Maximilian Schüle et al. (Hrsg.): BTW 2019,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 1

Ja-(zu-)SQL: Evaluation einer SQL-Skriptsprache für Hauptspeicherdatenbanksysteme

Maximilian Schüle[1] Linnea Passing[2] Alfons Kemper[3] Thomas Neumann[4]

Abstract: Große Datenmengen in Wirtschaft und Wissenschaft werden klassischerweise in Datenbanksystemen verwaltet. Um die Daten für maschinelles Lernen sowie für die Datenanalyse zu nutzen, ist ein zeitintensiver zyklischer Transformations- und Verschiebeprozess nötig, da die Daten hierfür in anderen Formaten oder schlicht in anderen Plattformen vorliegen müssen. Forschungsarbeiten der letzten Jahre widmen sich der Aufgabe, Datenverwaltung und Datenanalyse in *einem* System zu integrieren, um teure Datentransfers zu vermeiden.

Diese Arbeit untersucht die Leistungsfähigkeit gespeicherter Prozeduren (*stored procedures*) zur Implementierung von Data-Mining-Algorithmen in Datenbanksystemen. Grundlage hierfür bildet *HyPerScript*, die PL/SQL-ähnliche Skriptsprache des Hauptspeicherdatenbanksystems HyPer. Insbesondere evaluieren wir die prototypische Implementierung von fünf Algorithmen, die ganz ohne separates Datenanalysesystem auskommt.

1 Einleitung

Mit wachsenden Leistungsanforderungen bei der Datenverarbeitung gewinnen sowohl Hauptspeicher-Datenbanksysteme für die Datenhaltung als auch die automatisierte Verarbeitung großer Datenmengen an Bedeutung. Wenn beide Domänen kombiniert werden, entfällt das zyklische Extrahieren, Transformieren und Laden von Daten (sogenannter ETL-Prozess). Somit können die Daten in Echtzeit statt um die Prozessdauer verspätet analysiert werden. Datenbanksysteme bieten prozedurale Sprachen wie PL/SQL bis hin zu in C geschriebene Erweiterungen an, um innerhalb des Datenbanksystems Algorithmen zu spezifizieren.

Neben der Möglichkeit, Algorithmen in beliebigen prozeduralen Sprachen zu implementieren, integrieren verschiedene Datenbanksysteme Data-Mining-Algorithmen direkt. Für auf *PostgreSQL* basierende Datenbanksysteme stellt *MADlib* eine Bibliothek an Funktionen zur Datenanalyse bereit. Das Projektziel ist, eine Bibliothek an Erweiterungen bereitzustellen, die an die Pakete aus *GNU R* angelehnt ist. Für eine Echtzeitdatenanalyse in Hauptspeicherdatenbanksystemen stellt *HyPer* Operatoren für die Assoziationsanalyse mit Apriori, Clusteranalyse mit k-Means und DBSCAN und Graphanalyse mit PageRank bereit. Diese

[1] TU München, Lehrstuhl für Datenbanksysteme, Boltzmannstraße 3, 85748 Garching, m.schuele@tum.de
[2] TU München, Lehrstuhl für Datenbanksysteme, Boltzmannstraße 3, 85748 Garching, passing@in.tum.de
[3] TU München, Lehrstuhl für Datenbanksysteme, Boltzmannstraße 3, 85748 Garching, kemper@in.tum.de
[4] TU München, Lehrstuhl für Datenbanksysteme, Boltzmannstraße 3, 85748 Garching, neumann@in.tum.de

Operatoren sind Teil der Algebra und werden von *HyPer* in die ganzheitliche Anfrageoptimierung eingebunden. Vordefinierte Operatoren kommen dem Anspruch, Datenbanksysteme um beliebige Anwendungslogik zu erweitern, nur begrenzt nach.

Um Datenbanksysteme für beliebige Algorithmen und maschinelles Lernen zu erweitern ohne jeweils einen Operator zu implementieren, ist eine domänenspezifische Sprache nötig, die prozedurale Konstrukte mit eingebettetem SQL bereitstellt. Eine prozedurale Skriptsprache kombiniert mit der deklarativen Anfragesprache SQL ermöglicht nutzerseitig definierte Funktionen (*user defined function, UDF*) in kompakterer Darstellung als rein imperative Sprachen.

Weitere Vorteile einer Skriptsprache, bei der die Anwendungslogik in das Datenbanksystem hinein verlagert wird, sind die Integration in die Anfrageoptimierung des Datenbanksystems und ein geringerer Transfer von Daten. Als gespeicherte Prozeduren sind *UDF*s Teil des Anfrageplans und somit Teil der ganzheitlichen Anfrageoptimierung. Zudem erlaubt die Leistungssteigerung der Datenbankserver durch Verwendung moderner Hardware, diese für mehr als nur reine Datenverwaltungsaufgaben zu verwenden. Damit Datenbankserver mehr Anwendungslogik übernehmen können, muss es für Datenbankanwender möglich sein, Algorithmen für die Zielsprache universell zu entwickeln. Um den Funktionsumfang einer solchen Skriptsprache identifizieren zu können, ist relevant, wie sich Algorithmen abstrahieren lassen und welche Bausteine dafür unabdingbar sind.

In dieser Arbeit identifizieren wir notwendige Bausteine, indem wir ausgewählte Data-Mining/Machine-Learning-Algorithmen (Apriori, DBSCAN, k-Means, PageRank, Lineare Regression) in *HyPerScript*, der Skriptsprache des Hauptspeicherdatenbanksystems *HyPer*, implementieren. Anschließend vergleichen wir die Laufzeit und Skalierbarkeit der Algorithmen im Vergleich zu bereits implementierten Operatoren [Pa17], die das Datenbanksystem bereitstellt. Die Arbeit liefert folgende Beiträge:

- Die Vorstellung von *HyPerScript* als exemplarische Erweiterung eines Hauptspeicherdatenbanksystems, die es Nutzern erlaubt, Funktionen in um prozedurale Ausdrücke erweitertem SQL zu formulieren.
- Die beispielhafte Beschreibung einer betriebswirtschaftlichen Anwendung in der Form einer TPC-C-Anfrage in *HyPerScript*. Dabei handelt es sich um den primären Anwendungsfall der Skriptsprache.
- Die Erweiterung von SQL um einen Tensor-Datentyp, um bestimmte Datenanalyse-Algorithmen zu ermöglichen.
- Die Analyse des Potentials einer Skriptsprache in Datenbanksystemen, um beliebige Algorithmen zu formulieren; demonstriert an den Algorithmen PageRank, Apriori, lineare Regression, k-Means und DBSCAN, die bereits als feststehende Operatoren in *HyPer* existieren.

- Eine umfangreiche Evaluierung von *HyPerScript*-Funktionen als *stored procedures* im Vergleich zu implementierten Operatoren.

Eine umfassende und performante domänenspezifische Skriptsprache bildet die Grundlage für die Entwicklung einer deklarativen Sprache. Eine auf SQL basierende, prozedural-deklarative Sprache ermöglicht Datenanalysten, unabhängig vom darunterliegenden System zu programmieren. Die Plattformunabhängigkeit und die Wiederverwendbarkeit von SQL erhöht den Anreiz, komplexe Algorithmen bereits im Datenbanksystem auszuführen.

Die Arbeit ist wie folgt gegliedert: Nach einem Überblick über verwandte Arbeiten und einer Einführung in *HyPerScript* erfolgt eine Vorstellung der implementierten Algorithmen. Die Evaluation beurteilt die Leistungsfähigkeit der Skriptsprache im Vergleich zu den in *HyPer* integrierten Operatoren. Die Zusammenfassung diskutiert die Ergebnisse in Hinblick spezifischer Sprachen für die Datenanalyse.

2 Verwandte Arbeiten

Im Folgenden erläutern wir aktuell in der Praxis relevante Systeme und den Stand der Forschung, Algorithmen zur Wissensgewinnung in Datenbanksysteme zu integrieren.

Die code-kompilierenden, SQL-basierten Hauptspeicherdatenbanksysteme *EmptyHeaded* [Ab17] und *HyPer* [KN11] unterstützen bereits einige Data-Mining-Algorithmen. Dazu stellt *EmptyHeaded* eine Schnittstelle für SQL-Abfragen und andere Algorithmen bereit, die in einer übergeordneten Sprache wie Python adressiert werden. Das dieser Arbeit zugrundeliegende Hauptspeicherdatenbanksystem *HyPer* ist ein hybrides OLTP und OLAP Datenbanksystem, welches parallel Echtzeit-Transaktionen verarbeiten sowie mehrfache analysierende Anfragen ausführen kann. Es war bahnbrechend für die Kompilierung anstelle von Interpretation von SQL-Anfragen [Ne11], bietet einige Data-Mining-Algorithmen als feststehende Operatoren als Teil der Algebra an und integriert sie somit in die Anfrageoptimierung [Pa17; Th15].

Die für statistische Analysen verbreitete Open-Source-Anwendung *GNU R* [7] bietet Module wie für die hier behandelten Algorithmen an, die sich über eine eigene Kommandozeile steuern lassen. Diese Module sind vergleichbar mit den in *HyPer* angebotenen Operatoren, denn beide (sowohl die Module als auch die Operatoren) repräsentieren Bausteine mit einer hart kodierten Funktionalität. Wenn man neue Bausteine hinzufügen oder die bestehenden verändern möchte, so müssen in C++ neue Pakete (*GNU R*) bzw. neue Operatoren (*HyPer*) entwickelt werden. Die Alternative zur Neuentwicklung stellen in R definierbare Funktionen dar, für die prozedurale Sprachkonstrukte wie Schleifen und weitere Kontrollstrukturen bereitstehen. Das Gegenstück, um Funktionen in Datenbanksystemen zu definieren, sind die in den nachfolgenden Kapitel vorgestellten *stored procedures* mit *HyPerScript*.

[7] R `http://www.r-project.org/`

Neben *GNU R* existieren weitere Programme, wie das davon für performante Datenanalysen abgeleitete *Julia*[8], das, wie unser *HyPerScript*, LLVM-Code generiert. Für maschinelles Lernen, etwa, um neuronale Netze zu trainieren, sind *Python* mit *SciPy*[9] oder *TensorFlow* [Ab16] ausgelegt.

Ein vergleichbarer Ansatz zu mit *HyPerScript* kompilierten Funktionen benennen Crotty et. al. [Cr15]. Ihr Ansatz kompiliert, kombiniert und optimiert beliebige Funktionen (*User Defined Functions*) zu LLVM-Code, um Vektorinstruktionen und Pipelining auf Datensätzen zu nutzen. Eine Gemeinsamkeit ist die Kompilierung der nutzerseitig definierten Funktionen zu LLVM-Code (obwohl das Suffix „Script“ in *HyPerScript* fälschlicherweise zur Annahme verleitet, die Funktionen würden interpretiert). Um eine bessere Laufzeit von PostgreSQL zu erhalten, kompiliert der Ansatz in [BG16; BG17] SQL-Anfragen zu LLVM-Code. Eine weitere Modifikation von PostgreSQL ist die Erweiterung von PL/pgSQL um Funktionen höherer Ordnung [GSU13; GU13]. Die Integration algebraischer Datentypen in relationale Datenbanksysteme [Gi13] stellt eine weitere Herausforderung dar.

Eine zu *HyPerScript* vergleichbare Programmiersprache bietet *SAP HANA* mit *SQLScript* [BMM13]. *SQLScript* erweitert SQL um prozedurale und funktionale Prozeduren, erlaubt *MapReduce* auf analytischen Anfragen und erweitert Rekursion für die Graphanalyse. Alternativ zu prozeduralen Erweiterungen erweitert *FunSQL* [Bi12] SQL um funktionale Konstrukte. Die Erweiterung intendiert, ähnlich wie diese Arbeit, die Verlagerung von Applikationslogik auf Datenbankserver.

Um Applikationlogik auf Datenbankserver verlagern zu können, hilft eine eigene Sprache, die zu SQL übersetzt. Ein Beispiel hierfür ist *Ferry* [Sc10], eine Metasprache mit den Konstrukten *for*, *where*, *group by*, *order by* und *return*. *Ferry* wird zuerst aus gängigen Skript- oder Programmiersprachen erzeugt und anschließend in SQL übersetzt.

3 HyPerScript

HyPerScript ist die prozedurale Sprache zu *HyPer*, analog zu PL/SQL in Oracle [Lo08], zu PL/pgSQL[10] in *PostgreSQL* oder zu *SQLScript* in *SAP HANA* [BMM13]. Für transaktionale Anwendungen in *HyPer* entwickelt, erlaubt *HyPerScript* es, TPC-C-, TPC-E- und TPC-H-Benchmarks auszuführen. Außerdem können mit der um prozedurale Ausdrücke erweiterten SQL-Syntax beliebige Algorithmen formuliert werden.

Die in *HyPerScript* definierten Prozeduren werden während der Kompilierungsphase in LLVM-Code übersetzt. List. 1 zeigt die Sprachspezifikation in EBNF. HyPerScript verwendet die im SQL:2003-Standard [Ei04] spezifizierten Sprachkonstrukte inklusive Fensterfunktionen (*window functions*). Die SQL-Befehle lassen sich in prozedurale Konstrukten wie Schleifen (`while<Bedingung>{...}`) und Iterationen (`select index from`

[8] Julia `http://julialang.org/`

[9] SciPy `http://www.scipy.org/`

[10] PL/pgSQL `https://www.postgresql.org/docs/10/static/plpgsql.html`

```
Statement        = ( ( VarDeclaration  | VarDefinition  | TableDeclaration
                     | SelectStatement | EmbeddedSQL    | ReturnStatement
                     | IfStatement     | WhileStatement | ControlStatement | FunctionCall ) ";" )* ;

VarDeclaration   = "var" NAME Type "=" Expression ;
VarDefinition    = NAME "=" Expression ;
TableDeclaration = "table" NAME "(" NAME Type ("," NAME Type)* ")";
SelectStatement  = SQLSelect ("{" Statement "}") ("else" "{" Statement "}" ) ;
EmbeddedSQL      = SQLInsert | SQLUpdate | SQLDelete | CSVCopy ;
ReturnStatement  = "return" NAME | "rollback" ;
IfStatement      = "if" "(" Expression ")" "{" Statement "}" "else" "{" Statement "}" ;
ControlStatement = "break" | "continue" ;
WhileStatement   = "while" "(" Expression ")" "{" Statement "}" ;
FunctionCall     = NAME "(" Expression ("," Expression )* ")";
```

List. 1: Sprachdefinition von *HyPerScript*: `SQL{Select,Insert,Update,Delete}` entsprechen den SQL:2003-Befehlen, `Type` den SQL-Datentypen, `Expression` den SQL-Ausdrücken. Neben klassischen Ausdrücken prozeduraler Sprachen ermöglicht `EmbeddedSQL` auf einzelnen Tupeln der SQL-Anfrage zu operieren.

```
create or replace function insert_until(anzahl integer not null) as $$
   select index as i from sequence(0,anzahl){
      var rand = random(100); if (rand = 13) { continue }; insert into sample(rand));
   }
$$ language 'hyperscript' strict;
```

List. 2: Beispielhafte User-Defined-Function mit *HyPerScript*: Hier werden `anzahl` viele Zufallszahlen in eine Relation namens `sample` eingefügt, randomisiert zwischen `0` und `100`, eine Zahl `13` soll übersprungen werden.

`sequence(1,10){...}`) mit den Steuerungsbefehlen `break` und `continue` und Bedingungen (`if(<Bedingung>){...}`) einbinden. Zusätzlich lassen sich temporäre Tabellen erstellen, auf Basis der SQL-Datentypen Variablen deklarieren und definieren (`var foo int[]='{}'`) und ebenfalls als Parameter beim Funktionsaufruf mit übergeben oder zurückgeben.

List. 2 demonstriert eine beispielhafte Anwendung von *HyPerScript*. Mittels deklarativem SQL wird ein Intervall definiert, das hier als Iteration ähnlich einer For-Schleife genutzt wird. Auf die einzelnen Ergebnistupel kann innerhalb des Geltungbereiches zugegriffen und diese auch wieder in SQL-Befehlen verwendet werden. In diesem Fall wird eine Relation namens `sample` mit Zufallswerten befüllt.

3.1 HyPerScript mit Tensoroperationen

Zusätzlich zu prozeduralen Kontrollbefehlen erachten wir Tensoroperationen als elementar für die Eignung von Datenbanksystemen für maschinelles Lernen, um zum Beispiel lineare Regression numerisch zu lösen. Daher verwendet *HyPer* einen zu Tensoren erweiterten Array-Datentyp, der als Attribut in Datenbankschemata Tensoren von beispielsweise Ganzzahlen oder Gleitkommazahlen erlaubt. Die Array-Operationen erfolgen auf den Attributen von Relationen als Teil der Projektion im `SELECT`-Teil einer SQL-Anfrage. Das umschließt die folgenden Anwendungen auf Tensoren:

1. **Algebra**: Dazu gehören Tensorprodukt, -addition, -subtraktion und das Skalarprodukt, das Invertieren, Transponieren und Potenzieren von Tensoren.
2. **Erzeugen**: Neben dem Initialisieren mit `ARRAY[<WERTE>]` bzw. `{<WERTE>}`, benötigen wir die Identitätsmatrix `array_id(<Dimension>)` und einen Tensor gefüllt mit einem angegebenen Wert (`array_fill(<WERT>,<Dimensionen>)`).
3. **Zugriff**: Zugriff erfolgt über `array_access(<ARRAY>,<Indizes>)` bzw. `ARRAY[<Index>]`. Die Funktion zum Schneiden `array_slice(<ARRAY>,<Indizes>)` erlaubt eine Teilmenge eines Arrays zu extrahieren, um Daten zum Beispiel in Test- und Trainingsmenge zu teilen.
4. **Verändern**: Nach Erzeugung der Tensoren müssen einzelne Elemente effizient verändert werden, dazu dient `array_set(<ARRAY>,<WERT>,<Indizes>)`. Das Gegenstück zum oben erwähnten Schneiden ist das Konkatenieren von Arrays `<ARRAY>||<ARRAY>`.
5. **Mengenoperationen**: Um Teilmengen von Arrays zu identifizieren, braucht ein Datenbanksystem Mengenoperationen wie $A \subseteq B$ als `<ARRAY> <@ <ARRAY>`. Um Elemente in einer Teilmenge zu finden, steht $a \in A$ als `a ANY= A` zur Verfügung.
6. **Normalisieren**: `unnest(<ARRAY>)` als Gegenstück zum Initialisieren von Arrays extrahiert die Elemente in einzelne Tupel.

Die eingeführten Tensoroperationen erlauben die Verarbeitungen von Datensätzen aggregiert zu Tensoren. Als Beispielanwendungen mit Tensoren sind in List. 3 die Kreuzvalidierung auf einem Datensatz und in List. 4 die binäre Exponentiation in *HyPerScript* angeführt. Die binäre Exponentiation benötigt die Identitätsmatrix und die Multiplikation und steht stellvertretend für die C++-Implementierung im Datenbanksystem. Die aufgeführte einfache Kreuzvalidierung schneidet aus dem Datensatz jedes Tupel einmal heraus, um dieses als Testmenge und die restlichen als Trainingsmenge zu verwenden. Darauf wird lineare Regression (in einer eigenen Funktion) angewendet, um die optimalen Gewichte zu berechnen. Mit diesen wird der Vorhersagefehler auf dem herausgeschnitten Datensatz bestimmt.

```
create or replace function linearregression(x float[][], y float[][]) returns float[] as $$
   return (array_transpose(x)*x)^-1*(array_transpose(x)*y);
$$ language 'hyperscript' strict;
create or replace function cross_validate(x float[][], y float[][]) returns float as $$
   var error=0; var n=array_length(x,2); var m=array_length(x,1);
   select index i from sequence(2,n-1){
      var weights_o=linearregression(array_slice(x,1,i-1,1,m)||array_slice(x,i+1,n,1,m),
                        array_slice(y,1,i-1,1,n)||array_slice(y,i+1,n,1,1));
      error=error+((array_slice(x,i,i,1,m)*weights_o)[1][1]-y[i][1])^2;
   }
   error=error/(n-2); return error;
$$ language 'hyperscript' strict;
```

List. 3: Einfache Kreuzvalidierung in *HyPerScript*: Mittels `array_slice()` und der Konkatenation wird schrittweise eine Zeile aus dem Datensatz herausgeschnitten und als Testdatensatz verwendet. `array_slice()` erwartet als Argument den Tensor und für jede Dimension die Start- sowie Endposition.

```
create or replace function pow(a_in float[][], e_in int) returns float[] as $$
   var a=a_in; var e=e_in; if( e<0 ) { e=e*-1; a=array_transpose(a); }
   var mask = 1<<63; var result = array_identity(array_ndims(a));
   while(mask>0){ result=result*result; if( e&mask>0 ){ result=result*a; } mask=mask>>1; }
   return result;
$$ language 'hyperscript' strict;
```

List. 4: Binäre Exponentiation `pow()` von Tensoren implementiert in *HyPerScript*: als Eingabe wird ein Tensor und ein Exponent erwartet, eine Schleife mit Multiplikationen berechnet die Exponentiation.

3.2 HyPerScript für betriebswirtschaftlich transaktionale Anwendungen

HyPerScript erlaubt über die SQL-Schnittstelle betriebswirtschaftlich transaktionale Anwendungen auszuführen. Beispiele hierfür sind die TPC-C-, TPC-E- und TPC-H-Benchmarks des Transaction Processing Performance Councils[11]. So modelliert der TPC-C-Benchmark ein Handelsunternehmen mit eingehenden Aufträgen (`New-Order`) von Kunden und misst, wie viele schreibende Transaktionen ein Datenbanksystem pro Zeiteinheit verarbeiten kann.

```
create type newOrderPosition as (line_number int not null,supware int not null,itemid int not null, qty
      int not null);
create function newOrder (w_id int not null, d_id smallint not null, c_id int not null, positions setof
      newOrderPosition not null, datetime timestamp not null) as $$
   select w_tax from warehouse w where w.w_id=w_id;
   select c_discount from customer c where c_w_id=w_id and c_d_id=d_id and c.c_id=c_id;
   select d_next_o_id as o_id,d_tax from district d where d_w_id=w_id and d.d_id=d_id;
   update district set d_next_o_id=o_id+1 where d_w_id=w_id and district.d_id=d_id;
   select count(*) as cnt from positions;
   select case when count(*)=0 then 1 else 0 end as all_local from positions where supware<>w_id;
   insert into "order" values (o_id,d_id,w_id,c_id,datetime,null,cnt,all_local);
   insert into neworder values (o_id,d_id,w_id);
   update stock
   set s_quantity=case when s_quantity>=qty+10 then s_quantity-qty else s_quantity+91-qty end,
       s_remote_cnt=s_remote_cnt+case when supware<>w_id then 1 else 0 end,
       s_order_cnt=s_order_cnt+1, s_ytd=s_ytd+qty
   from positions where s_w_id=supware and s_i_id=itemid;
   insert into orderline
          select o_id,d_id,w_id,line_number,itemid,supware,null,qty,
                 qty*i_price*(1.0+w_tax+d_tax)*(1.0-c_discount),
                 case d_id when 1 then s_dist_01 when 2 then s_dist_02 when 3 then s_dist_03 when 4 then
                        s_dist_04 when 5 then s_dist_05 when 6 then s_dist_06 when 7 then s_dist_07 when
                        8 then s_dist_08 when 9 then s_dist_09 when 10 then s_dist_10 end
          from positions, item, stock
          where itemid=i_id
          and s_w_id=supware and s_i_id=itemid
   returning count(*) as inserted;
   if (inserted<cnt) rollback;
$$ language 'hyperscript' strict;
```

List. 5: *HyPerScript*-Funktion `newOrder()` des TPC-C-Benchmarks: Die Funktion nimmt eine Menge von Bestellpositionen `positions` für ein Warenhaus `w_id` eines Distrikts `d_id` von einem Kunden `c_id` entgegen.

List. 5 zeigt die zugehörige Prozedur in *HyPerScript*. Die Prozedur nimmt einen neuen Auftrag aus mehreren Bestellpositionen (`setof`) für ein Warenhaus (`w_id`) von einem

[11] `http://www.tpc.org`

Kunden (`c_id`) in einem Distrikt (`d_id`) entgegen. Zunächst wird dieser Auftrag in der Datenbank angelegt. Anschließend aktualisiert die Prozedur die Lagerbestände in `stock` und fügt die Bestellposition zusammen mit dem berechneten Preis in `orderline` ein.

HyPerScript ermöglicht Attributwerte von SQL-Anfragen in Variablen zwischenzuspeichern, um sie so wiederverwenden zu können. So werden beispielsweise der Steuersatz des Warenhauses und der Kundenrabatt in den Variablen `w_tay` und `c_discount` zwischengespeichert und beim Einfügen in die Relation `orderline` wiederverwendet.

Das Beispiel prüft die Atomarität und Konsistenz für Transaktionen, die implizit durch die Verwendung eines Datenbanksystems gegeben ist. So überprüft die Prozedur am Ende, ob die Bestellungen erfolgreich in `orderline` eingefügt worden sind (die Zahl der eingefügten Bestellungen valide ist) und setzt andernfalls die Transaktion zurück (`rollback`). Außerdem verdeutlicht das Beispiel die kompakte Schreibweise: Die *HyPerScript*-Prozedur kommt mit weniger als 40 Zeilen aus, während ein C++-Programm für dieselbe Funktion deutlich mehr Zeilen beansprucht[12].

4 HyPerScript für Datenanalysealgorithmen

Neben den klassischen transaktionalen Anwendungen ermöglicht *HyPerScript* auch die Ausführung beliebiger Algorithmen im Datenbanksystem. In diesen Kapitel stellen wir dazu beispielhaft Formulierungen einiger Algorithmen zur Datenanalyse dar. Um möglichst verschiedene Bereiche der Datenanalyse abzudecken, wählen wir grundlegende Algorithmen aus Assoziationsanalyse (Apriori-Algorithmus), Clustering (k-Means und DBSCAN), Graphmetriken (PageRank) und Regression (lineare Regression). Modernes SQL – mit Fensterfunktionen (*window functions*) und rekursiven temporären Tabellen (*recursive common table expressions*) – ist bereits Turing-vollständig. Mit *HyPerScript* lassen sich Algorithmen jedoch teils deutlich kompakter und verständlicher formulieren.

Dabei folgen die *HyPerScript*-Formulierungen jeweils folgendem Aufbau: Für jeden Algorithmus wird ein Schema als eigener Namensbereich definiert. Bei Aufruf der UDF `<Schemaname>.run()` wird der Algorithmus ausgeführt. Das Ergebnis des Algorithmus wird in eine Relation desselben Namensbereiches materialisiert. Die Logik der Algorithmen wird großteils in SQL-Funktionen ausgedrückt. Die UDF `<Schemaname>.run()` stellt diese Funktionen zum eigentlichen Algorithmus zusammen, beispielsweise durch iterativen Aufruf bis ein Abbruchkriterium erreicht wird.

Alle hier untersuchten Algorithmen sind in *HyPer* bereits als Operatoren vorhanden [Pa17; Th15]. Das heißt, sie wurden, ähnlich wie relationale Operatoren (etwa Hash-Join oder Aggregation), im Datenbankkern implementiert und lassen sich kombinieren, um SQL-Anfragen abzubilden. Die folgenden Abschnitte erläutern jeweils kurz den gewählten Algorithmus, stellen die Formulierung in *HyPerScript* dar und beschreiben die Vergleichsoperatoren. Im

[12] vgl.: `https://github.com/evanj/tpccbench/blob/master/tpcctables.cc`

anschließenden Evaluationskapitel werden Operatoren und *HyPerScript*-Formulierungen miteinander verglichen.

4.1 Assoziationsanalyse

Der 1993 eingeführte Apriori-Algorithmus [AIS93] ist der bekannteste Vertreter im Bereich der Assoziationsanalyse. Er basiert auf Warenkorbdaten, die als Tupel aus Transaktionsnummer (*tid*) und Waren abgelegt werden (sales: $\{[tid, item]\}$). Zuerst zählt er häufig vorkommende Item-Mengen, die mit einer minimalen relativen Häufigkeit (*Support*) von mindestens s_0 in allen Warenkörben vorkommen und bildet anschließend Assoziationsregeln daraus. Die Item-Mengen wachsen mit jeder Iteration um ein Element beginnend mit der einelementigen Menge. Dabei ist die Anzahl der Iterationen und zu überprüfender Mengen durch das *Apriori-Prinzip* begrenzt. Das Prinzip besagt, dass Mengen an Items, deren Teilmengen nicht häufig auftreten, selbst nicht häufig auftretend sein können. List. 6 zeigt den Aufruf des integrierten Operators wie der Skript-Funktion mit einem minimalen Support von 10 %, deren genaue Funktionsweisen kurz erklärt werden.

```
-- nativer Operator:
select * from apriori((table aprioriscript.sales),0.1,1);
-- HyPerScript
select aprioriscript.findFI(10); select * from aprioriscript.frequentitemsets;
```

List. 6: Aufruf des Apriori-Algorithmus als Operator wie als gespeicherte Prozedur.

Die Implementierung in *HyPerScript* (siehe List. 7) basiert auf zu Warenkörben aggregierten, häufig auftretenden Item-Mengen, die mit rekursivem SQL iterativ erweitert werden. Hier werden die Arrays als Mengen verwendet und in jeder Iteration um ein Element erweitert. Anschließend wird die Häufigkeit der Tupel gezählt. Dazu wird jedes Itemset mit jedem Warenkorb anhand des Mengenoperators `Tupel <@ Warenkorb` verglichen.

```
create or replace function aprioriscript.findFrequentItemsets(minsupp integer not null) as $$
   with recursive -- Erzeugung von Warenkörben
   transactions (tid, bucket) as (select tid, array_agg(item) from aprioriscript.sales group by tid),
   -- häufig auftretende 1-Item-Mengen
   sales_supp as (select item from aprioriscript.sales group by item having count(*) >= minsupp),
   frequentitemsets as ( -- Items mit support >= minsupp
      (select distinct array[p.item]::integer[] as items from sales_supp p) -- 1-Item-Mengen L_1 ∈ 𝓛_1
     union all ( -- erweitere Item-Mengen um ein Element: L_{k-1} erweitert
       select distinct array_append(t.items,p.item::integer)::integer[] -- L_k
       from frequentitemsets t, sales_supp p
       where minsupp <= ( -- zähle Support
          select count(*) from transactions t2 -- C_k ∈ C_k . support(C_k) >= minsupp ⇒ C ∈ 𝓛_k
          where array_append(t.items,p.item::integer)::integer[] <@ ( t2.bucket )
       ) and t.items[(select count(*) from unnest(t.items))]<p.item -- nur sortierte Arrays
   ))
   insert into aprioriscript.frequentitemsets (select * from frequentitemsets);
$$ language 'hyperscript' strict;
```

List. 7: Ermittlung der häufig auftretenden Item-Mengen für den Apriori-Algorithmus: Die Funktion erhält den minimalen Support s_0 als Parameter übergeben und berechnet eine rekursiv wachsende Relation mit häufigen Itemsets, beginnend bei den einelementigen, also den Waren als einelementige Arrays.

Der in *HyPer* implementierte Operator basiert auf der Speicherung der Elemente in einem mit jeder Iteration wachsenden Präfixbaum. Besonderheiten der Implementierung in *HyPer* sind die Parallelität pro Iterationsschritt sowie die Behandlung von Duplikaten. So berücksichtigen die Assoziationsregeln die Häufigkeit gleicher Elemente in Warenkörben.

4.2 Clustering

Ein weiterer wichtiger Bereich der Datenanalyse ist Clustering, also die Gruppierung ähnlicher Datenpunkte. Mit k-Means [Ll82] und DBSCAN [Sc17] formulieren wir zwei klassische Clustering-Algorithmen in *HyPerScript*, die über ganz unterschiedliche Iterationsmuster verfügen. k-Means weist in jeder Iteration jeden Datenpunkt dem nächstgelegenen Cluster zu. Durch die Zuweisung verändert sich der Mittelpunkt der Cluster, wodurch sich in der nächsten Iteration wiederum Zuordnungen ändern können. DBSCAN hingegen prüft für jeden Datenpunkt, ob in der Nähe ausreichend viele andere Datenpunkte liegen. Falls ja und falls diese Datenpunkte bereits einem Cluster zugeordnet wurden, wird auch der aktuell betrachtete Datenpunkt dem Cluster hinzugefügt. Andernfalls wird ein neuer Cluster begründet. Falls sich nicht genug andere Datenpunkte in der Nähe befinden, wird der aktuell betrachtete Datenpunkt als *Noise* deklariert. Anschließend wird der nächste Datenpunkt geprüft. List. 8 zeigt den Aufruf der beiden Clustering-Algorithmen sowohl als Operator wie als Skript-Funktion.

```
-- nativer Operator:
select * from kmeans((select x,y from kmeansscript.points),
                     (select x,y from kmeansscript.points LIMIT 5));
select * from dbscan((select x,y from dbscanscript.points),20,2);
-- HyPerScript:
select kmeansscript.run(5);
select dbscanscript.run(20,2);
```

List. 8: Aufruf der Clustering-Algorithmen in *HyPer*: Eingabedaten des k-Means-Operators sind die Datenpunkte und die initialen Zentren (im dargestellten Fall $k = 5$ zufällig gewählte Datenpunkte). Eingabe des DBSCAN-Operators sind ebenfalls die Datenpunkte, sowie als Parameter der Suchradius für nahegelegene Datenpunkte ϵ und die minimale Anzahl an Punkten pro Cluster `minPts`. Parameter für die *HyPerScript*-Funktionsaufrufe sind ebenfalls k für k-Means und ϵ sowie `minPts` für DBSCAN.

Die *HyPerScript*-Implementierung für k-Means (s. List. 9) basiert auf der in *HyPer* besonders effizient implementierten Fensterfunktion (*window function*) [Le15], die pro Datenpunkt eine Rangfolge der nächstgelegenen Clusterzentren berechnet. Aus den Mittelwerten der zugeordneten Punkte werden dann die neuen Clusterzentren bestimmt. Diese zwei Schritte werden wiederholt, bis die Zuordnung der Datenpunkte zu Clustern stabil ist.

Die DBSCAN-Implementierung [Hu17] (s. List. 10) basiert auf der rekursiven Erweiterung von Clustern: Zuerst bildet jeder Punkt einen eigenen Cluster. Anschließend werden Cluster, die weniger als ϵ weit voneinander entfernt liegen, vereinigt. Sowohl k-Means als auch DBSCAN liegen als in *HyPer* implementierte Operatoren vor. Der k-Means-Operator [Pa17] erhält als Eingabe die Datenpunkte sowie k initiale Clusterzentren. Im Normalfall wird

```
create or replace function kmeansscript.run(centers integer not null) as $$
   truncate kmeansscript.clusters;
   insert into kmeansscript.clusters(select id,x,y,0 from (select *,rank() over (order by id) from
         kmeansscript.points) where rank <= centers);
   while (true){
      select kmeansscript.computeCenters() as c_count;
      if(c_count = 0){ break; }
   }
$$ language 'hyperscript' strict;
create or replace function kmeansscript.computeCenters() returns integer not null as $$
   table clusters_tmp(cid int, x float, y float, count int);
   insert into clusters_tmp (select cid, avg(px), avg(py), count(*) from (
       select cid, p.x as px, p.y as py, rank() OVER ( partition by p.id
         order by (p.x-c.x)*(p.x-c.x)+(p.y-c.y)*(p.y-c.y) asc, (c.x*c.x+c.y*c.y) asc)
        from kmeansscript.points p, kmeansscript.clusters c ) x where x.rank=1 group by cid);
   select count(*) as c_count from (select * from kmeansscript.clusters except select * from
         clusters_tmp);
   truncate kmeansscript.clusters; -- lösche alte Daten
   insert into kmeansscript.clusters (select * from clusters_tmp);
   return c_count; -- gebe Anzahl der geänderten Zentren zurück
$$ language 'hyperscript' strict;
```

List. 9: Fixpunktiteration für k-Means: Die SQL-Funktion `computeCenters()` berechnet die Clusterzentren – im Beispiel als `avg(px), avg(py)` – und nutzt die Fensterfunktion `rank()`, um alle Datenpunkte dem jeweils nächstgelegenen Clusterzentrum zuzuordnen. Die Rahmenfunktion `run()` ruft `computeCenters()` so oft auf, bis in einer Iteration kein Datenpunkt mehr einem anderen Clusterzentrum zugeordnet wird.

```
create or replace function dbscanscript.run(eps float, minPoints int) as $$
   select count(*) as maxiter from dbscanscript.points;
   select index from sequence (1,maxiter){
      select dbscanscript.insertCluster(index,eps,minPoints,maxiter) as cond;
      if(cond=0){ break; }
   }
$$ language 'hyperscript' strict;
create or replace function dbscanscript.insertCluster(clusterid int not null, eps float not null,
      minPoints int not null, maxiter int not null) returns int not null as $$
   table pointstmp (id int, x float, y float);
   insert into pointsTmp(select * from dbscanscript.points except (select id,x,y from dbscanscript.
         pointsclustered) LIMIT 1);
   select count(*) as ret from dbscanscript.pointstmp;
   select index from sequence(1,maxiter){
      select count(*) as newc from pointsTmp c, dbscanscript.points p
         where (p.x-c.x)^2+(p.y-c.y)^2<eps^2 and c.id<>p.id;
      insert into pointsTmp(
         select * from (select * from dbscanscript.points except select * from pointsTmp) c where exists
                (select * from pointsTmp p where (p.x-c.x)^2+(p.y-c.y)^2<eps^2 and c.id<>p.id)
      );
      if(newc=0){ break; }
      ret=ret+newc;
   }
   if(ret < minPoints) { clusterid=NULL; }
   insert into dbscanscript.pointsclustered(select *,clusterid, false from dbscanscript.pointstmp);
   return ret;
$$ language 'hyperscript' strict;
```

List. 10: DBSCAN: Die Rahmenfunktion `run()` ruft `insertCluster()` auf: Pro Iteration ensteht ein neuer Cluster mit der als ersten Parameter angegebenen ID. Die Funktion `insertCluster()` nimmt pro Aufruf einen noch keinem Cluster zugewiesenen Punkt als neuen Cluster und erweitert diesen, bis er sich nicht mehr verändert. Die Funktion gibt die Anzahl der sich im Cluster befindlichen Punkte zurück. Die Anzahl der Iterationen ist limitiert durch die Anzahl aller Punkte.

der Operator parallel aufgerufen, das heißt die Datenpunkte liegen auf mehrere Threads verteilt vor. Die initialen Clusterzentren werden dann zunächst in jeden Thread repliziert, anschließend ordnet jeder Thread lokal seine Datenpunkte den nächstgelegenen Clusterzentren zu. Zur Ermittlung der neuen Clusterzentren müssen die Ergebnisse aller Threads zusammengefügt werden, jedoch werden die Ergebnisse bereits threadlokal aggregiert, um die Menge der zu übertragenen Daten sowie die Menge synchronisierter Berechnungen zu minimieren. Die neuen Clusterzentren werden nun wieder in alle Threads repliziert und die nächste Iteration beginnt, solange sich noch Clusterzuordnungen ändern. Der Operator generiert LLVM-Code, sodass der Algorithmus nahtlos in SQL-Anfragen hineinkompiliert werden kann und teure Funktionsaufrufe vermieden werden.

Der DBSCAN-Operator erhält als Eingabe die Datenpunkte sowie die Parameter ϵ und `minPts`. Für jeden unbesuchten Datenpunkt wird geprüft, ob genügend (`minPts`) andere Datenpunkte in der ϵ-Umgebung des Datenpunktes liegen, und welchem Cluster diese zugeordnet sind. Je nach Ergebnis dieser Prüfung wird der Datenpunkt einem Cluster zugeordnet, als *Noise* deklariert oder ein neuer Cluster wird begründet. Auch dieser Operator generiert LLVM-Code, ist im Gegensatz zu k-Means in *HyPer* jedoch nicht parallel implementiert.

4.3 Lineare Regression

Lineare Regression ist ein Optimierungsproblem um Labels vorherzusagen, dem ein lineares Modell $m(x) = w * x + w_0$ auf Daten x mit Gewichten w zugrunde liegt. Lineare Regression lässt sich als Optimierungsproblem mittels Gradientenabstiegsverfahren lösen. Der in *HyPer* implementierte Operator nutzt diesen Ansatz. Wir haben das Gradientenabstiegsverfahren mit festem Gradienten bei linearer Regression ebenfalls in *HyPerScript* implementiert. Als Alternative zu einem Optimierungsverfahren ist lineare Regression numerisch mit dem Gleichungssystem $\vec{w} = (X'^T X')^{-1} X'^T \vec{y}$ lösbar. List. 11 zeigt den Aufruf der verschiedenen Varianten.

```
-- nativer Operator:
select * from linearregression((select x_1, x_2, y  from linregscript.sample));
-- HyPerScript:
select linregscript.run();
-- mit Matrixoperationen:
select (array_transpose(x)*x)^-1*(array_transpose(x)*y)
from (select array_agg(x) x from (select ARRAY[1,x_1,x_2] as x from linregscript.sample) sx) tx,
     (select array_agg(y) y from (select ARRAY[y] y from linregscript.sample) sy) ty;
```

List. 11: Aufruf linearer Regression in *HyPer*: als Operator wie als gespeicherte Prozedur und komplett mit Matrixoperationen.

List. 12 zeigt den Gradientenabstieg programmiert in *HyPerScript*. Dazu haben wir den Gradienten hart kodiert, den wir in jeder Iteration für jedes Tupel berechnen. Als Besonderheit dieser Implementierung hängen die Rückgabewerte nur von den Eingabeparametern ab

(sind *immutable*, bzw. *stable* bei Lesezugriff auf die Datenbasis). Das ist möglich, da eine temporäre Tabelle (`table`) als Funktionsparameter mit übergeben werden kann (`setof`).

```
create or replace function linregscript.iterate(tuples float[] not null, y float not null, weights float
      [] not null, learningrate float not null) returns float[] not null as $$
   var gradient=array_access(weights,1);
   select index as i from sequence(1,array_length(tuples,1)){
      gradient = gradient + tuples[i]*weights[i+1];
   }
   gradient = gradient-y;
   var ret=array_set(weights,array_access(weights,1)-learningrate*gradient*2,1);
   select index as i from sequence(1,array_length(tuples,1)){
      ret=array_set(ret,weights[i+1]-learningrate*gradient*2*tuples[i],i+1);
   }
   return ret;
$$ language 'hyperscript' strict immutable;

create type features as (a float, b float, c float);
create or replace function linregscript.gradientdescent(tuples setof features, learningrate float not
      null, maxIter int not null) returns float[] not null as $$
   var weights='{1,1,1}'::float[];
   select index as i from sequence(1,maxIter){
      select a,b,c from tuples{
         weights=linregscript.iterate(ARRAY[a,b],c,weights,learningrate);
      }
   }
   return weights;
$$ language 'hyperscript' strict stable;

create or replace function linregscript.run(learningrate float not null, maxIter int not null) returns
      float[] not null as $$
   table sample(a float, b float, c float);
   insert into sample (select x_1,x_2,y from linregscript.sample);
   var weights=linregscript.run(sample,learningrate,maxIter);
$$ language 'hyperscript' strict stable;
```

List. 12: Stochastischer Gradientenabstieg mit linearer Regression in *HyPerScript*: Eine Funktion `iterate()` berechnet den Fehler pro Tupel mit den angegebenen Gewichten und gibt die optimierten Gewichte zurück. Eine Rahmenfunktion iteriert über alle Tupel, die die Lernrate und die Anzahl der Iterationen als Argumente erwartet.

Der *HyPer*-Operator unterstützt Parallelität, indem er wahlweise durch ein Flag parallelisiert zuerst lokale Gewichte berechnet oder die optimalen Gewichte pro eingehendem Tupel vorberechnet. Der Implementierung des Gradientenabstiegs in C++ gleicht der *HyPerScript*-Prozedur.

4.4 PageRank

PageRank ist ein Algorithmus für gerichtete Graphen, der ursprünglich zur Bestimmung der Wichtigkeit von Internetseiten gedient hat [BP98]. Eine Internetseite (Knoten) gilt dabei als wichtig, wenn wichtige Internetseiten auf sie verweisen (gerichtete Kanten). In jeder Iteration verteilt jeder Knoten einen Teil $(1 - \alpha)$ seines PageRank-Wertes gleichwertig über die ausgehenden Kanten auf die nächsten Knoten, und erhält entsprechend die Werte der auf ihn

verweisenden Knoten. Die Summe der eingehenden Werte ist der neue PageRank-Wert des Knotens. List. 13 zeigt den Aufruf von PageRank mittels Operators wie als Skript-Funktion, jeweils ohne Dämpfung.

```
-- nativer Operator:
select * from pagerank((table pagerankscript.edges),0);
-- HyPerScript:
select pagerankscript.run(0); select * from pagerankscript.pagerank;
```

List. 13: Aufruf von PageRank in *HyPer*: Sowohl der Operator als auch die *HyPerScript*-Formulierung benötigen als Eingabe die Kanten sowie den Dämpfungsfaktor $\alpha = 0$.

Die Berechnung der neuen PageRank-Werte lässt sich in SQL durch einen Join mit den Vorgängerknoten sowie eine Summen-Aggregation abbilden. Zur einfachen Formulierung der Iteration benötigen wir die Schleifen aus *HyPerScript* (s. List. 14).

```
create or replace function pagerankscript.computePR(alpha float not null) returns int not null as $$
   table pr_tmp(node int, edges int);
   insert into pr_tmp (
     select VTo, alpha*(cast((select count(*) from pagerankscript.pagerank) as float))
                 +(1-alpha)*sum(Beitrag)
     from (select e.VTo, p.Pagerank/(select count (*) from pagerankscript.edges x where x.VFrom=e.VFrom)
           as Beitrag from pagerankscript.edges e, pagerankscript.pagerank p where e.VFrom=p.Node) i
     group by VTo);
   select count(*) as c_count from (select * from pagerankscript.pagerank except select * from pr_tmp);
   truncate pagerankscript.pagerank;
   insert into pagerankscript.pagerank (select * from pr_tmp);
   return c_count; -- gebe Anzahl der geänderten Werte zurück
$$ language 'hyperscript' strict;

create or replace function pagerankscript.run(alpha float not null, maxIterations int not null) as $$
   select index as i from sequence(0,maxIterations) {
      select pagerankscript.computePR(alpha) as c_count;
      if(c_count = 0){ break; }
   }
$$ language 'hyperscript' strict;
```

List. 14: Approximation für PageRank: Die Rahmenfunktion `run()` ruft die Berechnung der PageRank-Werte `computePR()` solange auf, bis die Werte stabil sind. Die SQL-Funktion `computePR()` berechnet die PageRank-Werte aller Knoten mittels Aggregation. Dabei behält jeder Knoten den Anteil α seines alten PageRank-Wertes, und verteilt den Rest auf die durch ausgehende Kanten direkt mit ihm verbundenen Knoten.

Auch PageRank liegt als in *HyPer* implementierter Operator vor [Th15]. Die Besonderheit dieses Operators ist die Erzeugung eines temporären Indexes, der schnellen Zugriff auf die Vorgängerknoten erlaubt. Während der Ausführung des Algorithmus muss dann nicht mehr auf die Eingabedaten zugegriffen werden, sondern lediglich der Index verwendet und die PageRank-Werte aktualisiert werden. Dies kann von mehreren Threads parallel durchgeführt werden. Lediglich ein Synchronisationspunkt am Ende jeder Iteration ist nötig, um den Threads zu signalisieren, dass von nun an die neuen PageRank-Werte genutzt werden. Im folgenden Kapitel werden die Laufzeiten der Operatoren mit denen der vorgestellten *HyPerScript*-Implementierungen verglichen.

5 Evaluation

Die Evaluation vergleicht die Leistungsfähigkeit von mit *HyPerScript* erstellten Prozeduren und von Operatoren im Datenbankkern. Alle Experimente wurden auf einem Rechner mit Intel Xeon E5-2660 v2 CPU Prozessor (20 Kerne mit jeweils 2,20 GHz) und 256 GB DDR4 RAM gemessen, als Betriebssystem diente Ubuntu 17.04. Für den Apriori-Algorithmus wurden 100 verschiedenen Waren in 1000 Warenkörben synthetisch hergestellt. Die Anzahl der Elemente pro Warenkorb variierte gleichverteilt im Intervall $[0, 10]$. Für Clustering erzeugten wir 10^6 Punkte, deren x- wie y-Koordinaten gleichverteilt im Intervall $[0, 10^6]$ lagen. PageRank arbeitete auf 10^5 Knoten mit genauso vielen Kanten, für lineare Regressionen erzeugten wir 10^6 Tupel. Alle Experimente wurden dreimal wiederholt und für die Messungen der Median verwendet.

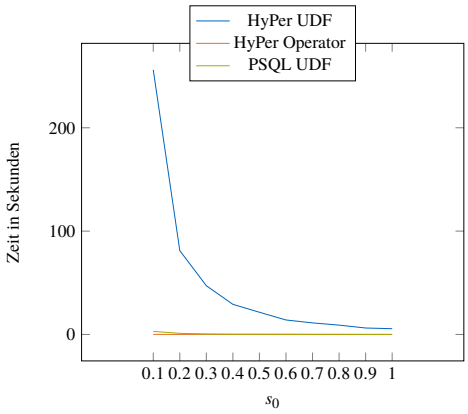
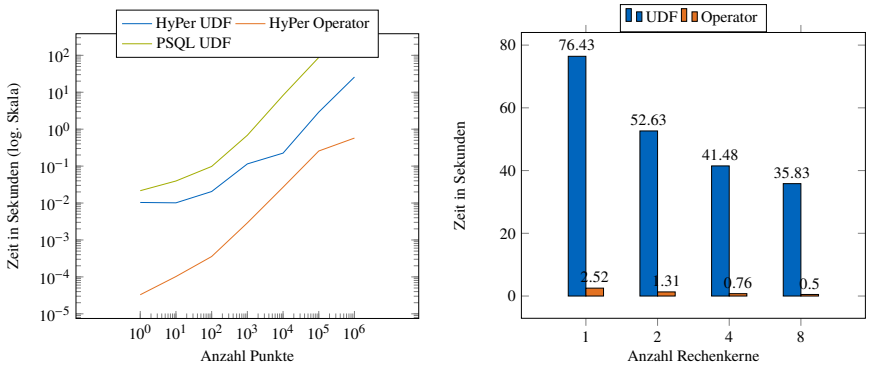


Abb. 1: Laufzeit für Assoziationsanalyse mit Apriori bei konstant 20 Rechenkernen: Abhängigkeit vom minimalen Support s_0 als Parameter von Apriori. Je größer s_0, desto strenger die Auswahl assoziierter Produkte, da weniger Kandidaten.

Für den Apriori-Algorithmus veränderten wir den minimalen Support (je größer, desto weniger häufige Item-Mengen existieren, s. Abb. 1). Mit wachsendem minimalen Support gleichen sich die Laufzeiten an, da die Zahl häufiger Item-Mengen abnimmt.

Die Laufzeiten der Clustering-Algorithmen wachsen linear mit der Eingabegröße (s. Abb. 2a, 3). Der k-Means-Algorithmus berechnet um 30 % schneller mit Hinzunahme eines weiteren Kernes (s. Abb. 2b). Die Parallelität des darunterliegenden Datenbanksystems gewährleistet die implizite parallele Ausführung des als Prozedur implementierten k-Means-Algorithmus; der k-Means-Operator ist explizit parallelisiert angelegt. Weder DBSCAN als Operator, noch als gespeicherte Prozedur unterstützen Skalierung. Während der Operator keine parallele Berechnung unterstützt, skalieren die verwendeten SQL-Anfragen in der *HyPerScript*-Prozedur schlecht, da pro (nicht parallelisierter) Iteration nur ein Tupel als neuer Cluster initialisiert wird.



(a) k-Means: Eingabegröße. (b) k-Means: Skalierung.

Abb. 2: Laufzeit für Clustering-Algorithmus k-Means mit fünf Zentren: (a) Laufzeit in Abhängigkeit von der Eingabegröße bei konstant 20 Rechenkernen sowie (b) in Abhängigkeit von der zur Verfügung stehenden Rechenkerne.

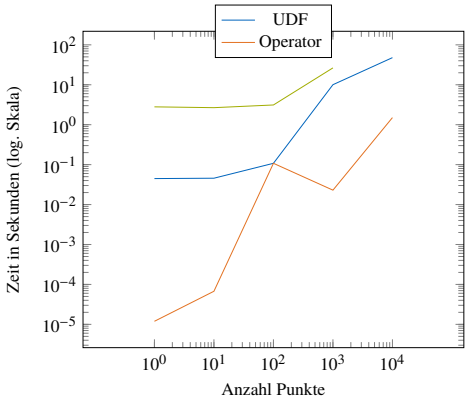
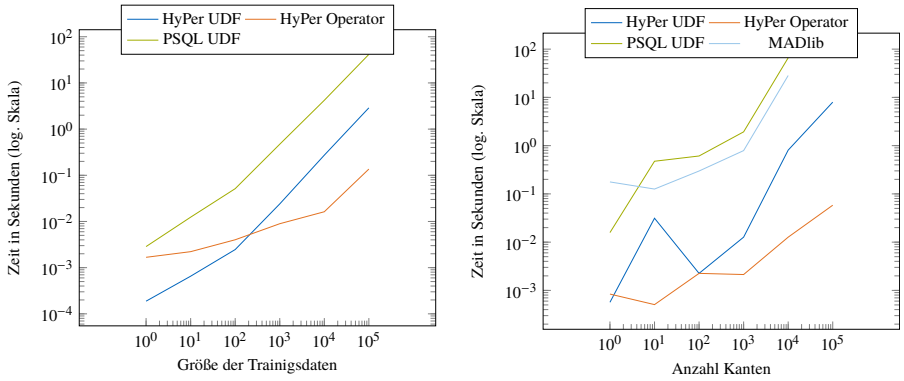


Abb. 3: Laufzeit für Clustering-Algorithmus DBSCAN mit $\epsilon = 20$ und `minPts` $= 2$: Laufzeit in Abhängigkeit von der Eingabegröße bei konstant 20 Rechenkernen.

Lineare Regression mit Gradientenabstieg in *HyPerScript* ist für bis etwa 100 Tupel schneller als der implementierte Operator (s. Abb. 4a). Beide Versionen sind gleich stukturiert, beide materialisieren zuerst die Tupel lokal und berechnen anschließend die optimalen Gewichte mit festem Gradienten. Allerdings kompiliert *HyPerScript* beide Schritte zu LLVM-Code basierend auf den Datenbanktypen, während der Operator C++-Funktionen auf den Basistypen aufruft. Bei wenigen Tupeln überwiegt der Aufwand durch Funktionsaufrufe, bei vielen Tupeln der Aufwand durch die Nutzung komplexer Datentypen.

Die Laufzeit des in *HyPerScript* implementierten PageRank-Algorithmus ist bei wenigen



(a) Lineare Regression (b) PageRank.

Abb. 4: Laufzeit bei jeweils konstant 20 Rechenkernen von (a) linearer Regression bei einer Lernrate von 0,85 und 45 Iterationen in Abhängigkeit der Größe zu trainierender Daten, sowie (b) zur Berechnung des PageRank-Wertes bei Veränderung der Anzahl der Kanten.

Tupeln vergleichbar mit der des PageRank-Operators (s. Abb. 4b). Bei wenigen Kanten ist die Stored-Procedure-Version des PageRank-Algorithmus sogar schneller als der optimierte Operator von *HyPer*, verliert allerdings mit zunehmender Anzahl an Kanten. Bei wenigen Kanten zeigt sich die Mehraufwand des integrierten Operators, da dieser ein Wörterbuch für die Knoten anlegt und Kanten in einer dünnbesetzte Matrix als Compressed-Sparse-Row (CSR) speichert. Mit zunehmender Kantenanzahl amortisieren sich der Mehraufwand für das Wörterbuch und die CSR-Datenstruktur, sodass der Operator schneller als die Skriptfunktion den PageRank-Wert berechnet.

Zusammenfassend zeigt die Evaluation, dass *HyPerScript* es ermöglicht, Prototypen zu entwickeln, ohne die relationale Algebra des Datenbanksystems zu erweitern. Die Laufzeiten der Prozeduren sind bei wenigen Tupeln vergleichbar zu derer der Operatoren der relationalen Algebra, erlauben allerdings nur Parallelität, wenn die zugrundeliegenden SQL-Anfragen entsprechend skalieren, wie es bei k-Means-Clustering zu sehen ist.

6 Zusammenfassung und Fazit

Skriptsprachen erlauben betriebswirtschaftlich transaktionale Anwendungen im Kern von Datenbanksystemen auszuführen. Typische Anwendungsfälle von SQL-Skriptsprachen sind die Implementierung von Datenbanktriggern und die serverseitige Ausführung von Teilen einer Applikation. Letzteres wird beispielsweise bei der Ausführung von Benchmarks benötigt, weshalb die große Mehrheit der Datenbanksysteme über solche prozeduralen SQL-Erweiterungen verfügt. Auch in *HyPer*, dem dieser Arbeit zugrundeliegenden Hauptspeicher-

datenbanksystem, wurde das prozedurale *HyPerScript* zunächst für die Leistungsbewertung verwendet.

Ziel der vorliegenden Arbeit war, das Potenzial von SQL-Skriptsprachen für andere Anwendungsfälle, insbesondere für Algorithmen zur Datenanalyse, auszuloten. Am Beispiel von *HyPerScript* zeigten wir, dass nutzerseitig definierte Funktionen in Kombination mit prozeduralen Ausdrücken eine kompakte Schreibweise von Algorithmen zur Datenanalyse erlauben. So zeigten wir anhand einer Beispielimplementierung, dass eine TPC-C-Implementierung in *HyPerScript* nur einen Bruchteil an Codezeilen einer Implementierung in einer rein prozeduralen Sprache benötigt. Neben Schleifen und Konditionen halfen Tensoroperationen, welche wir im Rahmen dieser Arbeit in *HyPer* integrierten, bei der numerischen Berechnung linearer Regression wie bei der Assoziationsanalyse.

Durch die Ausführung von Datenanalysealgorithmen im Datenbanksystem statt auf einer separaten Datenanalyseplattform werden teure ETL-Prozesse eingespart. Unsere Evaluation zeigt, dass die in *HyPerScript* formulierten Algorithmen nicht mit der Schnelligkeit hartkodierter Datenbankoperatoren mithalten können. Durch die kompakte Formulierung und die hohe Flexibilität im Vergleich zu hartkodierten Operatoren ist *HyPerScript* jedoch dennoch sinnvoll einsetzbar, um neue Algorithmen auf Datensätzen zu studieren ohne den Datenbankkern zu verändern. Zwar ist eine korrekte Ausführung der Algorithmen sichergestellt, deren Leistungsfähigkeit als Prozedur ist jedoch nachrangig. Eine SQL-nahe Skriptsprache wie *HyPerScript* spricht in erster Linie Nutzer aus dem Datenbankbereich an. Insofern ist *HyPerScript* erst als Beginn der Entwicklung einer universellen Sprache zu sehen, die Datenwissenschaftler direkt auf Datenbanksystemen nutzen können, um Daten aufzubereiten und zu analysieren.

Danksagung

Diese Arbeit ist im Rahmen des Projekts TUM Living Lab Connected Mobility (TUM LLCM) entstanden, das vom Bayerischen Staatsministerium für Wirtschaft, Energie und Technologie (StMWi) durch das Zentrum Digitalisierung.Bayern, einer Initiative der Bayerischen Staatsregierung, finanziert wird. Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung unter dem Förderkennzeichen TUM: 01IS17049 gefördert, sowie vom Europäischen Forschungsrat (ERC) im Rahmen des Forschungs- und Innovationsprogramms der Europäischen Union (Horizont 2020) (Finanzhilfevereinbarung Nr. 725286). Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Literatur

[Ab16] Abadi, M. et al.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. CoRR abs/1603.04467/, 2016, arXiv: 1603.04467, URL: http://arxiv.org/abs/1603.04467.

[Ab17] Aberger, C. R.; Lamb, A.; Olukotun, K.; Ré, C.: Mind the Gap: Bridging Multi-Domain Query Workloads with EmptyHeaded. PVLDB 10/12, S. 1849–1852, 2017, URL: http://www.vldb.org/pvldb/vol10/p1849-aberger.pdf.

[AIS93] Agrawal, R.; Imielinski, T.; Swami, A. N.: Mining Association Rules between Sets of Items in Large Databases. In: ACM SIGMOD, Washington, DC, USA, May 26-28, 1993. S. 207–216, 1993, URL: http://doi.acm.org/10.1145/170035.170072.

[BG16] Butterstein, D.; Grust, T.: Precision Performance Surgery for PostgreSQL: LLVM-based Expression Compilation, Just in Time. PVLDB 9/13, S. 1517–1520, 2016, URL: http://www.vldb.org/pvldb/vol9/p1517-butterstein.pdf.

[BG17] Butterstein, D.; Grust, T.: Invest Once, Save a Million Times - LLVM-based Expression Compilation in PostgreSQL. In: BTW (DBIS), 6.-10. März 2017, Stuttgart, Germany, Proceedings. S. 623–624, 2017, URL: https://dl.gi.de/20.500.12116/672.

[Bi12] Binnig, C.; Rehrmann, R.; Faerber, F.; Riewe, R.: FunSQL: it is time to make SQL functional. In: Proceedings of the 2012 Joint EDBT/ICDT Workshops, Berlin, Germany, March 30, 2012. S. 41–46, 2012, URL: https://doi.org/10.1145/2320765.2320786.

[BMM13] Binnig, C.; May, N.; Mindnich, T.: SQLScript: Efficiently Analyzing Big Enterprise Data in SAP HANA. In: BTW (DBIS), 11.-15.3.2013 in Magdeburg, Germany. Proceedings. S. 363–382, 2013, URL: https://dl.gi.de/20.500.12116/17332.

[BP98] Brin, S.; Page, L.: The Anatomy of a Large-Scale Hypertextual Web Search Engine. Computer Networks 30/1-7, S. 107–117, 1998, URL: https://doi.org/10.1016/S0169-7552(98)00110-X.

[Cr15] Crotty, A.; Galakatos, A.; Dursun, K.; Kraska, T.; Binnig, C.; Çetintemel, U.; Zdonik, S.: An Architecture for Compiling UDF-centric Workflows. PVLDB 8/12, S. 1466–1477, 2015, URL: http://www.vldb.org/pvldb/vol8/p1466-crotty.pdf.

[Ei04] Eisenberg, A.; Melton, J.; Kulkarni, K.; Michels, J.-E.; Zemke, F.: SQL:2003 Has Been Published. SIGMOD Conference 2004 33/1, S. 119–126, März 2004, ISSN: 0163-5808, URL: http://doi.acm.org/10.1145/974121.974142.

[Gi13] Giorgidze, G.; Grust, T.; Ulrich, A.; Weijers, J.: Algebraic data types for language-integrated queries. In: DDFP 2013, Rome, Italy, January 22, 2013. S. 5–10, 2013, URL: https://doi.org/10.1145/2429376.2429379.

[GSU13] Grust, T.; Schweinsberg, N.; Ulrich, A.: Functions Are Data Too (Defunctionalization for PL/SQL). PVLDB 6/12, S. 1214–1217, 2013, URL: http://www.vldb.org/pvldb/vol6/p1214-grust.pdf.

[GU13] Grust, T.; Ulrich, A.: First-Class Functions for First-Order Database Engines. In: (DBPL 2013), August 30, 2013, Riva del Garda, Trento, Italy. 2013, URL: http://arxiv.org/abs/1308.0158.

[Hu17] Hubig, N.; Passing, L.; Schüle, M. E.; Vorona, D.; Kemper, A.; Neumann, T.: HyPerInsight: Data Exploration Deep Inside HyPer. In: CIKM 2017, Singapore, November 06 - 10, 2017. S. 2467–2470, 2017, URL: https://doi.org/10.1145/3132847.3133167.

[KN11] Kemper, A.; Neumann, T.: HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In: ICDE 2011, April 11-16, 2011, Hannover, Germany. S. 195–206, 2011, URL: https://doi.org/10.1109/ICDE.2011.5767867.

[Le15] Leis, V.; Kundhikanjana, K.; Kemper, A.; Neumann, T.: Efficient Processing of Window Functions in Analytical SQL Queries. PVLDB 8/10, S. 1058–1069, 2015, URL: http://www.vldb.org/pvldb/vol8/p1058-leis.pdf.

[Ll82] Lloyd, S. P.: Least squares quantization in PCM. IEEE Trans. Information Theory 28/2, S. 129–136, 1982, URL: https://doi.org/10.1109/TIT.1982.1056489.

[Lo08] Loney, K.: Oracle Database 11g The Complete Reference. McGraw-Hill, Inc., 2008.

[Ne11] Neumann, T.: Efficiently Compiling Efficient Query Plans for Modern Hardware. PVLDB 4/9, S. 539–550, 2011, URL: http://www.vldb.org/pvldb/vol4/p539-neumann.pdf.

[Pa17] Passing, L.; Then, M.; Hubig, N.; Lang, H.; Schreier, M.; Günnemann, S.; Kemper, A.; Neumann, T.: SQL- and Operator-centric Data Analytics in Relational Main-Memory Databases. In: EDBT 2017, Venice, Italy, March 21-24, 2017. S. 84–95, 2017.

[Sc10] Schreiber, T.; Bonetti, S.; Grust, T.; Mayr, M.; Rittinger, J.: Thirteen New Players in the Team: A Ferry-based LINQ to SQL Provider. PVLDB 3/2, S. 1549–1552, 2010, URL: http://www.comp.nus.edu.sg/%5C%7Evldb2010/proceedings/files/papers/D09.pdf.

[Sc17] Schubert, E.; Sander, J.; Ester, M.; Kriegel, H.; Xu, X.: DBSCAN Revisited, Revisited: Why and How You Should (Still) Use DBSCAN. ACM Trans. Database Syst. 42/3, 2017, URL: http://doi.acm.org/10.1145/3068335.

[Th15] Then, M.; Passing, L.; Hubig, N.; Günnemann, S.; Kemper, A.; Neumann, T.: Effiziente Integration von Data- und Graph-Mining-Algorithmen in relationale Datenbanksysteme. In: Proceedings of the LWA 2015 Workshops: KDML, FGWM, IR, and FGDB, Trier, Germany, October 7-9, 2015. S. 45–49, 2015.