The Document Title

Example Author

Another Author

2022-11-08

1 Beschreibung

1.1 Projekt

NAME DES PROJEKTS: Dynamische Detektion von Inclusion Dependencies

STARTTERMIN: 24.10.2021

ENDTERMIN: 30.09.2022

Projektteilnehmende: Felix Köpge, Ragna Solterbeck, Helen Brüggmann

1.1.1 Die Ausgangslage

Im Status quo sind die meisten Data-Profiling Algorithmen statisch. Sie untersuchen eine statische Datenmenge auf Abhängigkeiten, wie **Functional Dependencies** oder **Inclusion Dependencies**. Wenn die Daten sich aber ändern, so muss der Algorithmus auf der gesamten Datenmenge neu ausgeführt werden. Für dynamische Datenmengen (bei denen Einträge hinzugefügt, gelöscht oder modifiziert werden) ist dieser Ansatz zu zeitaufwendig.

1.1.2 Projektvorhaben

Im Rahmen dieser Arbeit soll ein dynamischer Data-Profiling Algorithmus entwickelt werden, der Inclusion Dependencies auf dynamische Datenmengen fortlaufend entdeckt. Er soll alle Inclusion Dependencies entdecken, aber auch beim Einfügen oder Löschen von Einträgen überprüfen, ob dadurch Inclusion Dependencies aufgelöst werden oder neu-entstehen. Der Algorithmus soll auf große Datenmengen (= vorerst mehrere Gigabyte) skalierbar sein.

1.1.3 Projektidee

Als Ansatz soll ein verteilter Algorithmus entstehen, der alle Änderungen akzeptiert und prüft welche Kandidaten für neue Inklusions Abhängigkeiten entstanden sind oder welche Inclusion Dependencies sich aufgelöst haben könnten. Pruningmethoden sollen vermeiden, dass auf der gesamten Datenmenge gesucht wird. Beispielsweise sollen durch Betrachten von Metadaten und durch logische Implikationen bereits viele Datenkombinationen ausgeschlossen werden. Somit soll der dynamische Algorithmus wesentlich schneller ablaufen als ein statischer Algorithmus.

1.2 Auftraggeber

Die Arbeit ist entstanden im Rahmen einer Projektarbeit in der AG Big Data Analytics am Fachbereich Mathematik & Informatik der Philipps-Universität Marburg. Sie hat sich über zwei Semester erstreckt. Der leitende Professor ist Prof. Thorsten Papenbrock.

1.3 Qualitätsanforderungen

- **Exaktheit**: Der Algorithmus soll *alle* Inclusion Dependencies eines Datensets finden und *keine* falschen Resultate liefern.
- **Skalierbarkeit**: Der Algorithmus soll auf Datensets von mehreren GBs praktisch anwendbar sein und auf eine beliebige Anzahl an Host-Rechnern auslagerbar sein
- **Inkrementelle Ergebnisse**: Der Algorithmus soll periodisch (alle X Sekunden) Ergebnisse liefern. Er muss allerdings nicht für jeden einzelnen Daten-Poll (unter Poll-Architektur) seine Ergebnisse liefern.

2 Zusammenfassung

Wir haben ein verteiltes System für das Finden von Inclusion Dependencies (INDs) in dynamischen Datensets implementiert.

Unsere Lösung ist insofern verteilt, als dass das Speichern von Tabellenwerten und das Prüfen von IND-Kandidaten auf mehrere Data-Nodes verteilt werden kann. Das Einlesen von Datensets und das Generieren von IND-Kandidaten ist noch auf einen einzelnen Master-Node beschränkt.

3 Erfahrungsbericht

Felix

- Spark vs Akka
- Inclusion Deps vs Functional Deps
- Gegenprüfung gegen dynamischen, nur statischen
- Metronom (nicht auf wiederverwendbarkeit ausgelegt)

Kein Ausblick!

4 Entwicklungsprozess

4.1 Dynamischer Algorithmus

Wir begannen die Arbeit mit einer vorhandenen Akka Architektur von Prof. Papenbrock, die uns vorher als Hausaufgabenprojekt für das Modul Distributed Data Management diente.

Die vorhandene Architektur war bereits verteilt und konnte Inclusion Dependencies in statischen Datensets entdecken. Sie war allerdings nicht auf dynamische Datensets ausgelegt und stark begrenzt darin, dass ein einzelner Master-Node alle Werte eines Datensets im Hauptspeicher zwischenspeichern musste.

Über die erste Blockwoche hinweg haben wir unsere neue Lösung konzipierten und schrittweise Komponente entworfen. Der Einbau in dieser neuen Komponente in die vorhandene Architektur hat sich als eine schwerere Aufgabe erwiesen und zog sich bis zur zweiten Blockwoche und darüber hinweg. Die finale Lösung erinnert nur wenig an das ursprüngliche Hausaufgabenprojekt.

4.2 Datengenerator

Zu Beginn haben wir uns zunächst Gedanken darüber gemacht was der Datengenerator alles können muss, um einerseits der Aufgabenstellung gerecht zu werden und andererseits geeignete Datensätze für unser System zu liefern.

Wir kamen zu dem Schlüssen, dass 1. wir keinen vollständig synthetischen Datensatz einsetzen wollen und 2. wir unser System mit Datensätzen beliebiger Größe testen wollen. Es war also wichtig das der Generator aus einem verhanden Korpus einen beliebig langen Datenfluss generieren und zwischendurch einzelne Zeilen löschen kann.

Weiter mussten wir ein klares Format definieren, mit welchem die randomisierten Daten des Datengenerator in das verteilte System eingespeißt werden kann. Wir entschieden uns für ein CSV-basiertes Format, welches leicht in lesbarer Form auf der Kommandozeile ausgegeben werden kann.

Vor der Implementierung des Generator haben wir uns die einzelnen Klassen überlegt und definiert was diese jeweils können müssen und was sie dafür brauchen. Die Implementieren selbst wurde in Pair-Programming durchgeführt.

Die Planung und das Programmieren des Datengenerators fand zu großen Teilen in unserer ersten gemeinsamen Blockwoche statt und wurde stetig verbessert und schlussendlich finalisiert.

5 Team

Helen Brüggmann (M.Sc. Wirtschaftsinformatik):

- Protokollführung
- Projektdokumentation mit Jira
- Konzeption und Entwicklung des dynamischen Algorithmus (Pairprogramming mit Felix Köpge).

Felix Köpge (M.Sc. Informatik):

- Gesamt-Architekturkonzept
- Entwicklung des dynamischen Algorithmus (Pairprogramming mit Helen Brüggmann)
- Entwicklung des Datengenerators (Pairprogramming mit Ragna Solterbeck)

Ragna Solterbeck (M.Sc. Data Science):

- Konzeption und Entwicklung des Datengenerators (Pairprogramming mit Felix Köpge)
- Erstellung der Auslastungsdiagramme

6 Zeitplan

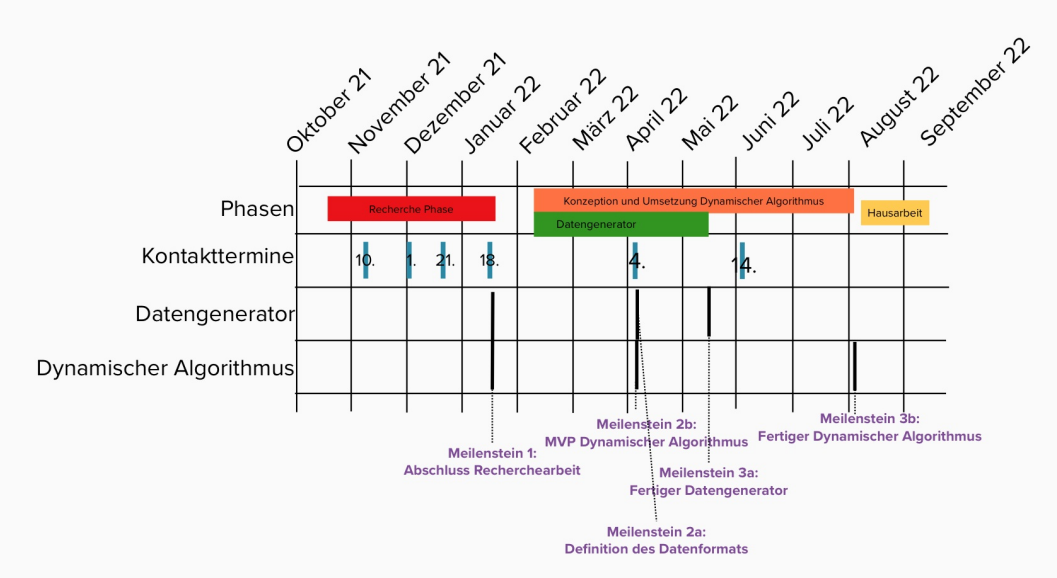
Projektzeitraum: 24.10.2021 - 31.08.2022

Aufgabe:

- Ein verteiltes System entwickeln welches auf dynamischen Datensätzen Inclusion Dependencies findet
- Einen Datengenerator entwickeln der es erlaubt mit zufälligen Daten das verteilte System zu testen

Projektphasen:

1. Recherche
2. Konzeptionierung und Umsetzung
3. Datengenerator
4. Dynamischer Algorithmus
5. Ausarbeitung der Hausarbeit



Phase 1: Recherchephase

Zeitraum: 24.10.2021 - 18.01.2021

In der Zeit haben wir uns in der Gruppe im Wochentakt getroffen und besprochen. Dazwischen hat jeder für sich recherchiert. Es ging darum zunächst das Thema zu durchdringen und Ideen zu sammeln, wie wir das Ganze umsetzen können. Die Kontakttermine mit Prof. Papenbrock hatten wir im 2 bis 4 Wochentakt. Dort haben wir unsere Ideen vorgestellt und besprochen. Parallel haben wir für das Modul Verteilte Systeme an einer Programmieraufgabe gearbeitet, in der wir mit einem verteilten Algorithmus auf statischen Daten Inclusion Dependencies finden sollten. Dadurch haben wir viel für unsere spätere Aufgabe gelernt.

Meilenstein 1: Abschluss Recherchearbeit

Ergebnisse der ersten Phase

Nach der Recherchephase haben wir uns auf folgende Aufgaben festgelegt

- Auffinden von unären Inclusion Dependencies in dynamischen Datensätzen
- Ein verteiltes System mit Akka in Java bauen, in dem die Inclusion Dependencies gesucht werden
- Eine Pipeline an Pruningschritten zu bauen um möglichst zeit- und datensparend Kombinationen für Inclusion Dependencies auszuschließen

Phase 2: Konzeptionierung und Umsetzung

In der nächsten Phase sind wir dazu übergegangen uns in größeren Abständen zu Blockwochen oder Sprintwochenenden zu treffen um am Stück runterprogrammieren zu können.

Datengenerator

Erste Programmiereinheit: 04.04.2022 - 08.04.2022

In einem einwöchigen Programmiersprint ist das Konzept und ein Großteil des Datengenerators entstanden. Als Datenformat wurden CSV Tabellen festgelegt, wobei die erste Spalte immer eine explizite Zeilen-Position ist, mit der man alte Daten überschreiben kann.

Meilenstein 2a: Definition des Datenformats

Zweite Programmiereinheit: 21.05.2022

Fertigstellung des Datengenerators

Meilenstein 3a: Fertiger Datengenerator

Ergebnis: g

Dynamischer Algorithmus

Erste Programmiereinheit: 04.04.2022 - 08.04.2022

In einem einwöchigen Programmiersprint sind erste Klassenentwürfe für den Algorithmus entstanden und ein erstes MVP in Form einer Dummy Main.

Meilenstein 2b: MVP Dynamischer Algorithmus

Zweite Programmiereinheit: 21.05.2022

Ausgehend vom MVP wurden nun die Klassenentwürfe und der Algorithmus iterativ und inkrementell immer wieder angepasst

Programmiereinheit: 01.08.2022 - 08.08.2022

Nachdem die Architektur für den Algorithmus noch einmal überarbeitet wurde, wurde der Algorithmus mitsamt seiner Pipeline final implementiert.

Meilenstein 3b: Fertigstellung Algorithmus

Phase 3: Hausarbeit

Parallel zur Fertigstellung des dynamischen Algorithmus wurde die Hausarbeit zu der Projektarbeit erstellt.

7 Definition des Zielsystems

7.1 Inclusion Dependencies

Inclusion Dependencies beschreiben, ob alle Werte die ein Attribut X annehmen kann auch von Attribut Y angenommen werden können. X und Y können aus Instanzen des gleichen Schemas (= der gleichen Tabelle) stammen, oder auch aus Instanzen zwei verschiedenen Schematas (= verschiedener Tabellen). Falls das der Fall ist, ist X abhängig von Y und man schreibt $X \subseteq Y$.

Formal bedeutet das: $\forall t_i[X] \in r_i, \exists t_j[Y] \in r_j$ mit $t_i[X] = t_j[Y]$ wobei t_i, t_j Schema-Instanzen sind und X, Y Attribute der Schemata.

Allgemein werden X und Y als Listen von Attributen gesehen, wobei stehts gelten muss $|X| = |Y|$.
Es wird von *unary* Inclusion Dependencies gesprochen wenn gilt $X \subseteq Y$ mit $|X| = |Y| = 1$. Falls $|X| = |Y| = n$ gilt, handelt es sich um eine *n-ary* Inclusion Dependency.

Für Inclusion Dependencies gelten immer folgende Eigenschaften:

- *Reflexiv:* Es gilt immer $X \subseteq X$
- *Transitiv:* Es gilt $X \subseteq Y \wedge Y \subseteq Z \Longrightarrow X \subseteq Z$
- *Permutationen:* Es gilt $(X_1, ..., X_n) \subseteq (Y_1, ..., Y_n)$, dann gilt auch $(X_1, ..., X_n) \subseteq (Y_1, ..., Y_n)$ für alle Permutationen σ1, … , σn

Beispiel für unary Inclusion Dependencies

Book

Title	Author	Price	Pages	Published
Database Systems	Ullman	214	1203	2007
Algorithms in Java	Sedgewick	130	768	2002
3D Computer Graphics	Watt	20	570	1999

Name ⊆ Title

Lending

ID	Name	Location	Student	Course
42	Database Systems	A-1.2	Miller	DBS 1
88	Database Systems	B-2.2	Miller	PT 1
73	Database Systems	A-1.2	Smith	DPDC
69	Algorithms in Java	C-E.1	Miller	PT 1
13	Algorithms in Java	C-E.1	Smith	DPDC

TODO quelle!

X := Attribut “Name” aus Tabelle “Lending”

Y := Attribut “Titel” aus Tabelle “Book”

Es ist leicht zu sehen, dass alle Werte die “Name” annehmen kann auch in Attribut “Titel” vertreten sind, daher folgt $X \subseteq Y$.
Es ist auch leicht zu sehen, dass $Y \subseteq X$ nicht gilt, da Y den Wert “3D Computer Graphics” annehmen kann, dieser jedoch nicht in X auftaucht.

Beispiel für n-ary Inclusion Dependencies

Student

Name	Lecture	Credit	Semester	Verified
Miller	DBS 1	20	2	false
Miller	PT 1	15	2	false
Smith	DPDC	10	6	true

Student, Course ⊆ Name, Lecture

Lending

ID	Name	Location	Student	Course
42	Database Systems	A-1.2	Miller	DBS 1
88	Database Systems	B-2.2	Miller	PT 1
73	Database Systems	A-1.2	Smith	DPDC
69	Algorithms in Java	C-E.1	Miller	PT 1
13	Algorithms in Java	C-E.1	Smith	DPDC

TODO quelle!

X := Attribute “Student” und “Course” aus Tabelle “Lending”

Y := Attribute “Name” und “Lecture” aus Tabelle “Student”

Bei n-ary Inclusion Dependencies ist es nicht nur wichtig das alle Werte der einzelnen Attribute aus X in Y auftauchen, sondern das sie vor allem in der Kombination in Y auftauchen, in der sie auch in X auftauchen.
Auch hier ist wieder einfach zu sehen, dass $X \subseteq Y$ gilt, denn die drei

unterschiedlichen Kombinationen aus “Student” und “Course” die in X auftauchen sind auch alle in Y vertreten. Das bedeutet also das hier ebenfalls $Y \subseteq X$ gelten würde.

7.2 Datenformat

Ein Datenset besteht aus mehreren **Tabellen**, die unterschiedliche Schematas haben können. Diese Tabellen werden als Stream eingelesen und einzelne Einträge eines Streams (= Zeilen einer Tabelle) können ältere Einträge überschreiben.

Wir konzipieren unseren Algorithmus so, dass er **Batches aus Änderungen** einliest. Ein Batch wird immer aus genau einem Input-Stream entnommen und hat das gleiche Schema wie seine Ursprungstabelle, bis auf eine $ Spalte am Anfang welche die *Position eines Eintrages* beschreibt.

Änderungen lassen sich in drei Arten unterteilen:

1. Eine **Hinzufügung** ist ein Änderung, deren Position das Erste mal im Stream auftaucht und bei der *alle Felder* einen Wert haben.
2. Eine **Modifikation** ist ein Änderung, deren Position bereits im Stream auftauchte und bei der *alle Felder* einen Wert haben. Die Position muss dem Eintrag entsprechen, der überschrieben werden soll.
3. Eine **Löschung** ist ein Änderung, deren Position bereits im Stream auftauchte und bei der *kein Feld* einen Wert hat. Die Position muss dem Eintrag entsprechen, der gelöscht werden soll.

Tabelle: Beispiel für eine Hinzufügung, Modifikation und Löschung eines Eintrags.

$	A	B	C
200	horse	lion	flamingo
200	horse	lion	**parrot**
200			

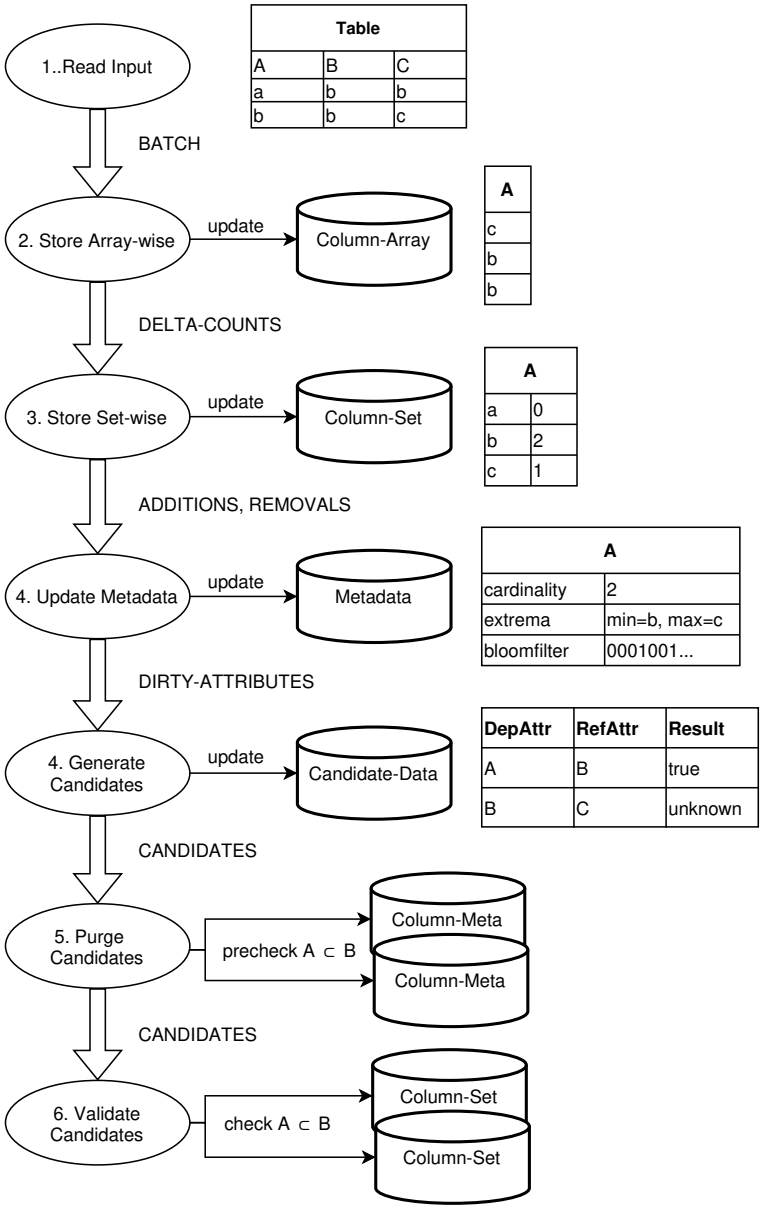
Wir definieren den leeren Zellenwert `NULL` als einen besonderen Marker, der die Abwesenheit eines Wertes beschreiben soll. Der `NULL` Marker muss immer von der Berechnung von Inclusion Dependencies ausgeschlossen werden - also ob ein Attribut fehlen kann oder nicht soll keine Auswirkung auf die gefundenen Inclusion Dependencies haben.

7.3 System Architektur (Datenfluss)

Bevor wir das System mit Akka Aktoren beschreiben, definieren wir den grundlegenden Datenfluss den wir damit umsetzen möchten. Dieser Datenfluss muss wiederholt-ausführbar sein und mit inkrementellen Updates (Batches) arbeiten.

Wir möchten pro eingelesenes Batch möglichst wenig Operationen durchführen. Die wohl teuerste Operation ist der *Subset-Check* für Validieren eines IND-Kandidaten. Hierbei werden alle Werte zweier Attribute abgefragt und verglichen.

Unser Ziel ist es also einen Datenfluss zu definieren, der es uns erlaubt möglichst wenige Subset-Checks (oder andere teure Operationen) durchzuführen.



1. Read Input

Es wird ein Batches von einer Quelle eingelesen. Das Format von Batches ist in der Sektion Datenformat beschrieben.

2. Write Array-wise

Ein Batch wird nach seinen Attributen aufgespalten und für jedes Attribut werden die Werte in ein eigenes *Column-Array* geschrieben. Ein Column-Array ist ein Array welches alle Werte eines Attributes an ihrer jeweiligen Positionen beinhaltet.

Anschließend werden die *Delta-Counts* berechnet. Diese beschreiben, wie häufig ein Wert eines Attributes hinzugefügt oder entfernt wurde.

Sollten alle Delta-Counts 0 sein, so haben die Änderungen des Batches definitiv keinen Einfluss auf Inclusion Dependencies und der Datenfluss kann vorzeitig enden.

3. Write Set-wise

Die Delta-Counts eines Attributs werden in das dem Attribut zugehörigen *Column-Set* geschrieben. Ein Column-Set ist ein zählendes Set, welches mitzählt wie häufig eine Ausprägung eines Wertes in einem Column-Array auftaucht.

Beim Schreiben der Delta-Counts wird ein *Set-Diff* erstellt. Dieses beschreibt, ähnlich dem Diff-Format des populären `diff` UNIX Tools, welche neuen Ausprägungen hinzugefügt oder entfernt wurden.

Fällt der Zähler von > 0 auf 0, so können wir feststellen, dass eine Ausprägung nicht mehr vorkommt (*entfernt wurde*). Gab es vorher keinen Zähler oder steigt der Zähler von 0 auf > 0, so so können wir feststellen, dass eine neue Ausprägung hinzugefügt wurde.

Sollten alle Set-Diffs leer sein - also keine Ausprägungen hinzugefügt oder verändert worden sein - so haben die Änderungen keinen Einfluss auf die INDs und der Datenfluss kann vorzeitig enden.

4. Update Metadata

Die Set-Diffs werden benutzt, um *Metadata* der dazugehörigen Attribute zu erstellen und zu aktualisieren.

Mehr zu den verschiedenen Arten von Metadata im Kapitel (TODO verlinkte).

5. Generate Candidates

Die Set-Diffs werden benutzt, um die *Candidate-Data* aller involvierten Attribute zu erstellen und zu aktualisieren.

Für alle neuen Attribute, die bisher nicht vorkamen, werden alle möglichen neuen (unären) Kandidaten generiert. Bereits-existierende Inclusion Dependencies, die sich geändert haben könnten, werden zurückgesetzt und neue Kandidaten generiert.

6. Purge Candidates

Die generierten Kandidaten werden anhand von *Subset-Prechecks* gefiltert. Ein Kandidat $A \subset B$ wird nur weiter verwendet, wenn die Metadata von A und B diese Subset-Relation erlaubt.

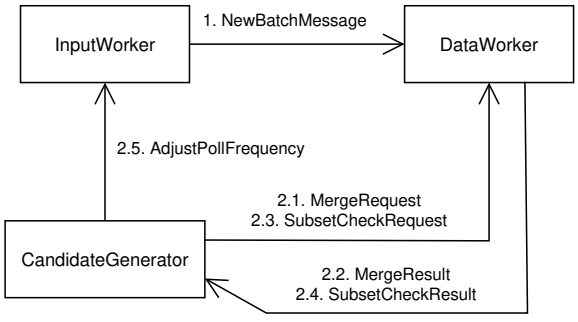
Mehr zu den verschiedenen Arten von Metadata im Kapitel (TODO verlinkte).

7. Validate Candidates

Die verbliebenen Kandidaten werden anhand von *Subset-Checks* validiert. Dabei müssen die Werte aus mehreren Column-Sets verglichen werden.

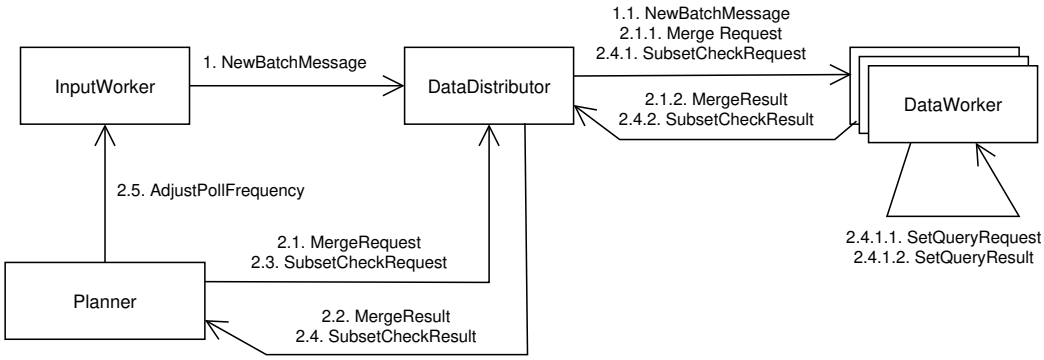
Die Ergebnisse werden anschließend in der Candidate-Data gespeichert und für subsequente Candidate-Generation benutzt.

7.4 System Architektur (1 System)



Kommunikationsdiagramm für die versimpelte 1-System Architektur

7.5 System Architektur (n Systeme)



Kommunikationsdiagramm für die verteilte Multi-System Architektur

8 Funktionale Anforderungen

8.1 Datengenerator

Der Datengenerator soll einen beliebig großen Batch einer beliebigen CSV Dateien generieren. In diesem Batch sollen anschließend mittels des dynamischen Algorithmus Inclusion Dependencys gefunden und ausgegeben werden.

Der Datengenerator soll…

- eine beliebige CSV-Datei einlesen.
- mehrere Batches im CSV-Format auf der Kommandozeile.
- jede Zeile mit einem eindeutigen Index versehen.
- eine bestimmte Anzahl an Zeilen generieren können.
- unendlich viele Zeilen durch Cycling generieren können (wieder von Vorne beginnen, sollte das Ende der CSV-Datei erreicht sein aber noch nicht die gewünschte Anzahl Zeilen).
- eine Zeile mit Wahrscheinlichkeit x löschen.

8.2 Dynamischer Algorithmus

Der dynamische Algorithmus soll…

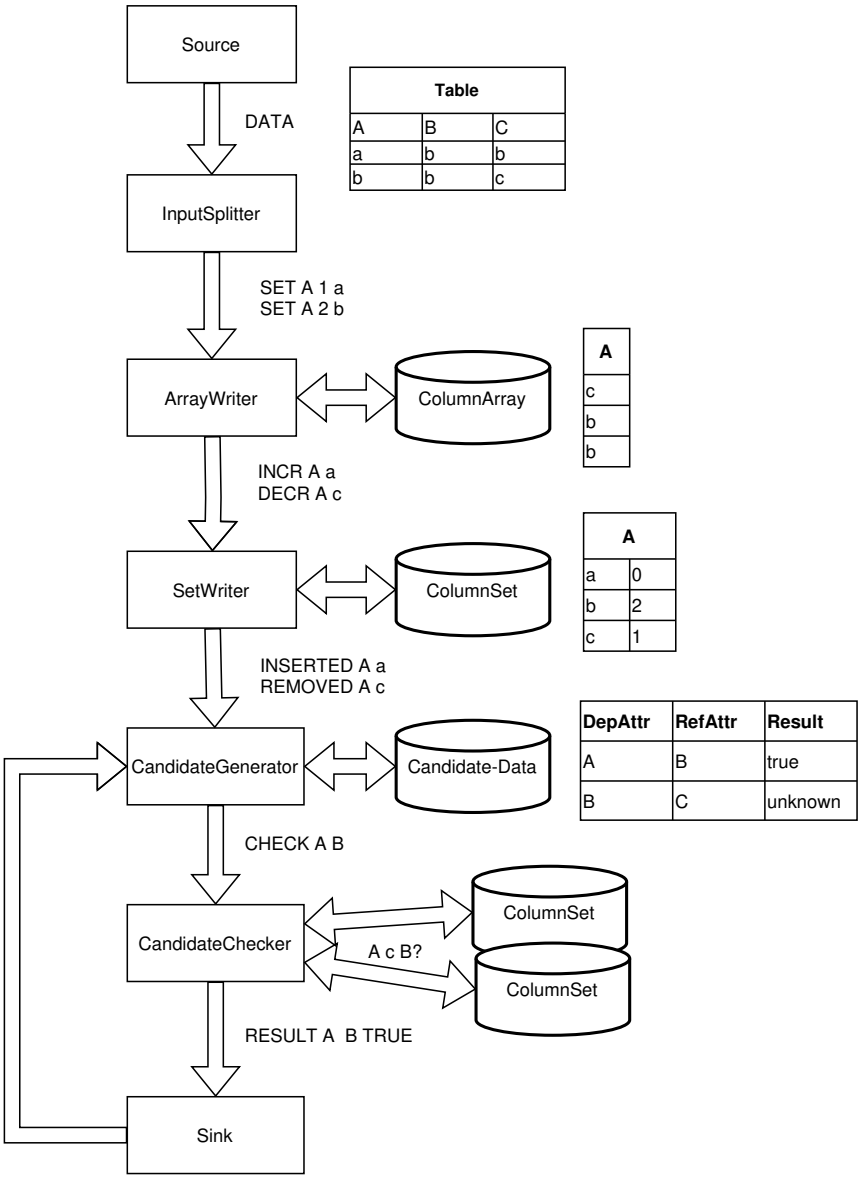
- für Hinzufügungen neue Einträge anlegen und Inclusion Dependencies finden.

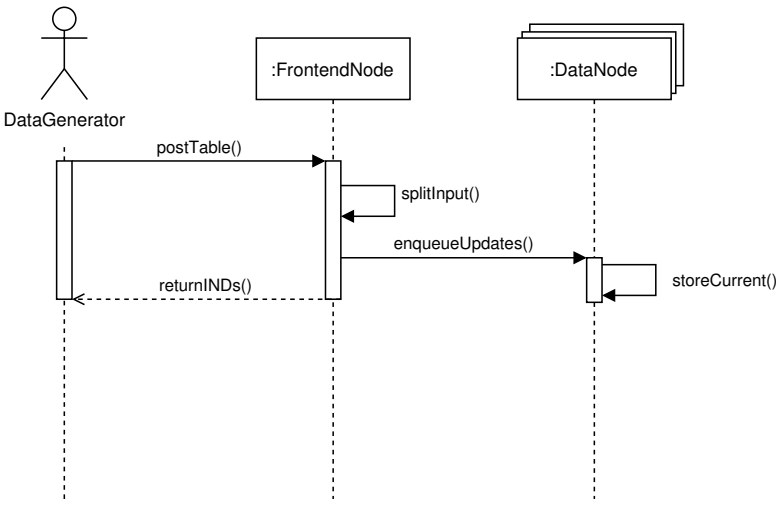
- für Modifikationen und Löschungen alte Einträge und dazugehörige Inclusion Dependencies updaten.
- alle X Sekunden gültige und nicht-mehr gültige Inclusion Dependencies ausgeben.
- auch mit großen Datensätzen zurecht kommen können.

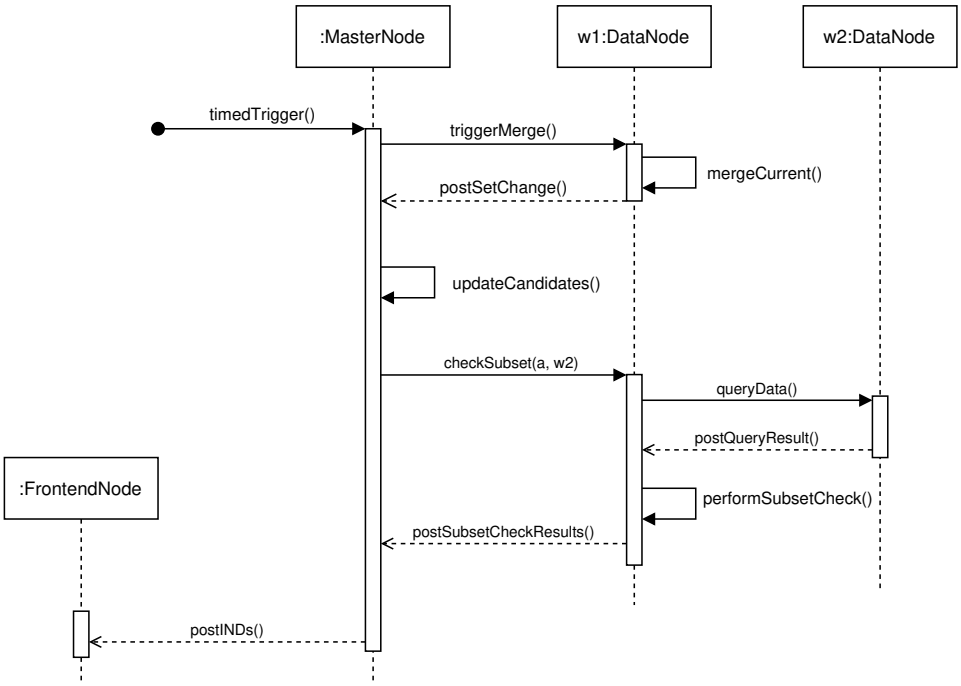
9 System-Entwurf

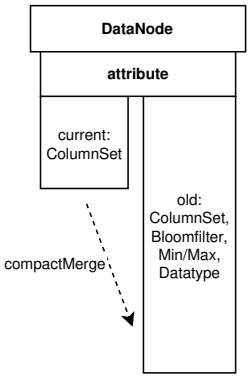
Felix

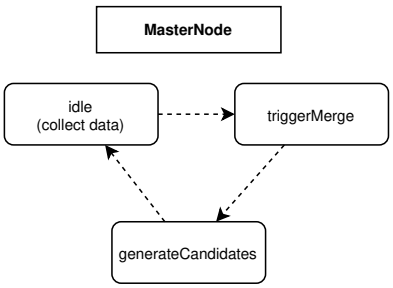
9.1 Überblick


Source
DATA
InputSplitter
SET A 1 a
SET A 2 b
ArrayWriter
ColumnArray
INCR A a
DECR A c
SetWriter
ColumnSet
INSERTED A a
REMOVED A c
CandidateGenerator
Candidate-Data
CHECK A B
CandidateChecker
A c B?
ColumnSet
ColumnSet
RESULT A B TRUE
Sink
Table
A B C
a b b
b b c
A
c
b
b
A
a 0
b 2
c 1
DepAttr RefAttr Result
A B true
B C unknown


DataGenerator
:FrontendNode
:DataNode
postTable()
splitInput()
enqueueUpdates()
returnINDs()
storeCurrent()







9.2 Value Representation

9.2.1 Hashing Long Values

Für lange Values kann stattdessen nur ein Hash gespeichert werden. Dadurch wird Speicher und Netzwerklast eingespart.

```
"foo" => "foo"
"bar" => "bar"
"Lorem ipsum {...}" => $124$cb24d439cebabab24
```

Indem wir mit dem Hash die Quell-Länge speichern (`${LEN}${HASH}`), erhöhen wir die Kollisionsresistenz noch ein wenig. Weiter könnte die Länge noch für die Single-Column-Analysis hilfreich sein.

9.2.2 Faster Hash Algorithm

Java's Builtin Hashing (4 byte) ist ob der hohen Kollisionsgefahr ungeeignet für Datenmengen unserer Größe.

Neben Algorithmen der SHA-Familie könnten wir auch xxHash (https://github.com/Cyan4973/xxHash) oder MurmurHash (https://en.wikipedia.org/wiki/MurmurHash) verwenden.

9.2.3 Byte Array Values

Statt Java's Builtin `String` Klasse, die mit ihren eigenen Problemen kommt (potentiell UTF-16 sowie Klassenoverhead), können wir Values im UTF-8 Format als `byte[]` behandeln.

9.3 Smart Candidate Generation

9.3.1 Elimination-by-Implication

Wenn bereits Kandidaten geprüft wurden, können die Ergebnisse genutzt werden, andere Kandidaten direkt auszuschließen.

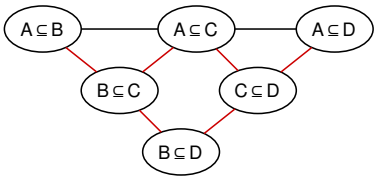
```
A c B  /\    B c D  ->   A c D
A c B  /\  !(A c D) -> !(B c D)
```

9.3.2 Candidate Picking

Statt dass sofort alle Kandidaten generiert und geprüft werden, wird nur eine bestimmte Anzahl von Candidaten generiert, um von den Prüfungs-Ergebnissen nutzen zu machen.

Die gewählten Kandidaten können zufällig sein oder bewusst gewählt, um die potentielle Nützlichkeit der Ergebnisse zu erhöhen.

Im Idealfall könnten z.B. drei Candidate-Checks zwischen vier Attributen dazu führen, dass man drei andere Candidate-Checks eliminieren kann.



9.3.3 Candidate Flagging

Nicht immer, wenn sich ein Column-Set verändert hat, müssen alle assoziierte Candidate-Checks neu ausgeführt werden.

- Counterexamples

9.4 Single-Column-Analysis Prechecking

Wenn wir bestimmte Eingenschaften einer Column kennen, können wir für einen Candidate-Check vorzeitig ein True-Negative zurückliefern.

- Distinct Value Count
- Datatype (Data Domain)
- Bloomfilter
- Minima/Maxima
- Column-Bytesum

Fraglich ist, wo dieser Filter angebracht werden sollte - vor oder nach der Candidate-Generation. Davor: Candidaten können früher eliminiert werden. Danach: Möglicherweise kostenspielig bei sehr vielen Attributen.

9.5 Optimierte Subset-Checks

9.5.1 Dirty-Ranges

Beim verändern von Werten eines Sets können dynamische Dirty-Ranges eingesetzt werden.

… (ähnlich wie Dirty-Flag, aber für eine Range)

9.5.2 Early-Return

Basierend auf den Distinct Value Counts kann die Iteration eines Subset-Check frühzeitig abgebrochen werden.

9.5.3 Bidirectional Check

Wenn `A c B` geprüft wird, können wir bei bedarf auch direkt `B c A` in einer Iteration prüfen.

10 Technologien

- **Java**: für verteiltes System und dynamischer Algorithmus
- **Apache Parquet**: Spaltenorientiertes Übertragungsformat
- **Python**: für Datengenerator
- **collectd** (https://collectd.org/): Linux-Daemon für System Performance Monitoring, erlaubt verteiltes Benchmarking von CPU/RAM-Nutzung, Netzwerklast, etc.

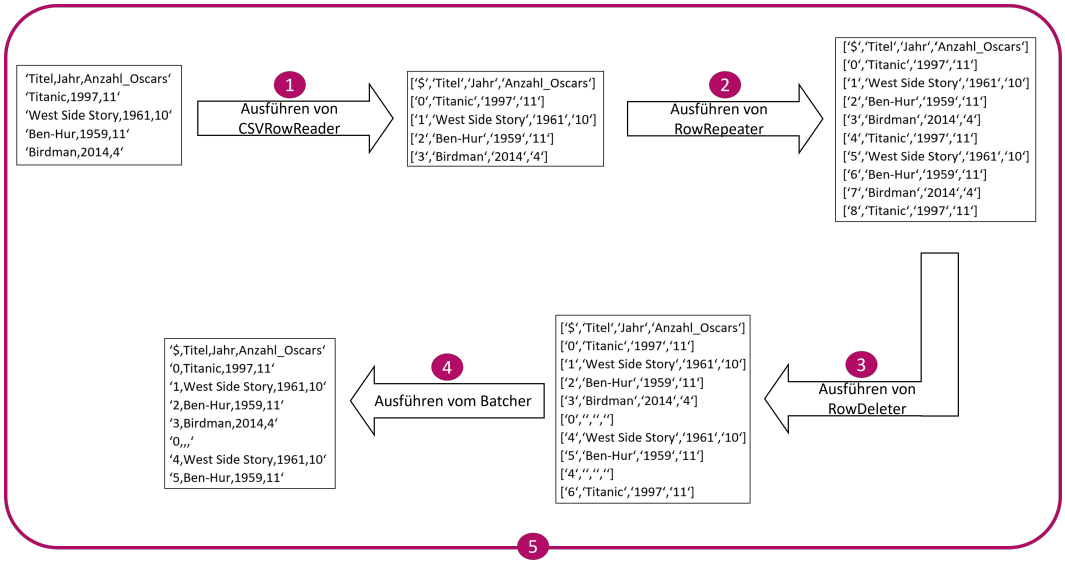
11 Detaillierter System-Entwurf

- Detail-Entwurf der Komponenten in UML-Diagrammen
    - Mindestens ein Klassendiagramm pro Komponente (mit Schwerpunkt auf den wichtigen Klassen wir persistenten Datenstrukturen und Services)
    - Für komplexere Abläufe bzw. komplexeres Verhalten jeweils ein Aktivitäts oder Zustandsdiagramm

- Entwurfsmuster
- Abhängigkeiten unter den Komponenten und zu externen Komponenten
- Datenmodell
- Verantwortlichkeiten der Entwickler
- allgemein Beschreiben wie der Datengenerator funktioniert
- für einzelen KOmponenten evtl. noch grapisch aufzeichnen und bisschen beschreiben was allgemein passiert

11.1 Datengenerator

Der Datengenerator besteht aus den fünf Klassen CSVRowReader, RowRepeater, RowDeleter, Batcher und CSVReadIn.



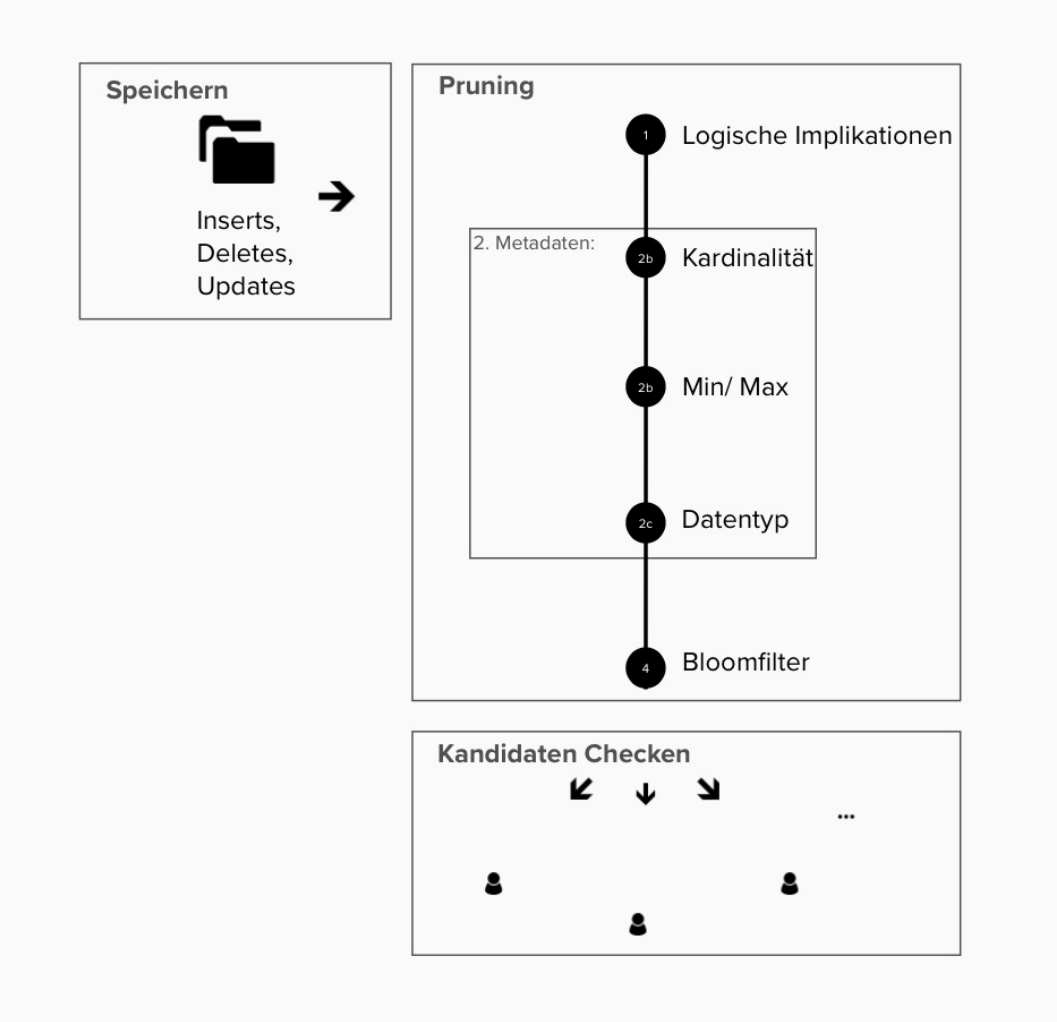
Beispielhafte Darstellung der einmaligen Ausführung des Datengenerators

Dem Datengenerator wird die Adresse einer CSV-Datei und die Anzahl an Reihen die insgesamt ausgegeben werden sollen, übergeben. Der Generator nimmt diese CSV-Datei und generiert daraus einen Batch *(5)*.
Dafür wird zunächchst die Datei eingelesen, wobei jede Spalte in ein Array das die Einträge der Spalte als String enthält, umgewandelt und Fortlaufend durchnummeriert *(1)*.
Diese Zeilen-Arrays werden jetzt so lange von vorne nach hinten wiederholt bis die übergebene Anzahl an gewünschten Reihen erreicht ist *(2)*. Dabei wird vor hinzufügen jedes neuen Zeilen-Arrays, eine vorherige Zeile mit einer

Wahrscheinlichkeit von 10% gelöscht. Dafür wird eine Array mit leerer Liste aber bekanntem Index hinzugefügt *(3)*.
Wenn die Anzahl an gewünschten Reihen erreicht wurde, wird daraus der Batch generiert. Dafür wird jedes Array wieder in eine String umgewandelt und die einzelnen Attribut-Werte durch Kommas getrennt. Es wird also wieder eine CSV-Datei generiert *(4)*.

Verteiltes System (Felix)

12 Algorithmenentwurf



Die Inserts, Updates und Deletes werden zunächst gespeichert.

12.1 Pruning Pipeline

In der Pruningphase sollen durch Vorarbeit viele mögliche Permutationen für Inclusion Dependencies ausgeschlossen werden. Anstatt also, dass auf der gesamten Datenmenge nach Inclusion Dependencies gesucht wird, wird nur in den Attributen gesucht, in denen eine Abhängigkeit überhaupt in Frage kommt.

(Im Status Quo suchen wir lediglich nach unären Inclusion Dependencies. Als Fortführung könnte man nach n-ären Inclusion Dependencies suchen.)

In einer Pipeline werden nacheinander durch verschiedene Prüfungen Permutationen ausgeschlossen.

12.1.1 Pruning durch logische Implikation

Durch logische Implikationen können Permutationen ausgeschlossen werden. Dafür werden zum Teil in vorherigen Iterationen Metadaten zu Permutationen gespeichert. Die Logischen Implikationen sind zum Beispiel:

12.1.2 Pruning durch Metadata

Aus den Metadaten der Attribute kann man auch Permutationen ausschließen. Durch eine Single Column Analysis erhalten wir verschieden Metadaten.

A	B	C
1	Mars	Luxemburg
2	Jupiter	Singapur
3	Jupiter	Lichtenstein
4	Luna	Singapur

X	Y	Z
10	Mars	Berlin
20	Mars	Berlin
30	Luna	Berlin

Innerhalb einer Tabelle ist num_rows für jede Spalte gleich. Über Tabellen hinweg darf es verschieden sein.

Eine `uniqueness` von `1.0` bedeutet, alle Werte einer Spalte sind unterschiedlich.

Eine `uniqueness` von `0.0` bedeutet, alle Werte einer Spalte sind gleich.

`datatype`	Gemeinsamer Datentyp für alle Werte einer Spalte	`datatype(A) = UnsignedInteger` `datatype(B) = String`
`highest_number` `lowest_number`	Höchster Zahlenwert einer Spalte Niedrigster Zahlenwert einer Spalte	`highest_number(X) = 30` `lowest_number(X) = 10`

`max_string_length`
`min_string_length`
	Maximale Länge eines Werts betrachtet als String.
	Minimale Länge eines Werts betrachtet als String.
		`max_string_length(A) = 1`
		`min_string_length(A) = 1`
		`max_string_length(B) = 7`
		`min_string_length(B) = 4`

Mögliche Datentypen:

- `UnsignedInteger`: 1, 2, 42, 35666
- `Integer`: -10, 0, 10, 20000
- `Real`: 1, 2.0, -1.0e-7
- `Timestamp`: z.B. 2012-12-01 10:00:30
- `String`: the above and anything else, including this sentence

Datentypen können andere Datentypen enthalten:

$\texttt{UnsignedInteger} \subset \texttt{Integer} \subset \texttt{Real} \subset \texttt{String}$ $\texttt{Timestamp} \subset \texttt{String}$ (diese Datentypen sind nur ein Vorschlag - cool wäre es, wenn wir eine Auswahl hätten, die wir auch anhand von Papers/Statistiken begründen können! Wir könnten auch Charakterklassen betrachten, z.B. Numeric, Alphabetic, ASCII, Unicode…)

12.1.2.1 Kardinalitäten

A	B
chihuahua	dog
chihuahua	dog
dropbear	horse
elephant	cat
dugong	cat

$\texttt{num\_distinct\_values(A)}=5$ $\texttt{num\_distinct\_values(B)}=3$ => $A \not\subset B$

Wenn A mehr einzigartige Werte als B hat, dann kann A nicht in B enthalten sein. Somit muss eine Inclusion Dependency von A in B nur überprüft werden, wenn `num_distinct_values(A)<num_distinct_values(B)` oder `num_distinct_values(A)=num_distinct_values(B)`. Nicht aber wenn `num_distinct_values(A)>num_distinct_values(B)`. Wenn A keine Inclusion Dependency von B: A erhält ein neues Input und B bleibt gleich, dann kann A immer noch kein Inclusion Dependency sein.

12.1.2.2 Min-/Max-Werte

Für die Extremwerte in einem Attribut kann man überprüfen ob eine Inclusion Dependency besteht.

Wenn der Maxwert von A größer ist als der Maxwert von B, so enthält A Werte die es nicht in B gibt, also kann A nicht in B enthalten sein, B aber in A.

Wenn der Minwert in A kleiner ist als in B, kann A nicht in B enthalten sein, B aber in A.

Somit können bei allen Kombinationen von Inclusion Dependencies die Min- und Maxwerte überprüft werden.

12.1.2.3 Datentyp

12.1.3 Pruning durch Sketches

12.1.3.1 Bloom Filter

Ein weiterer Ausschluss findet durch Nutzung von Bloom Filtern statt. Genutzt wird ein Counting-Bloomfilter mit einer Größe von 128 und zwei Hash-Funktionen.

Bloomfilter sind eine probabilistische Datenstruktur, die Daten repräsentieren. Ein Bloom Filter ist ein Array aus `m` Bits, die ein Set aus `n` Elementen repräsentiert. ZU Beginn sind alle Bits auf `0`. Für jedes Element im Set werden nun `k` Hashfunktionen ausgeführt, die ein Element auf eine Nummer zwischen `1` bis `m` mappen. Jede dieser Positionen im Array werden dann auf `1` gesetzt. Will man nun prüfen ob ein Element in einer Datenmenge enthalten ist, kann man die Werte berechnen und prüfen ob die Positionen auf `1` sind. Wegen Kollisionen kann das Verfahren zu False Positives führen, allerdings nicht zu False Negatives. Wenn ein Element im Array `0` ist, so wurde der Wert definitiv noch nicht gesehen.

Counting Bloomfilter ergänzen Bloomfilter dahingehend, dass nun mitgezählt wie oft ein Bit im Array auf `1` gesetzt wird. Das ermöglicht auch Elemente zu löschen.

https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.457.4228&rep=rep1&type=pdf

12.1.4 Kandidaten üperprüfen

Nachdem wir in der Vorarbeit die Anzahl an Attributen, die wir auf Inclusion Dependencies überprüfen so weit wie möglich reduziert haben, werden nun die übrig gebliebenen Permutationen überprüft. Erst jetzt werden die gespeicherten Tabellen abgerufen um die relevanten Spalten miteinander zu vergleichen.

Hierbei betreiben wir ebenfalls eine Optimierung. Wenn eine gewisse Anzahl an Werten in beiden Attributen untersucht wurde, und die Anzahl verbliebener Werte nicht mehr ausreicht um noch eine Inclusion Dependency zu ergebn, brechen wir ab.

Beispiel:

> A hat 100 einzigarte Werte, B hat 80 einzigartige Werte: Wenn in den ersten 21 Werten von A kein einziger Wert von B auftaucht, so kann B nicht mehr vollständig in A enthalten sein. Hier kann bereits abgebrochen werden.

13 Optimierungen (Benchmarks?)

Felix

14 Benutzerdokumentation

(später, Readme.md(

15 Entwicklerdokumentation

verlinken! github repo

16 Projektdokumentation

Generierung des PDF Dokuments:

```
pandoc *.md  -t html --pdf-engine-opt=--enable-local-file-access -o
        full.pdf  -f markdown+implicit_figures --toc
```