will keep all column indices behind a
221 #   soft reference.  This enables the JVM garbage collector
222 #   to reclaim memory used by the indexing system if the
223 #   memory is needed.
224 #
225 #   This is useful for an embedded database where requests
226 #   are rare.  When the database part is idle, the index
227 #   memory (that can take up significant space for large
228 #   tables) is reclaimed for other uses.  For a dedicated
229 #   database server it is recommended this is disabled.
230 #
231 #   Enable this if you need the engine to use less memory.
232 #   I would recommend the config property
233 #   'lookup_comparison_list' is disabled if this is enabled.
234 #   The default setting is 'disabled'.
235
236 soft_index_storage=disabled
237
238 #
239 # dont_synch_filesystem - If this is enabled, the engine
240 #   will not synchronize the file handle when a table change
```

```
#   is committed.  This will mean the data is not as
#   safe but the 'commit' command will work faster.  If this
#   is enabled, there is a chance that committed changes will
#   not get a chance to flush to the file system if the
#   system crashes.
#
#   It is recommended this property is left commented out.
#
#dont_synch_filesystem=enabled

#
# transaction_error_on_dirty_select - If this is disabled
#   the 4th conflict (dirty read on modified table) will
#   not be detected.  This has transactional consequences
#   that will cause data modifications to sometimes be
#   out of syncronization.  For example, one transaction
#   adds an entry, and another concurrent transaction
#   deletes all entries.  If this is disabled this
#   conflict will not be detected.  The table will end up
#   with the one entry added after commit.
#
#   It is recommended this property is left commented out.
#
#transaction_error_on_dirty_select=disabled

# ----- SPECIAL -----

#
# read_only - If this is set to 'enabled' then the database
#   is readable and not writable.  You may boot a database
#   in read only mode from multiple VM's.  If the database
#   data files are stored on a read only medium such as a
#   CD, then the property must be enabled else it will not
#   be possible to boot the database.
#   ( Uncomment the line below for read only mode )
#read_only=enabled

# ----- DEBUGGING -----

#
# debug_log_file - The file that is used to log all debug
#   information.  This file is stored in the 'log_path'
#   path.

debug_log_file=debug.log

#
# debug_level - The minimum debug level of messages that
#   are written to the log file.  Reducing this number
```

```
290 #   will cause more debug information to be written to
291 #   the log.
292 #     10 = INFORMATION
293 #     20 = WARNINGS
294 #     30 = ALERTS
295 #     40 = ERRORS
296
297 debug_level=20
298
299 #
300 # table_lock_check - If this is enabled, every time a
301 #   table is accessed a check is performed to ensure that
302 #   the table owns the correct locks.  If a lock assertion
303 #   fails then an error is generated in the log file.
304 #   This should not be enabled in a production system
305 #   because the lock assertion check is expensive. However
306 #   it should be used during testing because it helps to
307 #   ensure locks are being made correctly.
308
309 table_lock_check=enabled
```

A.6. Konfigurationsdatei für PSQL

Die Konfigurationsdatei für PSQL enthält momentan lediglich einen Eintrag für den gewünschten Detailgrad der Systemausgaben (Debuglevel), siehe Kapitel 1.7.

Listing 18: Konfigurationsdatei psql.conf

```
1 #Configuration for PreferenceBase
2 #Debuglevel: 0=QUIET, 1=LOW, 2=NORMAL, 3=HIGH 4=INSANE
3 #Fri Jul 07 02:46:56 CEST 2006
4 debuglevel=2
```