

# Dynamische Detektion von Inclusion Dependencies

Helen Brüggmann (brueggmh@students.uni-marburg.de)

Felix Köpge (koepge@students.uni-marburg.de)

Ragna Solterbeck (solterbe@students.uni-marburg.de)

10-10-2022

## Inhaltsverzeichnis

- 1 Beschreibung
  - 1.1 Projekt
    - 1.1.1 Die Ausgangslage
    - 1.1.2 Projektvorhaben
    - 1.1.3 Projektidee
  - 1.2 Auftraggeber
  - 1.3 Qualitätsanforderungen
- 2 Zusammenfassung
  - 2.1 Evaluation
- 3 Erfahrungsbericht
- 4 Entwicklungsprozess
  - 4.1 Dynamischer Algorithmus
  - 4.2 Datengenerator
- 5 Team
- 6 Zeitplan
  - Phase 1: Recherchephase 24.10.2021 - 18.01.2021
  - Phase 2: Konzeptionierung und Umsetzung 04.04.2022 - 08.08.2022
    - a) Data-Generator
    - b) Dynamischer Algorithmus
  - Phase 3: Hausarbeit und Festhalten der Ergebnisse 01.08.2022-31.09.2022
- 7 Inclusion Dependencies
- 8 Definition des Zielsystems
  - 8.1 System-Kontext

- 8.2 Datenformat (Batches)
- 9 Funktionale Anforderungen
  - 9.1 Datengenerator
  - 9.2 Dynamischer Algorithmus
- 10 Algorithmenentwurf
  - 10.1 Datenfluss
  - 10.2 Pruning Pipeline
    - 1. Pruning durch logische Implikation
    - 2. Pruning durch Metadata
- 11 System-Entwurf
  - 11.1 Datengenerator
  - 11.2 Single-Host Akka System
  - 11.3 Multi-Host Akka System
  - 11.4 Weiterentwicklung
    - 11.4.1 Logische Implikationen
    - 11.4.2 Parallelisierung der Dateneinlese und -verteilung
    - 11.4.3 Partitionierung anhand des initialen Batches
    - 11.4.4 Horizontale Partitionierung
    - 11.4.5 Parallelisierung des Pruning
  - 11.5 Erweiterung der Aufgabenstellung
- 12 Benutzerdokumentation
  - 12.1 Datengenerator
  - 12.2 Akka-System
- 13 Entwicklerdokumentation
  - 13.1 Projektstruktur
  - 13.2 Testen

# 1 Beschreibung

## 1.1 Projekt

NAME DES PROJEKTS: Dynamische Detektion von Inclusion Dependencies

STARTTERMIN: 24.10.2021

ENDTERMIN: 10.10.2022

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 (FD's)** oder **Inclusion Dependencies (IND's)**. 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 zeit- und arbeitsaufwendig.

### 1.1.2 Projektvorhaben

Im Rahmen dieser Arbeit soll ein dynamischer Data-Profiling Algorithmus entwickelt werden, der IND's auf dynamische Datenmengen fortlaufend entdeckt. Er soll alle IND's entdecken, aber auch beim Einfügen oder Löschen von Einträgen überprüfen, ob dadurch IND's 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 IND's entstanden sind oder welche IND's 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* IND's 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 IND's in dynamisch wachsenden Datensets implementiert. Außerdem wurde ein Datengenerator erstellt, mit dem wir die Generierung von dynamischen Daten simulieren. Diese wachsenden Daten werden genutzt um darauf INDs zu suchen.

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 Datensätzen und das Generieren von IND-Kandidaten ist noch auf einen einzelnen Master-Node beschränkt.

Der Datengenerator und das verteilte System lassen sich unabhängig voneinander ausführen, dazu sind zahlreiche Konfigurationsmöglichkeiten gegeben (siehe [Benutzerdokumentation](#)). Wir prüfen die Ergebnisse unserer Lösung gegen ein alte Version unserer Code-Base (siehe [Entwicklerdokumentation - Testen](#)). Die Performance unserer Lösung wird weiterhin verglichen zu einer Spark-Implementation des SINDY-Algorithmus (siehe [Evaluation](#)).

**Fazit unserer Evaluation** ist, dass unsere Lösung bei statische Datensätzen schlechter performt als der SINDY-Algorithmus. Bei dynamische Datensätze hingegen degradiert unsere Performance nur wenig (~10% Durchsatz-Degradation bei 50% Löschungen). Vergrößern wir unsere Datensätze, so verbessert sich der Durchsatz - scheinbar steigen unsere Ausführungszeiten logarithmisch mit der Input-Größe statt linear. Dies müsste man aber noch weiter auf leistungsstärkeren Host-System erproben.

### 2.1 Evaluation

Wir haben verschiedene Konfigurationen unserer Lösungen auf dem TPC-H Datensatz ausgeführt und die Ergebnisse visualisiert.

Die **Live-Results** zeigen, wieviele IND-Kandidaten zu jedem Zeitpunkt (neu-)generiert wurden und wie sie behandelt wurden. Um mehr über die Pruning-Pipeline und verschiedenen Pruning-Methoden zu lernen, siehe [Algorithmenentwurf - Pruning Pipeline](#)

Die **System-Benchmarks** zeigen die CPU- und RAM-Auslastung des Host-Systems während einer Ausführung. Alle Ausführungen fanden auf einem **Intel(R) Core(TM) i7-1165G7 (2.8GHz Clockspeed, 4 Cores, 8 Threads)** mit **16GB DDR4 RAM** statt.

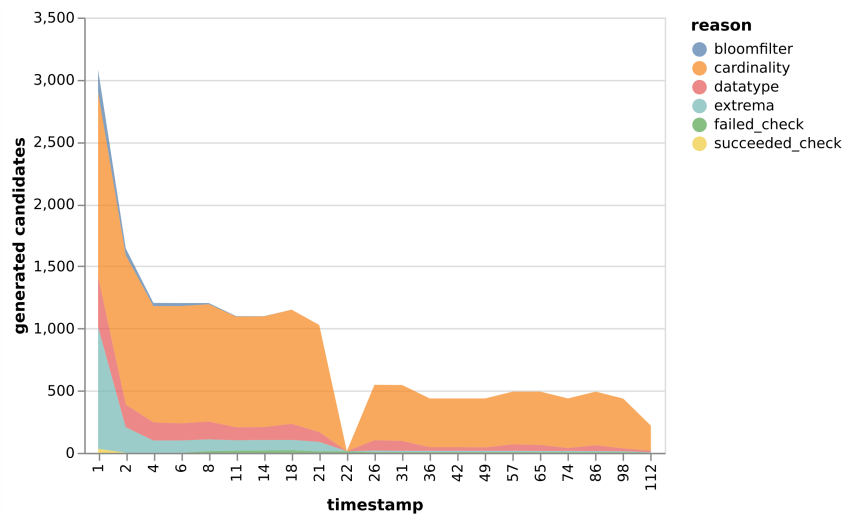
Die **Ausführungszeiten** wurden zusätzlich mit dem UNIX `time` Tool gemessen. *Real-Time* beschreibt die Ausführungszeit von Anfang bis Ende. *User-Time* beschreibt die CPU-Zeit über mehrere Threads/Prozesse hinweg.

Tabelle: Überblick der verschiedenen System-Konfigurationen und einmalig-gemessene Ausführungszeiten (UNIX time).

Konfiguration	Erklärung	Input-Größe	Real-Time (secs)	User-Time (secs)	Durchsatz (MB/s)
TPCH-static	Statischer Datensatz	415MB	116	447	3.57
TPCH-spark	Statischer Datensatz, Spark Implementation	415MB	68	408	6.1
TPCH-del10	Dynamischer Datensatz, 10% Löschungen	415MB	122	503	3.4
TPCH-del50	Dynamischer Datensatz, 50% Löschungen	415MB	129	502	3.2
TPCH-repeat300-del10	Dynamischer Datensatz mit Wiederholung bis 300MB pro Datei, 10% Löschungen	2.1GB	305	1682	6.85
TPCH-mutate300-del10	Dynamischer Datensatz mit Mutation bis 300MB pro Datei, 10% Löschungen	2.1GB	471	3142	4.46

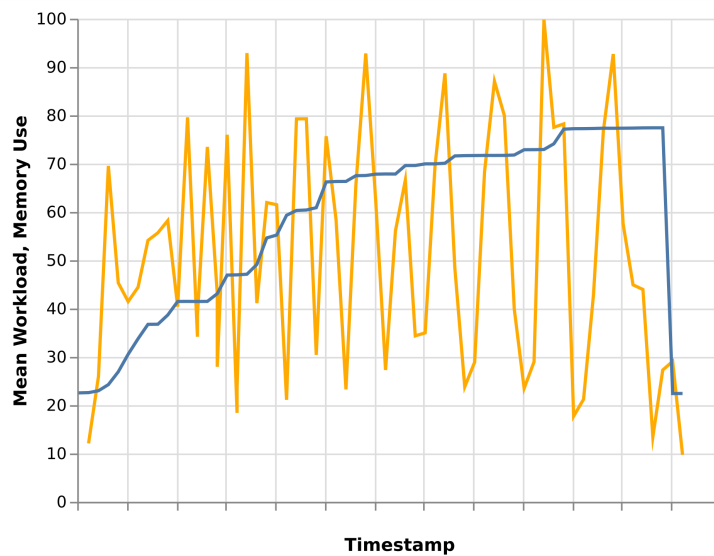
#### TPCH-static (Statischer Datensatz)

Im statischen Fall lesen wir den Datensatz einmal ohne Löschungen ein. Der Durchsatz in diesem Fall liegt bei 3.57MB/s.



Live-Results für TPCH-static.

Zu Beginn wurden 3000 Kandidaten generiert, wovon die allermeisten geprunt werden konnten. In nachfolgenden Durchläufen wurden typischerweise 50-25% der Kandidaten re-generiert und auch in den allermeisten Fällen erfolgreich geprunt.



System-Benchmark für TPCH-static. Blau ist die RAM-Auslastung, Orange die CPU-Auslastung.

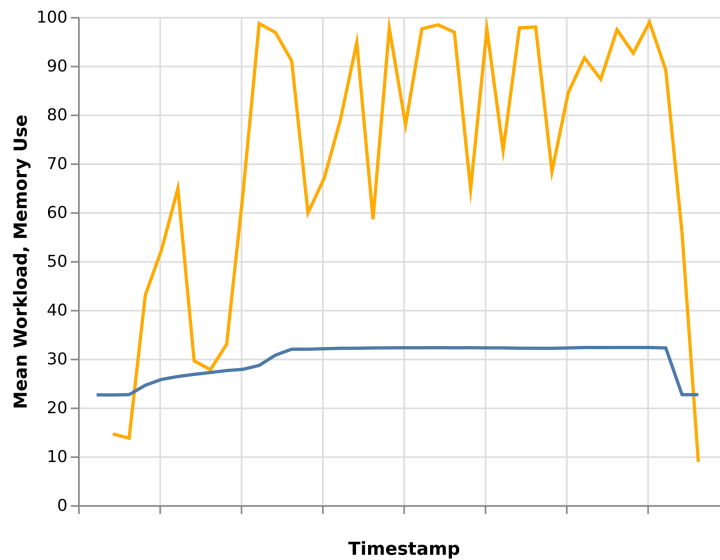
Die System-Benchmark zeigt, dass sowohl RAM als auch CPU besser ausgenutzt werden könnten. Der RAM-Auslastung steigt bis zu ~75% an (12GB), was für eine Input-Größe von 415MB nicht verhältnismäßig ist. Die CPU-Auslastung liegt im Mittel etwa bei 50%, mit zunehmend stärker-werdenden Schwingungen.

Die Text-Logs des Akka-Systems zeigten, dass die Ausführung häufig auf ein paar wenige Data-Worker warten musste. Wir vermuten, dass man mit zusätzlicher [horizontaler Partitionierung \(Aufteilung nach Zeilen\)](#) man eine bessere Auslastung erzielen würde.

### TPCH-spark (Statischer Datensatz, Spark Implementation)

Die Spark-Implementation des SINDY-Algorithmus zeigt im statischen Fall einen höheren

Durchsatz von 6.1MB/s (im Vergleich zu den 3.57/MBs unserer Lösung).



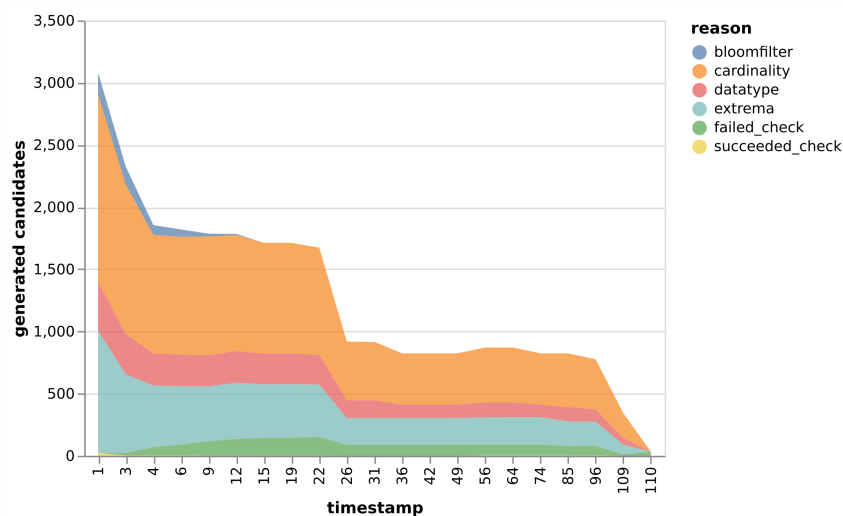
System-Benchmark für TPC-H-spark. Blau ist die RAM-Auslastung (von 16GB), Orange die CPU-Auslastung.

Die System-Benchmark zeigt eine bessere CPU-Auslastung, was vermutlich auf die hohe Verteilbarkeit des SINDY-Algorithmus und dem schlaun Partitionieren des Spark-Frameworks zurückzuführen ist.

Die RAM-Auslastung pendelt sich auf ~30% (5GB) ein. Wir vermuten, dass es darauf zurückzuführen ist, dass wir in unserer Lösung alle Daten zwischenspeichern und zusätzlich viel Daten-Redundanz haben (siehe [DataWorker](#)), während die Spark-Implementation Input-Daten und Zwischenergebnisse verwirft.

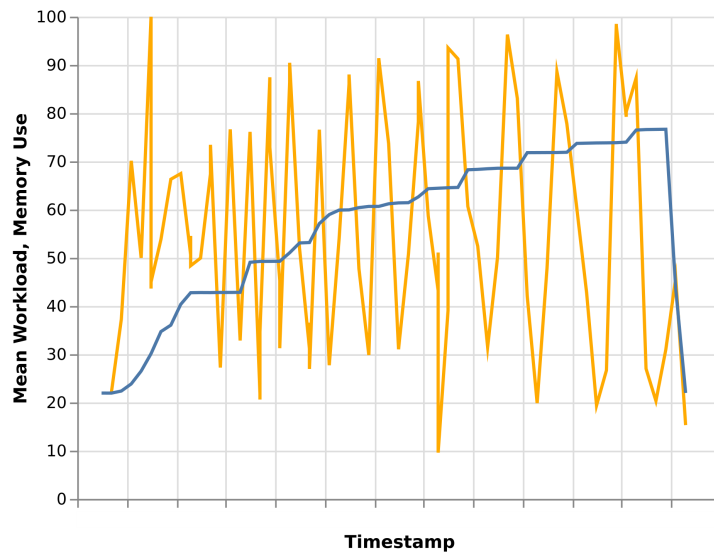
### TPCH-del10 (Dynamischer Datensatz, 10% Löschungen)

Löschen wir während der Ausführung 10% der Einträge, so scheint das nur wenig Auswirkungen auf den Durchsatz zu haben: von 3.57MB/s im statischen Fall (TPCH-static) zu 3.4MB/s im dynamischen Fall.



Live-Results für TPC-H-del10.

Die Live-Results zeigen, dass es häufiger zu gescheiterten Subset-Checks kam. Es scheint, dass die Pruning-Methoden hier weniger häufig greifen.

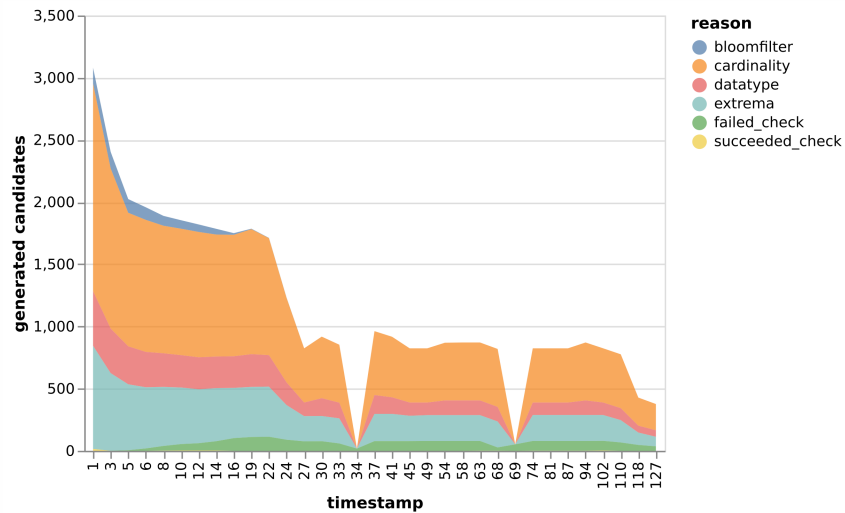


System-Benchmark für TPCH-del10. Blau ist die RAM-Auslastung (von 16GB), Orange die CPU-Auslastung

Die System-Benchmark zeigt eine ähnliche Auslastung zu TPCH-static.

### TPCH-del50 (Dynamischer Datensatz, 50% Löschungen)

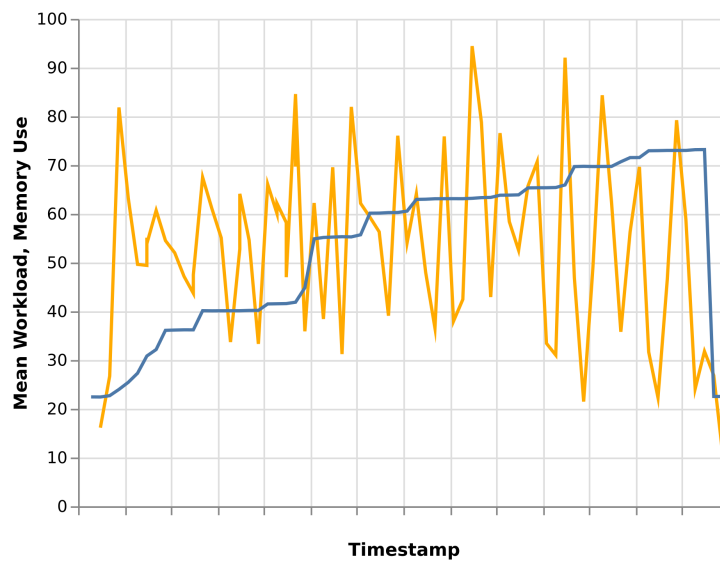
Löschen wir während der Ausführung 50% der Einträge, sinkt der Durchsatz von 3.57MB/s im statischen Fall (TPCH-static) zu 3.2MB/s im dynamischen Fall.



Live-Results für TPCH-del50

Die Live-Results sind ähnlich zu dem dynamischen Fall mit 10% Löschungen (TPCH-del10).





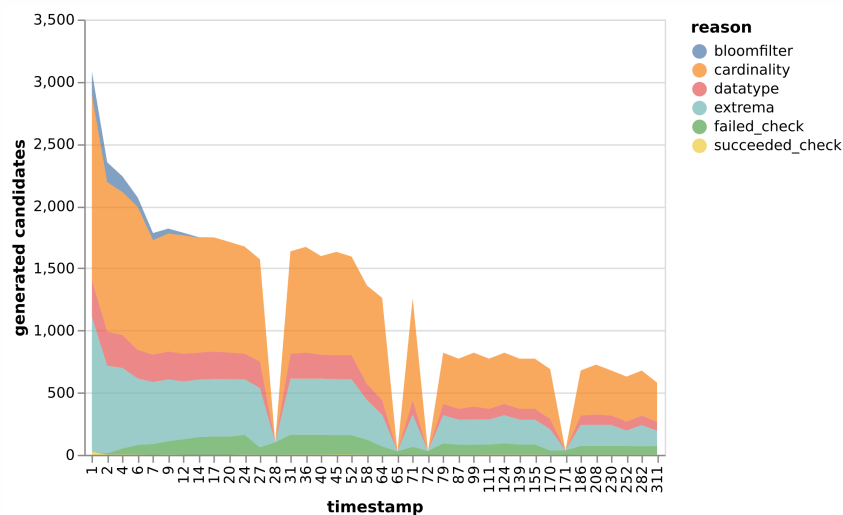
System-Benchmark für TPCH-del50. Blau ist die RAM-Auslastung (von 16GB), Orange die CPU-Auslastung

Die System-Benchmark zeigt eine ähnliche Auslastung zu TPCH-static und TPCH-del10.

### TPCH-repeat300-del10 (Dynamischer Datensatz mit Wiederholung bis 300MB pro Datei, 10% Löschungen)

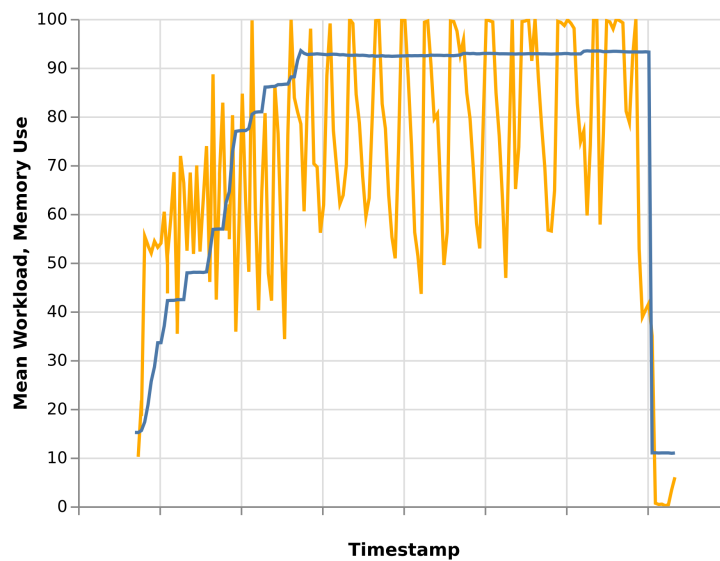
Mit "Wiederholen" beschreiben wir, dass wir jede CSV-Datei nach vollständigem Einlesen wieder von-vorne einlesen und mit neuen Zeilenpositionen ausgeben (siehe [Datengenerator](#)).

Wiederholen wir jede CSV-Datei bis zu 300MB pro Datei (2.1GB Input insgesamt), so zeigt dies eine starke Verbesserung im Durchsatz: von 3.4 MB/s im 415MB Fall (TPCH-del10) zu 6.1 MB/s im 2.1GB Fall. Das ist zu erwarten - bereits bekannte Attribut-Werte werden im [Datenfluss](#) schnell abgefangen und verursachen überhaupt keine Neu-Generation von Kandidaten.



Live-Results für TPCH-repeat300-del10

Die Live-Results sind sehr ähnlich zu TPCH-del10 über einen längeren Zeitraum.



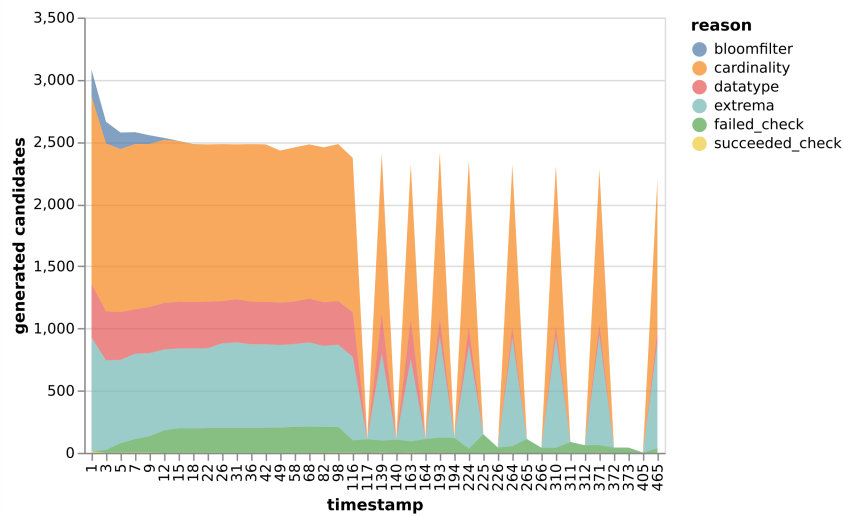
System-Benchmark für *TPCH-repeat300-del10*. Blau ist die RAM-Auslastung (von 16GB), Orange die CPU-Auslastung

Die System-Benchmark ähnelt *TPCH-del10* über einen längeren Zeitraum mit einer höheren Basis-Auslastung. Das Problem der fehlenden horizontalen Partitionierung (siehe *TPCH-static*) wird hier in immer-länger-werdenden Schwingungen der CPU-Auslastung umso deutlicher. Die maximale RAM-Auslastung von 95% (15.2GB) zeigt im Vergleich zum 415MB Fall (*TPCH-del10*) mit 70% (11.2GB), dass der RAM-Verbrauch logarithmisch steigt. Würde es linear steigen, so würde man ~57GB RAM benötigen.

#### **TPCH-mutate300-del10 (Dynamischer Datensatz mit Mutationen bis 300MB pro Datei, 10% Löschungen)**

Mit "Mutieren" beschreiben wir, dass wir jede CSV-Datei nach vollständigem Einlesen wieder von-vorne einlesen und mit neuen Zeilenpositionen ausgeben. Zusätzlich werden alle Werte so modifiziert, dass sie verschieden sind zu der vorherigem Durchlauf (siehe [Datengenerator](#)).

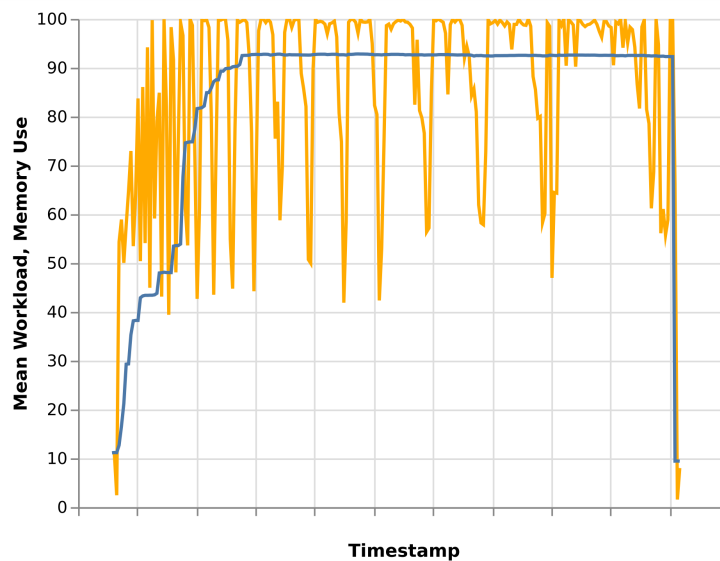
Mutieren wir jede CSV-Datei bis zu 300MB pro Datei (2.1GB Input insgesamt), so sinkt der Durchsatz von 6.1 MB/s im 2.1GB Fall mit Wiederholungen (*TPCH-repeat300-del10*) zu 4.46 MB/s im Fall mit Mutationen. Der Durchsatz ist aber immernoch höher als im vergleichbaren 415MB Fall (*TPCH-del10*) mit 3.4 MB/s.



Live-Results für TPC-H-modify300-del10

Die Live-Results zeigen, dass in der ersten Hälfte der Ausführung zu jedem Zeitpunkt etwa 2500 Kandidaten generiert wurden. Das macht Sinn - die Mutationen führen dazu, dass *alle* Attribute neue Werte erhalten und fortlaufend neu auf INDs geprüft werden müssen.

In der zweiten Hälfte der Ausführungen kommt es zu großen Lücken, in denen hauptsächlich nur gescheiterte Subset-Checks durchgeführt wurden. Wir vermuten, dass es hier zu sehr langwierigen Subset-Checks kam, auf die das System warten musste.



System-Benchmark für TPC-H-mutate300-del10. Blau ist die RAM-Auslastung (von 16GB), Orange die CPU-Auslastung

Die System-Benchmark zeigt eine ähnliche Auslastung wie im vergleichbaren Fall mit Wiederholungen (TPCH-repeat300-del10). Überrascht hat uns, dass sich auch die RAM-Auslastung von bis zu 95% sehr ähnelt. Wir hatten erwartet, dass die Ausführung wegen der modifizierten Werte mehr RAM beanspruchen würde, als wir zur Verfügung haben.

# 3 Erfahrungsbericht

Wir mussten feststellen, dass es besser ist mit existierenden Algorithmen zu arbeiten bevor man einen eigenen Algorithmus versucht zu implementieren. Außerdem wäre Spark ein deutlich besseres Framework für unsere Problemstellung gewesen wäre (wie es auch für die meisten OLAP Anwendungen ist).

## **Eigener Algorithmus vs Existierender Algorithmus**

Da wir sofort einen eigenen Algorithmus implementiert haben, hat uns für die Entwicklung eine Referenz-Implementierung gefehlt. Dadurch konnten wir nicht feststellen, ob unser System besser oder schlechter als existierende Lösungen performt.

Insgesamt hat uns auch sehr viel mehr Aufwand beschert, da wir häufig unseren Ansatz anpassen mussten, was jedesmal große und radikale Änderungen in der Codebase bedeutete. Hätten wir einen existierenden Algorithmus gewählt und diesen versucht weiter zu verbessern, hätten wir schnell bei jeder Iteration feststellen können, ob wir den Algorithmus verbessert oder verschlechter haben.

## **Akka vs Spark**

Wir haben Akka gewählt, da wir uns größere Kontrolle über Task-Abarbeitung und Datenverteilung gewünscht haben. In Anbetracht darauf, dass wir einen eigenen Algorithmus implementiert haben, hätten wir in Spark allerdings deutlich schneller arbeiten können.

In Akka überlegt man sich einen abstrakten Datenfluss (oder Algorithmus) und versucht diesen dann mit einem Aktoren-Protokoll zu implementieren. Das bedeutet auch, dass bei jeder Anpassung des Algorithmus auch das Aktoren-Protokoll angepasst werden muss. Wenn man also seinen Algorithmus falsch konzipiert, muss man möglicherweise große Teile seiner Arbeit verwerfen und neu implementieren.

In Spark hingegen kann man einen Algorithmus sehr natürlich in wenigen Zeilen implementieren. Schwerer wird es dann nur, die Implementierung weiter zu optimieren (wie z.B. die Partitionierung anzupassen). Spark ist damit allerdings deutlich besser geeignet für das Prototyping neuer Algorithmen.

# 4 Entwicklungsprozess

## 4.1 Dynamischer Algorithmus

Wir haben die Arbeit mit einer vorhandenen Akka Architektur von Prof. Papenbrock begonnen, die

uns vorher als Hausaufgabenprojekt für das Modul Distributed Data Management diene.

Die vorhandene Architektur war bereits verteilt und konnte IND's 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 konzipiert 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):**

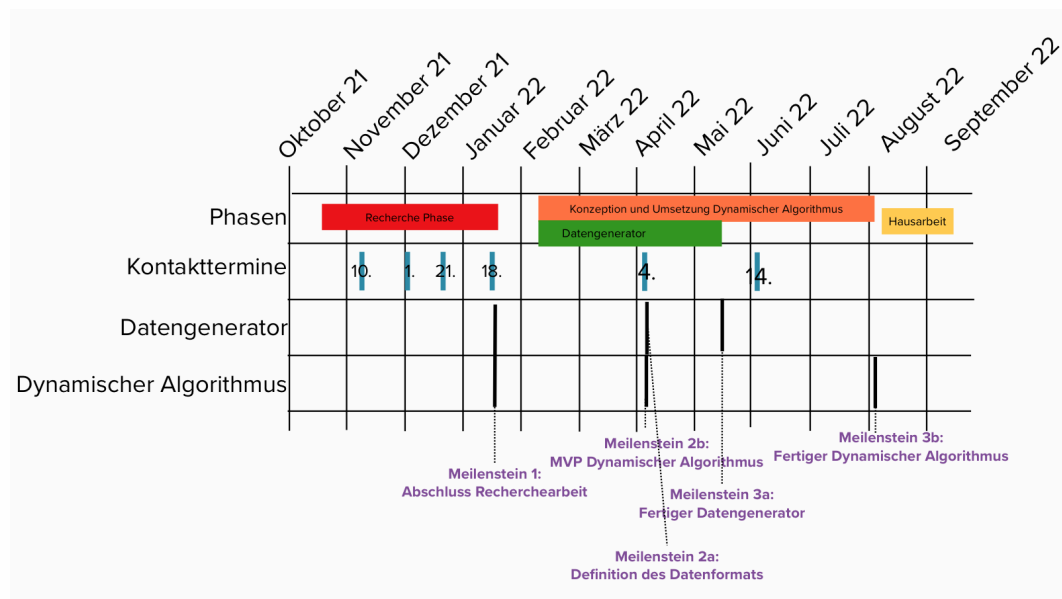
- Konzeption und Entwicklung des Datengenerators (Pairprogramming mit Felix Köpge)
- Erstellung der Auslastungsdiagramme

## 6 Zeitplan

**Projektzeitraum: 24.10.2021 - 31.09.2022**

**Projektphasen:**

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



## Phase 1: Recherchephase 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 Phase 1**

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**

### **04.04.2022 - 08.08.2022**

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

#### **a) Data-Generator**

##### **Erste Programmiereinheit: 04.04.2022 - 08.08.2022**

In einem einwöchigen Programmiersprint ist das Konzept und ein Großteil des Data-Generators 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**

In der zweiten Programmiereinheit wurde der Data-Generator fertiggestellt.

#### ***Meilenstein 3a: Fertiger Data-Generator***

##### **Ergebnis der Phase 2a**

Fertiger Data-Generator

## b) 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 des dynamischen Algorithmus 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 das Akka-System mitsamt seiner Pipeline final implementiert.



*Meilenstein 3b: Fertigstellung Dynamischer Algorithmus*



**Ergebnis der Phase 2b**



Fertiges Akka-System

## Phase 3: Hausarbeit und Festhalten der Ergebnisse 01.08.2022-31.09.2022

Parallel zur Fertigstellung des Akka-Systems wurde die Hausarbeit zu der Projektarbeit erstellt. Neben der Dokumentation wurden außerdem graphische Plots der Ergebnisse erstellt.



**Ergebnis der Phase 3**



Ausarbeitung und Darstellung der Ergebnisse

## 7 Inclusion Dependencies

Inclusion Dependencies (IND's) 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 stets gelten muss  $|X| = |Y|$ .

Es wird von *unary* IND's gesprochen wenn gilt  $X \subseteq Y$  mit  $|X| = |Y| = 1$ . Falls  $|X| = |Y| = n$  gilt, handelt es sich um eine *n-ary* IND.


Für IND's gelten immer folgende Eigenschaften:

- *Reflexiv*: Es gilt immer  $X \subseteq X$
- *Transitiv*: Es gilt  $X \subseteq Y \wedge Y \subseteq Z \implies X \subseteq Z$
- *Permutationen*: Es gilt  $(X_1, \dots, X_n) \subseteq (Y_1, \dots, Y_n)$ , dann gilt auch  $(X_1, \dots, X_n) \subseteq (Y_{\sigma(1)}, \dots, Y_{\sigma(n)})$  für alle Permutationen  $\sigma_1, \dots, \sigma_n$

### Beispiel für unary IND's

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  $\subseteq$  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

unary IND-Beispiel[1]

$X$  := 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 IND's

### Student

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

### 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

Student, Course  $\subseteq$  Name, Lecture

n-ary IND Beispiel[1]

$X$  := Attribute "Student" und "Course" aus Tabelle "Lending"

$Y$  := Attribute "Name" und "Lecture" aus Tabelle "Student"

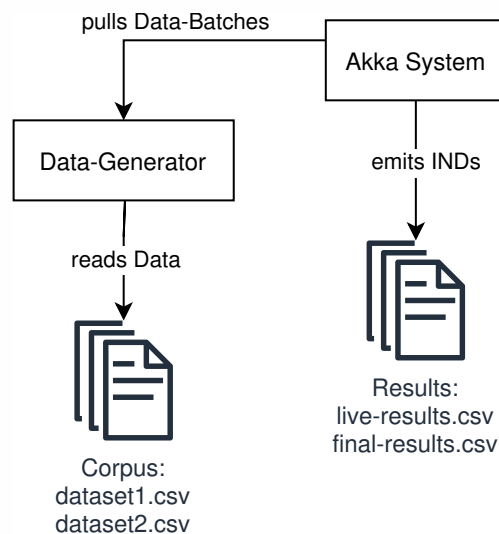
Bei n-ary IND's ist es nicht nur wichtig das alle Werte der einzelnen Attribute aus  $X$  in  $Y$  auftauchen, sondern das sie vor allem in der Kombination in  $Y$  auftauchen, in der sie auch in  $X$  auftauchen.

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.

[1] Papenbrock, Thorsten 2021: DDM Hands-on Akka Actor Programming, Distributed Data Management, WS 21/22 . Foliensatz. Marburg: Philipps-Universität Marburg

## 8 Definition des Zielsystems

### 8.1 System-Kontext



## Darstellung des Systems und seines Kontexts

Das System spaltet sich in den **Data-Generator** und in das **Akka System**.

Die Aufgabe des **Data-Generators** ist, ein synthetisches dynamisches Dataset von beliebiger Länge zu generieren. Dazu liest er Korpus von bestehenden Datasets aus. Die Einträge dieser Datensätze werden wiederholt, umgeordnet, gelöscht und modifiziert als Batches aus Änderungen verpackt.

Weil das Generieren dieser Batches sehr viel günstiger als ihre Analyse sein wird, werden Batches vom Empfänger gepullt statt zum Empfänger gepusht (Pull-Architektur statt Push-Architektur).

Die Aufgabe des **Akka Systems** ist es, Batches aus Änderungen anzunehmen und das synthetische Dataset zu rekonstruieren und zu updaten. Dabei soll es fortwährend auf INDs überprüfen. Es pullt Batches vom Data-Generator so schnell, wie es sie analysieren kann.

	A	B	C	D	E	
1	timestamp	attribute_a	attribute_b	is_valid	reason	info
2		2 tpch_customer[C_NAME]	tpch_nation[N_NATIONKEY]	false	cardinality	9151 < 25
3		2 tpch_lineitem[L_TAX]	tpch_region[R_REGIONKEY]	false	cardinality	9 < 5
4		2 tpch_part[P_PARTKEY]	tpch_lineitem[L_DISCOUNT]	false	cardinality	12264 < 11
5		2 tpch_nation[N_NAME]	tpch_lineitem[L_RECEIPT]	false	extrema	extremaA=[IRAN,UNITED KII
6		2 tpch_region[R_COMMENT]	tpch_nation[N_NATIONKEY]	false	extrema	extremaA=[108\$33144561,I
7		2 tpch_orders[O_ORDER]	tpch_supplier[S_ADDRESS]	false	datatype	timestamp $\neq$ string
8		2 tpch_lineitem[L_SUPPKEY]	tpch_customer[C_NAME]	false	datatype	integer $\neq$ string
9		2 tpch_lineitem[L_COMMENT]	tpch_part[P_MEGR]	false	cardinality	11146 < 5

Auszug einer live-results.csv

Die gefundenen INDs werden im laufenden Betrieb in eine `live-results.csv` Datei ausgegeben. Dabei werden folgende Informationen über IND-Kandidaten festgehalten:

- `timestamp`: Der relative Zeitstempel seit Start des Programms
- `attribute_a`: Der Name des abhängigen Attributs
- `attribute_b`: Der Name des referenzierten Attributs
- `is_valid`: `true` wenn der Kandidat valide ist, `false` wenn der Kandidat invalide ist
- `reason`: Der Grund warum der Kandidat als valide/invalide befunden wurde
  - `cardinality`: Purged anhand der Kardinalität (= `false`)
  - `extrema`: Purged anhand der Extremwerte (= `false`)
  - `datatype`: Purged anhand des Bloomfilters (= `false`)
  - `bloomfilter`: Purged anhand des Bloomfilters (= `false`)
  - `check_failed`: Gescheiterter Subset-Check (= `false`)
  - `check_succeeded`: Erfolgreicher Subset-Check (= `true`)

```

1 tpch_customer.csv → tpch_nation.csv: C NATIONKEY ⊆ N NATIONKEY
2 tpch_customer.csv → tpch_part.csv: C CUSTKEY ⊆ P PARTKEY
3 tpch_customer.csv → tpch_supplier.csv: C NATIONKEY ⊆ S NATIONKEY
4 tpch_lineitem.csv → tpch_customer.csv: L LINENUMBER ⊆ C CUSTKEY
5 tpch_lineitem.csv → tpch_customer.csv: L LINENUMBER ⊆ C NATIONKEY
6 tpch_lineitem.csv → tpch_customer.csv: L SUPPKEY ⊆ C CUSTKEY
7 tpch_lineitem.csv → tpch_lineitem.csv: L COMMIT ⊆ L RECEIPT
8 tpch_lineitem.csv → tpch_lineitem.csv: L COMMIT ⊆ L SHIP
9 tpch_lineitem.csv → tpch_lineitem.csv: L TAX ⊆ L DISCOUNT
10 tpch_lineitem.csv → tpch_nation.csv: L LINENUMBER ⊆ N NATIONKEY
11 tpch_lineitem.csv → tpch_orders.csv: L LINENUMBER ⊆ O ORDERKEY
12 tpch_lineitem.csv → tpch_orders.csv: L LINESTATUS ⊆ O ORDERSTATUS

```

Auszug einer final-results.txt

Sobald der Data-Generator keine Data-Batches mehr liefert, wird der finale Zustand der synthetischen Datensets analysiert und alle am Ende validen INDs werden nochmal in eine final-results.txt Datei ausgegeben.

## 8.2 Datenformat (Batches)

Ein Datenset besteht aus mehreren **Tabellen**, die unterschiedliche Schemata 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	<b>parrot</b>
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.

## 9 Funktionale Anforderungen

### 9.1 Datengenerator

Der Datengenerator soll einen beliebig großen Batch einer beliebigen CSV Dateien generieren. In diesem Batch sollen anschließend mittels des dynamischen Algorithmus Inclusion Dependencies 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.

### 9.2 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 von bis zu mehreren Gigabyte zurecht kommen können.

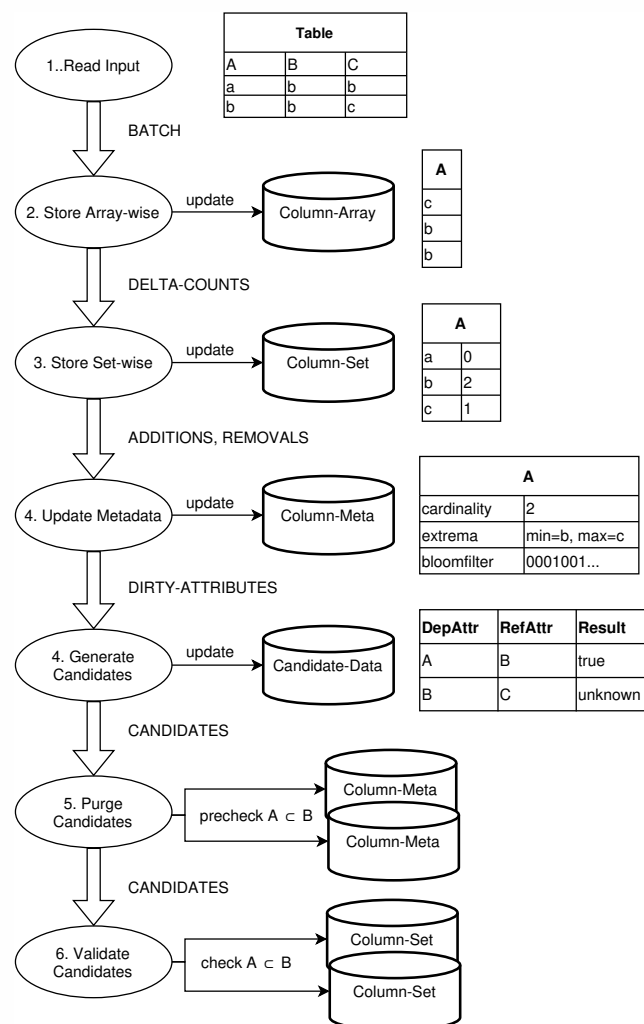
## 10 Algorithmenentwurf

## 10.1 Datenfluss

Bevor wir das Akka System mit Akka Aktoren implementieren, definieren wir den grundlegenden Datenfluss den wir 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 das 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.



Datenfluss für inkrementelle Updates und dazugehörige Speicher

### 1. Read Input

Es wird ein Batch 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 IND's 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  $\geq 1$  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  $\geq 1$ , 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 verlinke).

## 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-Prechecking* 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 [Pruning Pipeline](#).

## 7. Validate Candidates

Nachdem wir in der Vorarbeit die Anzahl an Attributen, die wir auf IND's überprüfen so weit wie möglich reduziert haben, werden nun die übrig gebliebenen Kandidaten mittels *Subset-Checking* validiert. Erst jetzt werden die Werte der Column-Sets 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 ergeben, brechen wir ab.

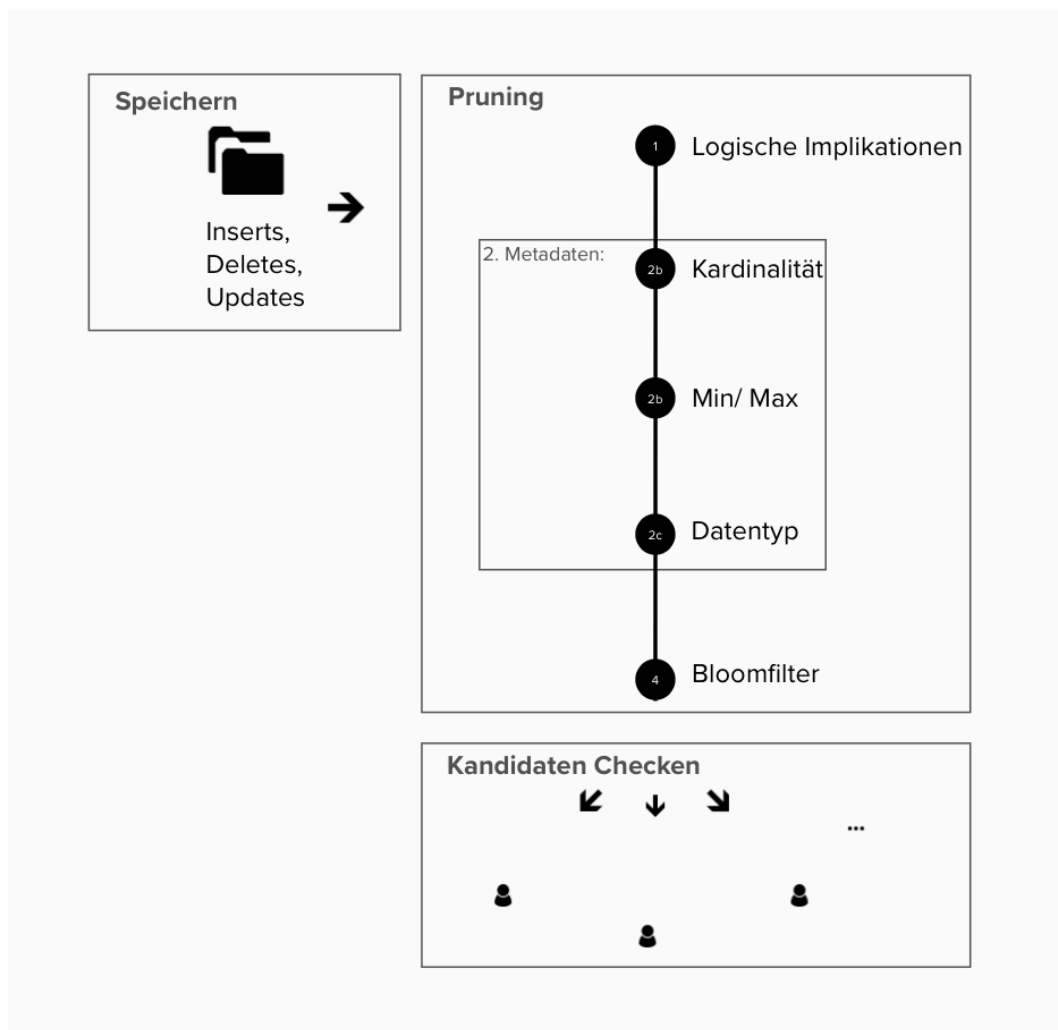
Beispiel:

A hat 100 einzigartige 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.

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

# 10.2 Pruning Pipeline





Pruning Pipeline

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

Im Status Quo suchen wir lediglich nach unären IND's. Als Fortführung könnte man nach n-ären IND's suchen.

## 1. Pruning durch logische Implikation

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

Bei Hinzufügen oder Löschen von Werte Wenn  $A \subset B$ : A erhält ein neues Element und B bleibt gleich  $\Rightarrow A \subset B$ .

Wenn  $B \subset A$ : A erhält ein neues Element und B bleibt gleich  $\Rightarrow B \subset A$ .

## 2. Pruning durch Metadata

Aus den Metadaten der Attribute kann man Kandidaten ausschließen. Durch Single-Column-Analysis erhalten wir verschiedenen Metadaten.

### 2a. Kardinalitäten

Eine Art der Metadaten sind die Kardinalitäten. Über die Anzahl von unterschiedlichen Werten kann man IND's ausschließen.

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

`cardinality(A)=5`

`cardinality(B)=3`

$\Rightarrow A \not\subset B$

Wenn A mehr einzigartige Werte als B hat, dann kann A nicht vollständig in B enthalten sein. Somit muss eine IND von A in B nur überprüft werden, wenn  $cardinality(A) \leq cardinality(B)$ . Nicht aber wenn  $cardinality(A) > cardinality(B)$ .

### 2b. Min/Max

Mittels der Extremwerte eines Attribut kann man ausschließen, ob eine IND besteht. Unter Extremwerte verstehen wir die Min-Werte und Max-Werte nach lexikographischer Ordnung der Werte-Strings.

Ist ein Extremwert des Attributes A in  $A \subset B$  kleiner oder größer als die Extremwerte des Attributs B, so können wir ausschließen dass A vollständig in B enthalten ist.

### 2c. Datentyp

Weiterhin prüfen wir die Datentypen, die in einer Spalte vorkommen.

Mögliche Datentypen:

- Integer: -10, 0, 10, 20000
- Real: 1, 2.0, -1.0e-7
- Timestamp: 2012-12-01 10:00:30
- String: Alle oberen Beispiele und auch sonst alles Zeichenfolgen, inklusive dieses Satzes.

Datentypen können andere Datentypen enthalten:

`Integer ⊂ Real ⊂ String Timestamp ⊂ String`

Sollte vor einem Subset-Check  $A \subseteq B$  A einen Datentyp haben, dessen Werte per Definition nicht in B enthalten sein können, so kann A nicht in B enthalten sein.

$\text{datatype}(A) \not\subseteq \text{datatype}(B) \Rightarrow A \not\subseteq B$

### 3. Bloomfilter

Ein weiterer Ausschluss findet durch Nutzung von Bloomfiltern<sup>[1]</sup> statt. Derzeit genutzt wird ein Counting-Bloomfilter mit einer Größe von 16000 Elementen 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, in unserem Fall zwei, 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.

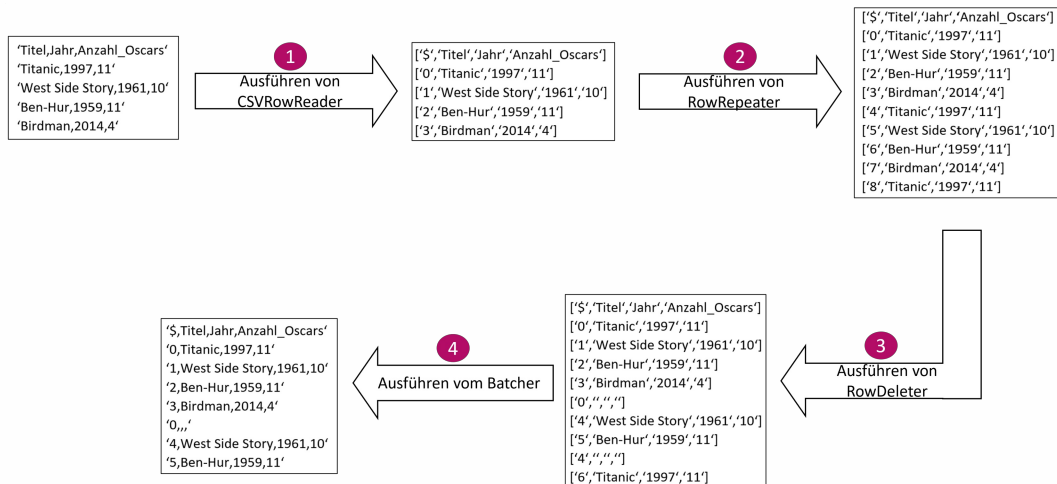
Counting Bloomfilter ergänzen Bloomfilter dahingehend, dass nun mitgezählt wie oft ein Bit im Array auf 1 gesetzt wird. Das ermöglicht auch Elemente zu löschen. Jedes der  $m$  Elemente besitzt einen Counter. Wird ein Element hinzugefügt, so werden die zugehörigen counter hochgezählt, wird ein Element entfernt, so wird der Counter heruntergezählt.

[1] Tarkoma, Sasu, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and practice of bloom filters for distributed systems." IEEE Communications Surveys & Tutorials 14.1 (2011): 131-155.(#a1)

# 11 System-Entwurf

## 11.1 Datengenerator

Der Datengenerator ist eine Komposition aus den vier Klassen `CSVRowReader`, `RowRepeater`, `RowDeleter` und `Batcher`.



Beispielhafte Darstellung einer einmaligen Ausführung des Datengenerators

Dem Datengenerator wird der Pfad einer CSV-Datei sowie eine Anzahl an Konfigurationsparametern übergeben. Der Generator nimmt diese CSV-Datei und generiert die Batches auf die gewünschte Weise.

### 1. CSVRowReader

Dafür wird zunächst die Datei eingelesen, wobei jede Zeile in ein String-Array umgewandelt wird. Zusätzlich wird eine Spalte angefügt, in der jede Zeile fortlaufend durchnummeriert wird.

### 2. RowRepeater

Diese Zeilen-Arrays werden jetzt so lange von vorne nach hinten wiederholt bis die übergebene Anzahl an gewünschten Reihen erreicht ist.

Wenn ein `mutate` Flag gesetzt wird, wird zusätzlich nach jeder Wiederholung ein `${nth_repeat}` Postfix zu jedem Zellenwert angefügt. Also `$1` bei der 1ten Wiederholung, `$2` bei der 2ten, etc..

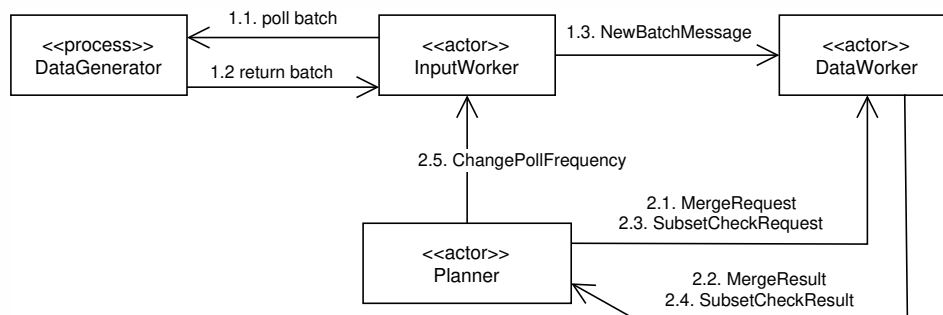
### 3. RowDeleter

Bei der Generierung neuer Zeilen-Arrays wird mit 10% Wahrscheinlichkeit stattdessen eine vorherige Zeile. Dafür wird eine Array mit leerer Liste aber bekanntem Index hinzugefügt.

#### 4. RowDeleter

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.

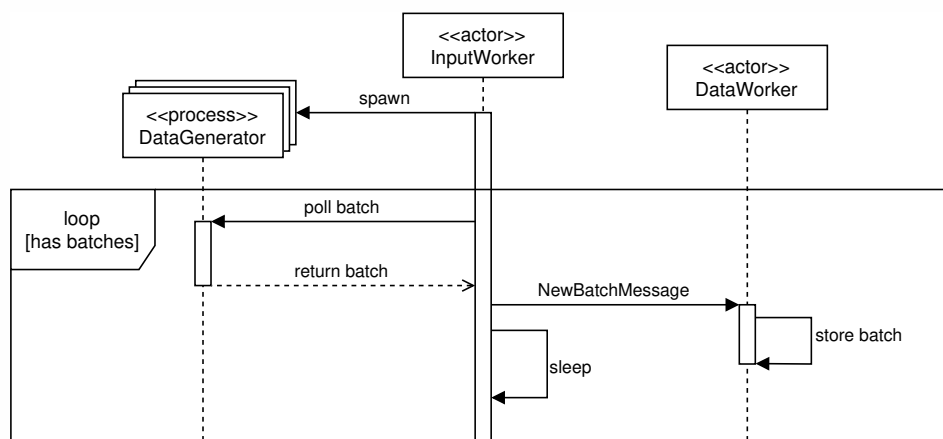
## 11.2 Single-Host Akka System



Kommunikationsdiagramm für das versimplerte Single-Host Akka System

Das versimplerte Single-Host Akka System dient dazu, ein erstes funktionsfähiges MVP (Minimum Viable Product) zu liefern, ohne Rücksicht auf Daten- oder Taskverteilung. Es kann dennoch dazu genutzt werden, die Korrektheit des dynamischen Algorithmus und die Funktionsweise bestimmter Aktoren zu testen.

#### InputWorker



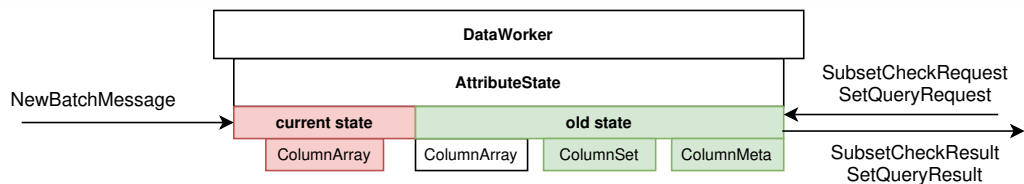
Aktivitäten des InputWorker

Der `InputWorker` spawnst pro Datenset des Korpus (also pro CSV-Datei) einen `DataGenerator`-Prozess und hat als Aufgabe, Batches zu pollen solange diese Prozesse leben. Jedes gepollte Batches wird als `NewBatchMessage` an den `DataWorker` weitergeleitet. Während der

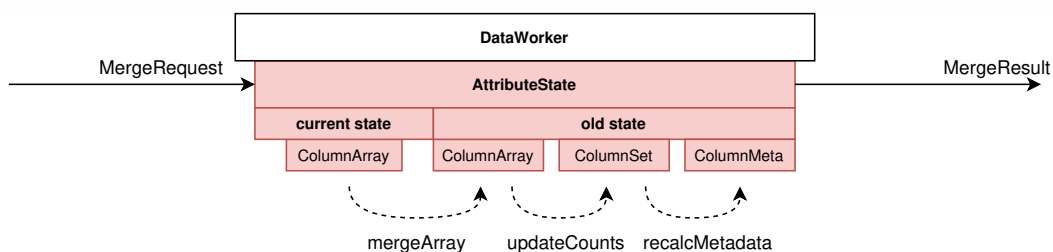
`DataWorker` diese Batch abspeichert, wartet der `InputWorker` für eine konfigurierbare Zeitdauer (gesetzt über `ChangePollFrequency`) bevor er versucht neue Batches zu pollen.

Der `InputWorker` beendet sich, sobald sich alle gespawnten `DataGenerator`-Prozesse beendet haben.

## DataWorker



### Parallelisierbarkeit des DataWorkers



### Merge-Process des DataWorkers

Damit das Einlesen von Batches und das Generieren/Prüfen von Kandidaten parallel stattfinden kann, wird der `DataWorker` auf eine besondere Weise konzipiert.

Für jedes Attribut ist der dazugehörige `AttributState` aufgeteilt in einen `current` und einen `old` Teil. Im `current` Teil werden alle neuen Daten vorgehalten, die als Batches empfangen werden (`NewBatchMessage`). Sie werden auf Anfrage hin in den `old` Teil geschrieben (`MergeRequest`). Die Daten im `old` Teil werden verwendet um Kandidaten zu prüfen (`SubsetCheck`) und um Werte abzufragen (`SetQueryRequest`).

Diese Aufteilung in `current` und `old` ist später im [Multi-Host Akka System](#) notwendig, um Inkonsistenzen zwischen mehreren `DataWorker` Instanzen zu vermeiden. Sie erleichtert außerdem dem `Planner` die Arbeit, sodass er immer mit dem stabilen `old` Teil statt dem volatilen `current` Teil arbeiten kann.

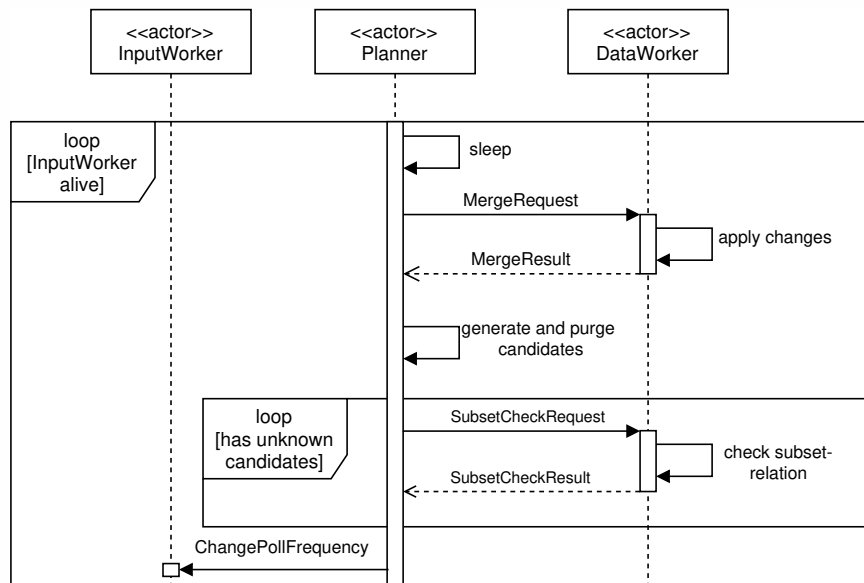
Wird ein Merge mittels eines `MergeRequest` angestoßen, so werden die Daten im `ColumnArray` des `current` Teils in das `ColumnArray` des `old` Teils geschrieben. Indessen wird gezählt berechnet, wie häufig ein Wert hinzugefügt oder entfernt wurde, und anschließend werden diese Delta-Counts in das `ColumnSet` des `old` Teils geschrieben. Aus dem geupdateten `ColumnSet` wird ein neues `ColumnMeta` berechnet, welches alle Metadaten des Attributs enthält, und in einem `MergeResponse` an den Requestor zurückgeschickt.

Siehe [Datenfluss](#) für die verwendeten Datenstrukturen wie `ColumnArray`, `ColumnSet`,

ColumnMeta.

Der DataWorker wird beendet, sobald sich der Planner beendet.

## Planner



Aktivitäten eines Planner

Der Planner hat mehrere Aufgaben:

1. Scheduling von Merges (MergeRequest)
2. Generation und Purging von neuen Kandidaten
3. Scheduling von Kandidats-Prüfungen (SubsetCheckRequest)
4. Regulieren der Batch-Einlese-Frequenz (UpdatePollFrequency)

Zu Beginn eines Durchlaufes schläft der Planner für einen konfigurierbaren Zeitraum (merge frequency), damit sich genug Änderungen im DataWorker ansammeln.

Als Nächstes sendet er einen MergeRequest an den DataWorker und wartet auf alle MergeResponses, welche die geupdateten Attributes und ihre Metadaten beschreiben.

Die geupdateten Attributes und ihre Metadaten werden abgespeichert. Dann werden aus den neuen Attributen alle neuen IND-Kandidaten generiert und aus allen geupdateten Attributen alle Kandidaten re-generiert, die sich potentiell geändert haben könnten.

Alle generierten Kandidaten werden anhand ihrer Metadaten in einer [Purging-Pipeline](#) gefiltert. Die gepurgten Kandidaten werden in der `live-results.csv` vermerkt. Nur die verbliebenen Kandidaten müssen noch geprüft werden.

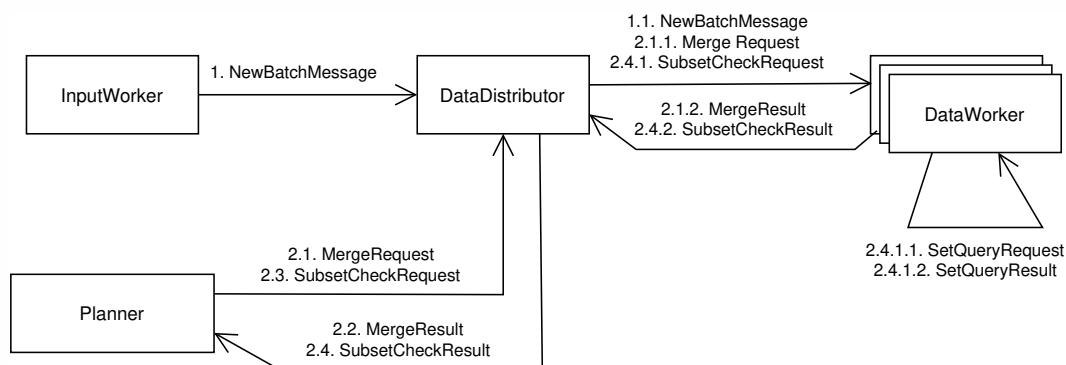
Für jeden zu-prüfenden Kandidaten wird ein SubsetCheckRequest an den DataWorker gesendet und das entsprechende SubsetCheckResult abgewartet. Die Ergebnisse werden gespeichert und in der `live-results.csv` ausgegeben.

Zuletzt wird die dynamische Batch-Einlese-Frequenz berechnet und als `ChangePollFrequency` an den `InputWorker` gesendet.

**Hinweis:** Wir haben bisher keine zufriedenstellende Heuristik für die Berechnung der Batch-Einlese-Frequenz gefunden. Daher wird die Batch-Einlese-Frequenz derzeit als Kommandozeilen-Parameter übergeben und bleibt konstant.

Der `Planner` beendet sich, sobald der `InputWorker` sich beendet. Mit dem Beenden des `Planners` wird das gesamte Akka System beendet.

## 11.3 Multi-Host Akka System



Kommunikationsdiagramm für das verteilte Multi-Hosts Akka System

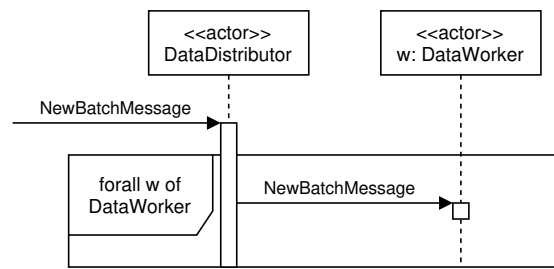
Das komplexere Multi-Host Akka System das finale Produkt unserer Entwicklung. Nachdem die `Planner`, `InputWorker` und `DataWorker` Aktoren im versimpelten Single-Host Akka System entwickelt und geprüft wurden, wird es mittels eines `DataDistributor` Aktor erweitert. Dieser `DataDistributor` hat das selbe Interface wie ein `DataWorker`, delegiert aber auf mehrere `DataWorker` die möglicherweise auf unterschiedlichen Hosts laufen.

### DataDistributor

Zu Beginn wird der `DataDistributor` mit einer fixen Anzahl an `DataWorker`-Referenzen konstruiert. Weitere `DataWorker` können sich im Lauf der Ausführung registrieren.

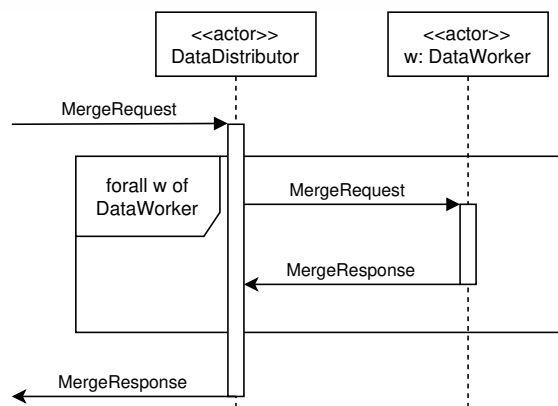
Für den `InputWorker` und den `Planner` muss der `DataDistributor` die identische Funktionalität liefern wie ein `DataWorker`. Intern muss der `DataDistributor` Anfragen auf auf verschiedene Weisen abhandeln und die Arbeit zwischen mehreren `DataWorker` koordinieren.





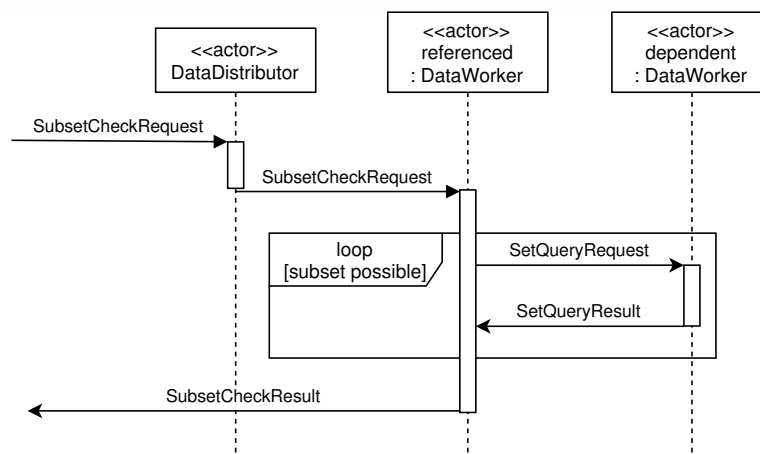
Abhandlung einer NewBatchMessage an den DataDistributor

Im Falle einer NewBatchMessage wird das Batch gemäß Modulo-Partitionierung vertikal (= spaltenweise) aufgeteilt und als kleinere NewBatchMessage an die zuständigen DataWorker weitergeleitet. Es wird gespeichert, welche DataWorker für welche Attribute zuständig sind.



Abhandlung eines MergeRequest an den DataDistributor

Im Falle eines MergeRequest wird ein MergeRequest an alle DataWorker weitergeleitet. Sobald alle mit einem MergeResponse geantwortet haben, wird ein MergeResponse zurückgesendet welches die Informationen aller DataWorker verpackt.



Abhandlung eines SubsetCheckRequest an den DataDistributor

Im Falle eines SubsetCheckRequest wird ein SubsetCheckRequest an den DataWorker gesendet, der für das referenzierte Attribut zuständig ist. Falls ein anderer DataWorker für das abhängige Attribut zuständig ist, so wird eine Referenz zu diesem dem SubsetCheckRequest

angehängen.

Jetzt wird zwischen zwei Fällen unterschieden:

1. Empfängt der `DataWorker` ein `SubsetCheckRequest` ohne Referenz auf einen anderen `DataWorker`, so führt er den Subset-Check direkt auf den beiden `AttributeState`-Instanzen der Attribute aus.
2. Empfängt der `DataWorker` ein `SubsetCheckRequest` mit Referenz auf einen anderen `DataWorker`, so fragt er die Werte des abhängigen Attributs von diesem in einer Schleife ab. Werte werden abgefragt, bis eine Subset-Beziehung zwischen den Wertemengen ausgeschlossen werden kann oder bis alle Werte abgefragt wurden (und eine Subset-Beziehung besteht).

In beiden Fällen wird das Ergebnis als `SubsetCheckResponse` direkt an den Requestor zurückgesendet, dessen Aktor-Referenz im `SubsetCheckRequest` mitgeführt wird.

## 11.4 Weiterentwicklung

Potenzial für zukünftige Arbeiten liegt in der Optimierung des Algorithmus und in der Erweiterung der Aufgabenstellung.

Eine der wichtigsten Eigenschaften des Algorithmus sollte sein, dass er sehr schnell ist. Dies ist sehr wichtig um mit den dynamisch wachsenden Daten mitzukommen. So könnte man noch weitere Möglichkeiten suchen um den Algorithmus zu verschnellern und mehr Kandidaten auszuschließen.

### 11.4.1 Logische Implikationen

Anhand von weiteren logischen Implikationen könnte man weitere Kandidaten ausschließen. Es wäre beispielsweise möglich das eigentliche Candidate Checking in mehrere Schritten auszuführen und jeweils nur einen Teil der Kandidaten zu überprüfen. Aus den Zwischenergebnissen können dann für die weiteren Kandidaten wieder vorher einige ausgeschlossen werden. Durch die logischen Implikationen:

Wenn  $A \subset B$  und  $B \subset C$ , dann  $A \subset C$ .

Wenn  $A \subset B$  und  $B \not\subset C$ , dann  $A \not\subset C$ .

So müssten noch weniger von dem teuersten Kandidaten-Checking durchgeführt werden.

### 11.4.2 Parallelisierung der Dateneinlese und -verteilung

In unserer derzeitigen [Akka Architektur](#) existiert nur eine einzige `InputWorker`-Instanz, der Daten einliest. Das bedeutet, es kann nur ein Host Daten einlesen. Es liessen sich zwar leicht weitere `InputWorker` hinzufügen, aber das macht keinen Sinn, solange man nur eine einzige `DataDistributor`-Instanz hat.

Der `DataDistributor` verteilt `Input-Batches` auf mehrere `DataWorker`. Alle Daten, die derzeit eingelesen werden, müssen an den `DataDistributor` gesendet werden. Das macht den `DataDistributor` zum Haupt-Bottleneck unseres Systems.

Das Erstellen mehrerer `DataDistributor`-Instanzen über mehrere Hosts hinweg wäre komplex. Einseits müssten sich die `DataDistributor`-Instanzen darüber abstimmen, wie die Daten auf mehrere `DataWorker` partitioniert werden sollen. Weiterhin müsste Inkonsistenzen verhindert werden, die durch die parallelen Abläufe entstehen können (siehe [Single-Host Akka System](#) und [Multi-Host Akka System](#)):

- Out-of-order `NewBatchMessages` und `MergeRequests` können dazu führen, dass einzelne `DataWorker` den Merge durchführen, bevor sie die neuen Daten erhalten haben.
- Out-of-order `MergeRequests` und `SubsetChecks` können dazu führen, dass `Subset-Checks` auf einem teilweise veralteten Zustand durchgeführt werden.
- Out-of-order `MergeRequestss` und `SetQueryRequestss` können dazu führen, dass sich während subsequenten `Queries` sich die unterliegenden Daten ändern.

Derzeit sind diese Abläufe konsistent, eben weil nur eine einzige `DataDistributor`-Instanz eine zeitlich-geordnete Reihenfolge der Nachrichten erzwingt.

### 11.4.3 Partitionierung anhand des initialen Batches

Das erste eingelesene `Input-Batch` kann auf Ähnlichkeiten der Spalten geprüft werden. Diese Information könnte dazu genutzt werden, die Attribute effizienter auf `DataWorker` zu partitionieren.

Attribute die sich sehr ähnlich sind müssen vermutlich später mit teuren `Subset-Checks` überprüft werden. Diese `Subset-Checks` sind bereits teuer, wenn ein `DataWorker` die Werte beider Attribute hat; aber noch umso teurer, wenn diese beiden Attribute unterschiedlichen `DataWorker`-Instanzen zugewiesen sind.

### 11.4.4 Horizontale Partitionierung

Derzeit wird nur vertikale Partitionierung (spaltenweise Aufteilung) durchgeführt. Es könnte aber effizienter sein, horizontale Partitionierung (zeilenweise Aufteilung) durchzuführen. Man könnte auch beide Partitionierungen gleichzeitig durchführen.

Wir haben zu Beginn versucht, gleichzeitig vertikal und horizontal zu partitionieren. Es hat den Code aber signifikant komplexer gemacht, weswegen wir am Ende einzig vertikale Partitionierung umgesetzt haben.

Die Wiedereinführung von horizontale Partitionierung liesse sich mit moderatem Aufwand im `DataDistributor` und `DataWorker` bewerkstelligen. Möchte man allerdings mehrere `DataDistributor` Instanzen laufen lassen (siehe [Parallelisierung der Dateneinlese und -verteilung](#)), könnte es großen Aufwand erfordern.

### 11.4.5 Parallelisierung des Pruning

Um das Pruning zu verschnellern könnte man einige der Pruning Aufgaben auch bereits Parallelisieren, da sie nicht unbedingt aufeinander aufbauen. Hierbei ist wichtig, beim Speichern der zu prüfenden Kandidaten sorgfältig zu sein. Allerdings muss man prüfen ob es tatsächlich dadurch schneller wird. Vielleicht ist auch das Pipelineprinzip am schnellsten, weil alle Kandidaten, die in einem Schritt ausgeschlossen werden nicht mehr im Nächsten geprüft werden müssen.

## 11.5 Erweiterung der Aufgabenstellung

Zurzeit werden nur unary INDs geprüft. Man könnte den Algorithmus dahingehend erweitern, dass man auch nach n-ary INDs sucht (siehe [Inclusion Dependencies](#)).

# 12 Benutzerdokumentation

## 12.1 Datengenerator

Bevor das Datengenerator-Script ausgeführt werden kann muss der TPC-H Datensatz entpackt und die Abhängigkeiten installiert werden:

```
cd data && unzip TPCH.zip
pip install argparse
```

Das Script kann wie folgt ausgeführt werden:

```
scripts/datagenerator.py CSV_FILE [OPTIONS]
```

Wobei `CSV_FILE` der Pfad zu einer CSV-Datei sein muss. `OPTIONS` kann eine Kombination aus verschiedenen Parametern sein:

- (keine Parameter): Liest einen Datensatz einmal aus.
- `--repeat`: Wiederholt einen Datensatz unendlich oft.
- `--repeat --max-output 100`: Wiederholt einen Datensatz, bis 100MB an Daten ausgegeben wurden.
- `--delete 0.1`: Liest einen Datensatz einmal aus mit 10% Wahrscheinlichkeit, dass alte Zeilen gelöscht werden.
- `--repeat --mutate`: Wiederholt einen Datensatz unendlich oft mit Mutationen nach jeder Wiederholung.
- `--batch-size 1`: Liest einen Datensatz einmal aus mit einer Batch-Größe von maximal 1KB (sonst: 64KB).

Weitere Optionen sind unter `./scripts/datagenerator.py --help` aufgelistet.

```
49;4;MIDDLE EAST;uickly special accounts cajole carefully
50;0;AFRICA;lar deposits. blithely final packages cajole.
51;1;AMERICA;hs use ironic, even requests. s
52;2;ASIA;ges. thinly even pinto beans ca

$,R_REGIONKEY;R_NAME;R_COMMENT
53;3;EUROPE;ly final courts cajole furiously final excuse
54;4;MIDDLE EAST;uickly special accounts cajole carefully
52;;;
55;0;AFRICA;lar deposits. blithely final packages cajole.
56;1;AMERICA;hs use ironic, even requests. s
57;2;ASIA;ges. thinly even pinto beans ca
9;;;
58;3;EUROPE;ly final courts cajole furiously final excuse
59;4;MIDDLE EAST;uickly special accounts cajole carefully
60;0;AFRICA;lar deposits. blithely final packages cajole.
47;;;
61;1;AMERICA;hs use ironic, even requests. s
62;2;ASIA;ges. thinly even pinto beans ca
63;3;EUROPE;ly final courts cajole furiously final excuse
64;4;MIDDLE EAST;uickly special accounts cajole carefully
65;0;AFRICA;lar deposits. blithely final packages cajole.

$,R_REGIONKEY;R_NAME;R_COMMENT
23;;;
```

*Beispielausgabe des Datengenerators mit `--repeat --batch-size 1`. Einzelne Batches werden mit einer leeren Zeile getrennt.*

## 12.2 Akka-System

Das Akka-System läuft unter Java 18. Maven wird für die Kompilation benötigt:

```
mvn package -f pom.xml
```

Das Akka-System lässt sie wie folgt ausführen:

```
java -Xmx8g -ea -cp target/ddm-akka-1.0.jar de.ddm.Main master [OPTIONS]
```

OPTIONS kann eine Kombination aus verschiedenen Parametern sein:

- (keine Parameter): In seiner Default-Konfiguration liest das Akka-System den TPC-H Datensatz einmal aus und berechnet INDs mit 8 lokalen Workern.
- `-ip data/example`: Verwendet den `data/example` Datensatz statt den TPC-H Datensatz.
- `-dg '--repeat --max-output 100'`: Führt das Datengenerator-Skript mit den Parametern `--repeat --max-output 100` aus (wiederholt Datensatz ausgeben bis 100MB an Output erreicht).
- `-w 16`: Berechnet die INDs mit 16 lokalen Workern.

Weitere Optionen sind unter `java -Xmx8g -ea -cp target/ddm-akka-1.0.jar de.ddm.Main --help` aufgelistet.

Eine Ausführung mit dem `data/example` Datensatz, mit 100MB Output pro CSV-Datei und 16 lokalen Workern sähe also so aus:

```
java -Xmx8g -ea -cp target/ddm-akka-1.0.jar de.ddm.Main master -ip
data/example -dg '--repeat --max-output 100' -w 16
```

Das Akka-System beendet sich automatisch und gibt seine Ergebnisse in `live-results.csv` und `final-results.txt` aus.

## 13 Entwicklerdokumentation

### 13.1 Projektstruktur

Die `main` Branch enthält unsere CodeBase.

Die `doku` Branch enthält die schriftliche Ausarbeitung mit generierter PDF Datei.

- `scripts/`: Hier befindet sich derzeit nur das `datagenerator.py` Skript.
- `src/main/java/de/ddm/actors/profiling/`: Hier befinden sich alle unsere Aktoren für unser Aktoren-Protokoll.
- `src/main/java/de/ddm/structures`: Hier befinden sich alle unsere Klassen, die nicht direkt von Akka abhängig sind.
- `src/main/java/de/ddm/configuration`: Hier können die Kommandozeilen-Befehle des Akka-Systems angepasst werden.
- `ddm-akka/`: Hier befindet sich eine alte Version unserer Codebase, die für das Testen verwendet wird.
- `ddm-spark/`: Hier befindet sich eine Implementation des SINDY-Algorithmus in Spark.

### 13.2 Testen

Wir testen die Ergebnisse mittels dem `run_tests.sh` Skript.

Dabei testen wir unserer neuen Codebase gegen die Ergebnisse einer alten Version unserer Codebase. Diese alte Version befindet sich in der Unter-Repository `ddm-akka`. Unsere Tests prüfen derzeit nur, ob alle INDs in einem statischen Datensatz korrekt gefunden wurde. Für dynamische Datensätze haben wir derzeit keine Tests.

Bei unserem Test werden die beiden Output-Dateies `final-results.txt` und `ddm-akka/results.txt` verglichen. Beiden Dateien werden mittels dem UNIX `diff` Tool auf Gleichheit geprüft: `sort ddm-akka/results.txt | diff final-results.csv -`. Bei

gescheiterten Tests werden die nicht-gefunden und falsch-gefunden Einträge hervorgehoben.

```
DynamicDataProfile git:(main) x ./run_tests.sh

+++++ Testing Akka-System on example dataset +++++

SLF4J: A number (1) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
SLF4J: A number (4) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay

+++++ SUCCESSFUL TEST +++++

+++++ Testing Akka-System on TPC-H dataset +++++
SLF4J: A number (1) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay
SLF4J: A number (4) of logging calls during the initialization phase have been intercepted and are
SLF4J: now being replayed. These are subject to the filtering rules of the underlying logging system.
SLF4J: See also http://www.slf4j.org/codes.html#replay

+++++ SUCCESSFUL TEST +++++
```

*Ein erfolgreicher Durchlauf des run\_tests.sh Skripts*