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 (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 zeitaufwendig.

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 IND's 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 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 haben die Arbeit mit einer vorhandenen Akka Architektur von Prof. Papenbrock begonnen, die uns vorher als Hausaufgabenprojekt für das Modul Distributed Data Management diente.

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):

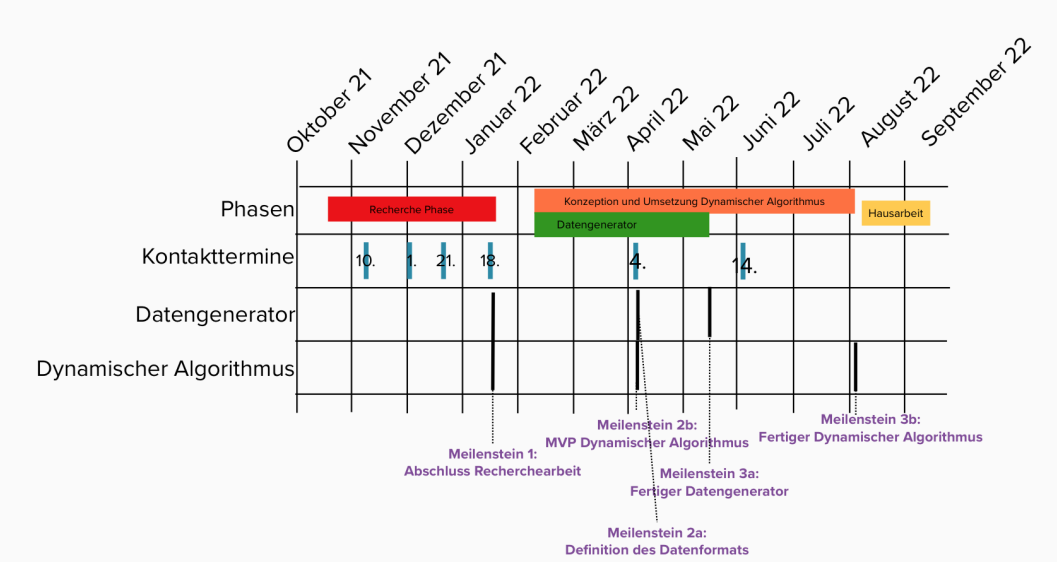
- Konzeption und Entwicklung des Datengenerators (Pairprogramming mit Felix Köpge)
- Erstellung der Auslastungsdiagramme

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

6.0.0.1 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***

6.0.0.2 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

6.0.0.3 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***

6.0.0.4 Zweite Programmiereinheit: 21.05.2022

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

6.0.0.5 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 stehts 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 $X \subseteq Y \wedge Y \subseteq Z \Longrightarrow X \subseteq Z$
- *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 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 ⊆ 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

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

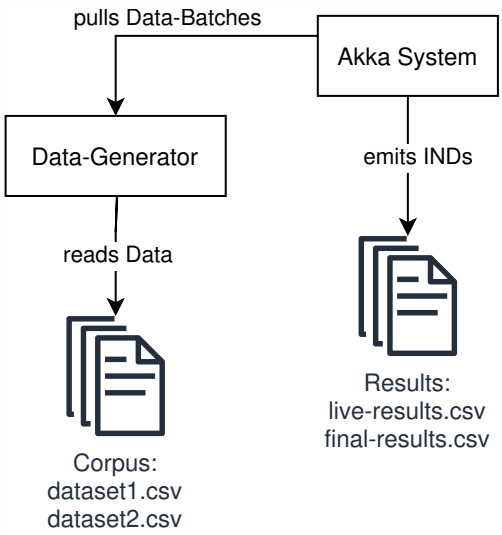
n-ary IND Beispiell[1]
X := Attribute “Student” und “Course” aus Tabelle “Lending”
Y := Attribute “Name” und “Lecture” aus Tabelle “Student”

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 synthethische 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
2	1	tpch_customer[C_NAME]	tpch_nation[N_NATIONKEY]	false	cardinality
3	1	tpch_nation[N_NAME]	tpch_lineitem[L_RECEIPT]	false	extrema
4	1	tpch_lineitem[L_SHIPMODE]	tpch_lineitem[L_RETURNFLAG]	false	cardinality
5	1	tpch_lineitem[L_EXTENDEDPRICE]	tpch_lineitem[L_COMMIT]	false	cardinality
6	1	tpch_customer[C_COMMENT]	tpch_supplier[S_COMMENT]	false	extrema
7	1	tpch_orders[O_TOTALPRICE]	tpch_customer[C_ADDRESS]	false	cardinality
8	1	tpch_region[R_COMMENT]	tpch_nation[N_NATIONKEY]	false	extrema
9	1	tpch_orders[O_COMMENT]	tpch_lineitem[L_SHIP]	false	cardinality
10	1	tpch_supplier[S_SUPPKEY]	tpch_lineitem[L_RETURNFLAG]	false	cardinality
11	1	tpch_orders[O_TOTALPRICE]	tpch_supplier[S_SUPPKEY]	false	cardinality
12	1	tpch_orders[O_ORDERKEY]	tpch_supplier[S_ACCTBAL]	false	cardinality
13	1	tpch_supplier[S_COMMENT]	tpch_orders[O_TOTALPRICE]	false	extrema
14	1	tpch_orders[O_ORDERKEY]	tpch_customer[C_CUSTKEY]	false	cardinality
15	1	tpch_orders[O_ORDER]	tpch_supplier[S_ADDRESS]	false	cardinality
16	1	tpch_lineitem[L_PARTKEY]	tpch_orders[O_CLERK]	false	extrema
17	1	tpch_lineitem[L_ORDERKEY]	tpch_lineitem[L_RETURNFLAG]	false	cardinality

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 abhängige 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`)

final-results-2022-09-30-05-43-41.txt

```
tpch_nation[N_NATIONKEY] c tpch_supplier[S_NATIONKEY]
tpch_lineitem[L_LINENUMBER] c tpch_nation[N_NATIONKEY]
tpch_region[R_REGIONKEY] c tpch_supplier[S_PHONE]
tpch_region[R_REGIONKEY] c tpch_supplier[S_NATIONKEY]
tpch_orders[O_SHIPPRIORITY] c tpch_customer[C_NATIONKEY]
tpch_orders[O_SHIPPRIORITY] c tpch_supplier[S_NATIONKEY]
tpch_nation[N_REGIONKEY] c tpch_customer[C_PHONE]
tpch_lineitem[L_TAX] c tpch_lineitem[L_DISCOUNT]
tpch_customer[C_CUSTKEY] c tpch_supplier[S_SUPPKEY]
tpch_region[R_REGIONKEY] c tpch_nation[N_REGIONKEY]
tpch_lineitem[L_LINENUMBER] c tpch_customer[C_NATIONKEY]
tpch_nation[N_REGIONKEY] c tpch_supplier[S_NATIONKEY]
tpch_nation[N_REGIONKEY] c tpch_supplier[S_PHONE]
tpch_orders[O_SHIPPRIORITY] c tpch_nation[N_REGIONKEY]
```

Auszug einer final-results.txt

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

8.2 Datenformat (Batches)

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.

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 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.

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 zurecht kommen können.

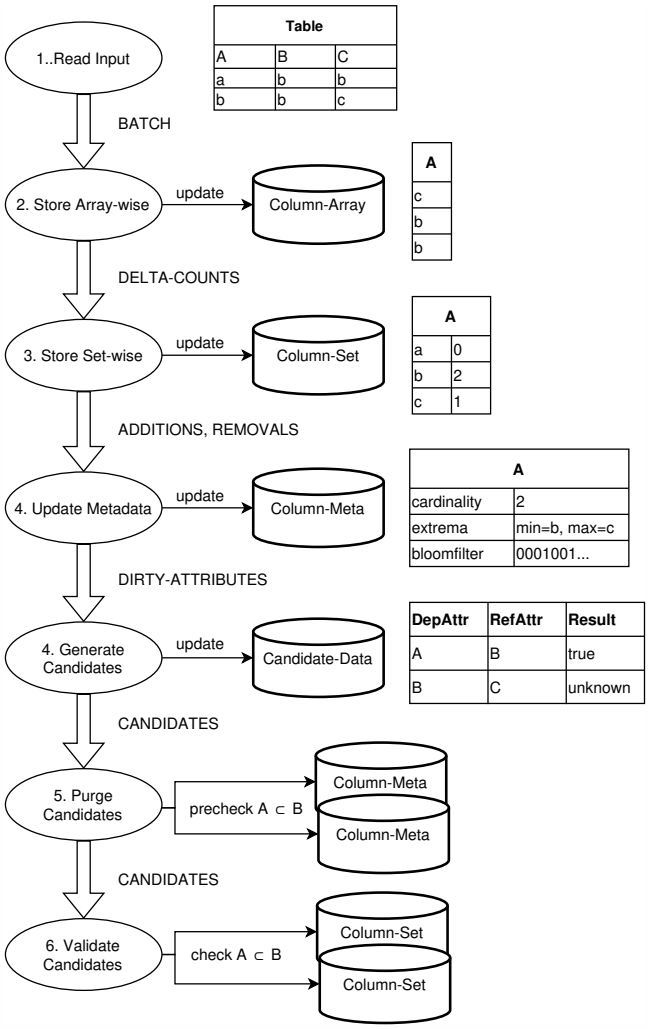
10 Algorithmenentwurf

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 ≥ 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 ≥ 1, 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 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

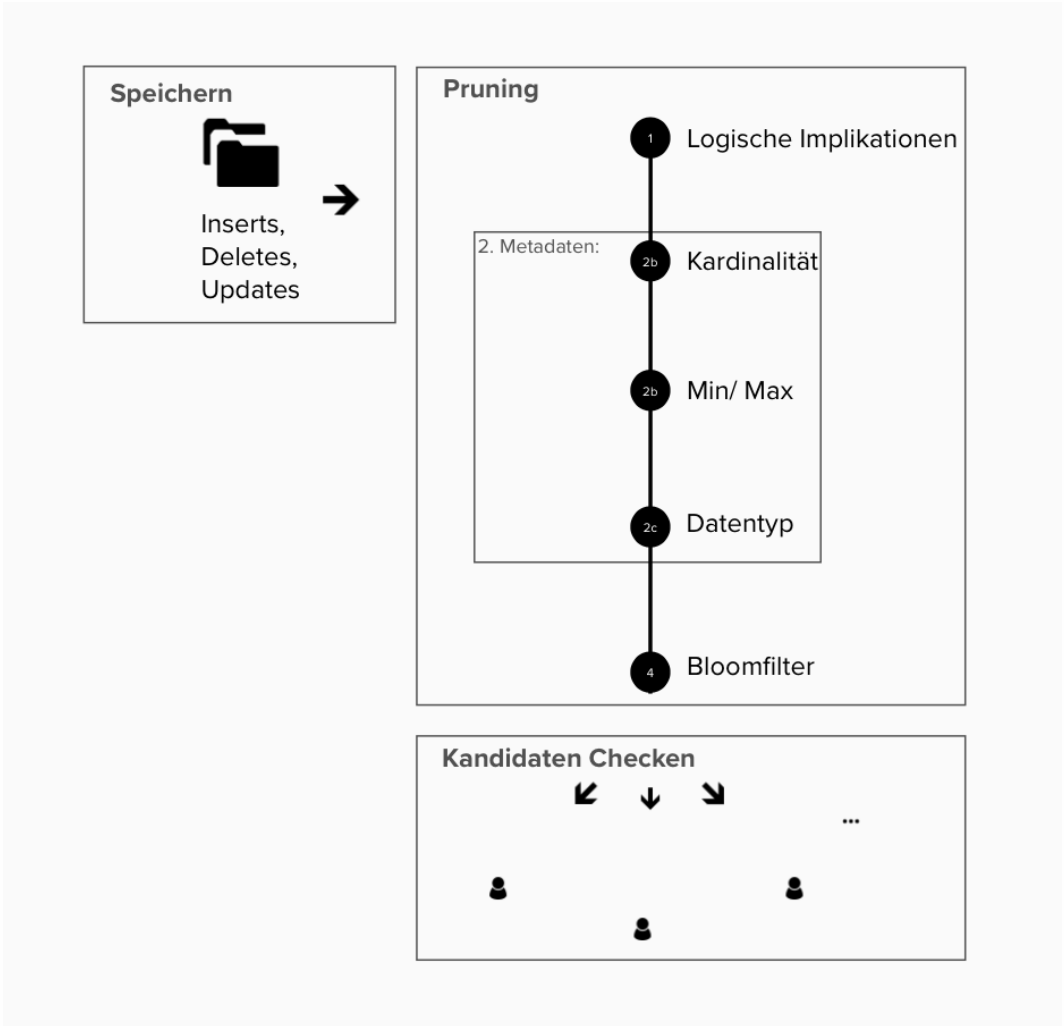
Attributen untersucht wurde, und die Anzahl verbliebener Werte nicht mehr ausreicht um noch eine Inclusion Dependency zu ergeben, brechen wir ab.

Beispiel:

> A hat 100 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.

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

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 $\not\subset$ B: A erhält ein neues Element und B bleibt gleich => A $\not\subset$ B.
Wenn B $\subset$ A: A erhält ein neues Element und B bleibt gleich => B $\subset$ A.

Pruning durch Metadata

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

Tabelle 1

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

Tabelle 2

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

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
=> A ⊄ 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 $cardinality(A) > cardinality(B)$.

Extremwerte

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.

Datentyp

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

Metadata	Charakterisierung	Beispiel
`datatype`	Gemeinsamer Datentyp für alle Werte einer Spalte	datatype(A) = UnsignedInteger datatype(B) = String

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`: Alle oberen und auch sonst alles, inklusive diesen Satzes.

Datentypen können andere Datentypen enthalten:

`UnsignedInteger` $\subset$ `Integer` $\subset$ `Real` $\subset$ `String` `Timestamp` $\subset$ `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.

`datatype`(A) $\not\subset$ `datatype`(B) $\Rightarrow$ A $\not\subset$ B

Bloom Filter

Ein weiterer Ausschluss findet durch Nutzung von Bloomfiltern[1] 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, 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.

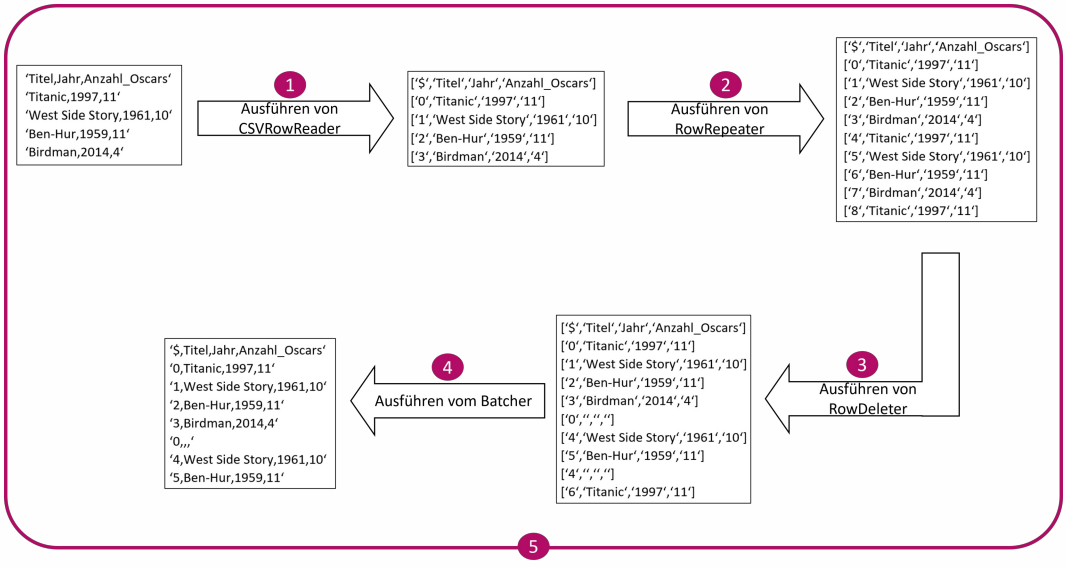
Counting Bloomfilter ergänzen Bloomfilter dahingehend, dass nun mitgezählt wie oft ein Bit im Array auf `1` gesetzt wird. Das ermöglicht auch Elemente zu löschen. Jedes der `m` Elemente besitzt einen Counter. Wird ein Element hinzugefügt, so werden die zugehörigen counter hochgezählt, wird ein Element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

> TODO: Können wir (5) entfernen?

5. Datengenerator

Dem Datengenerator wird der Pfad 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.

> TODO: Das ist noch nicht alles - es gibt mehr Parameter!

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.

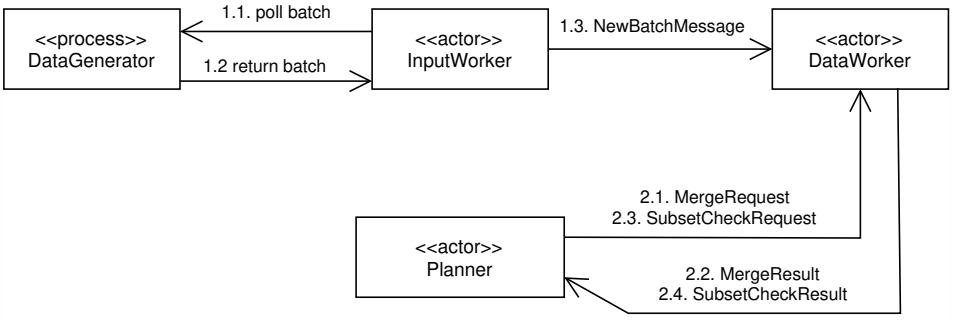
3. RowDeleter

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

4. RowDeleter

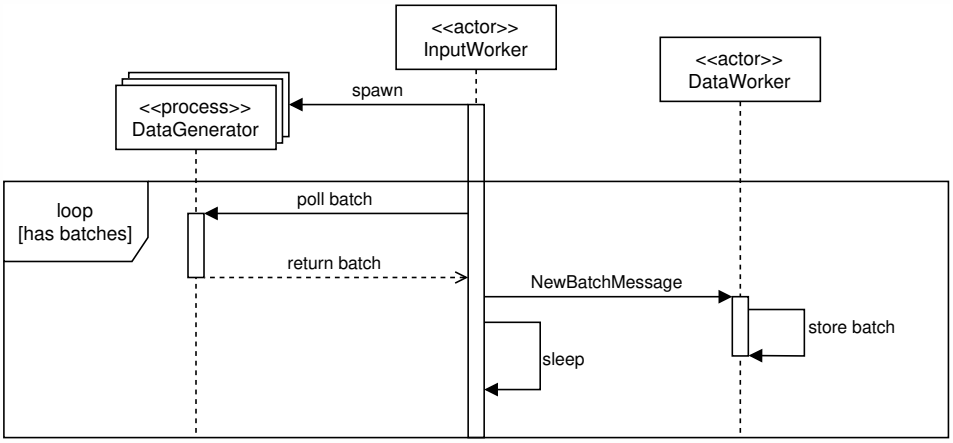
Wenn die Anzahl an gewünschten Reihen erreicht wurde, wird daraus der Batch generiert. Dafür wird jedes Array wieder in eine String umgewandelt und die einzelnen Attribut-Werte durch Kommas getrennt. Es wird also wieder eine CSV-Datei generiert.

11.2 Single-Host Akka System

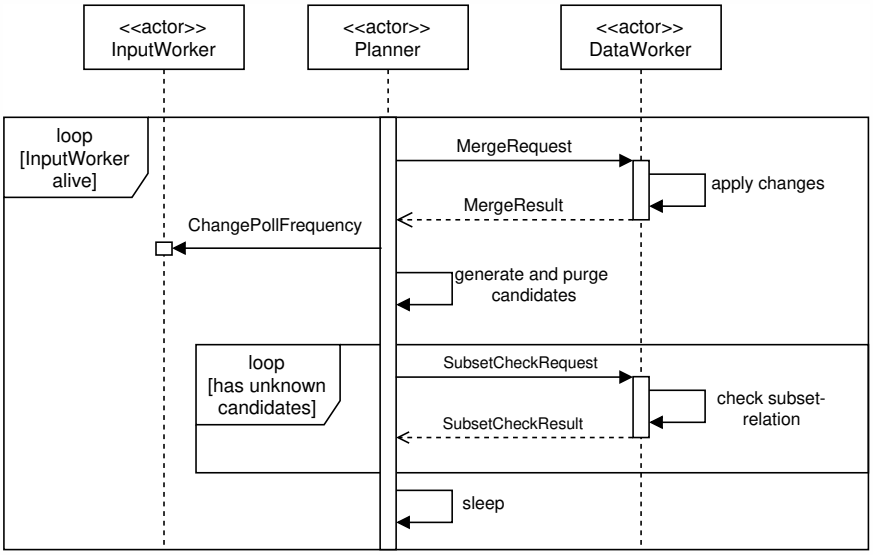


Kommunikationsdiagramm für das versimpelte Single-Host Akka System

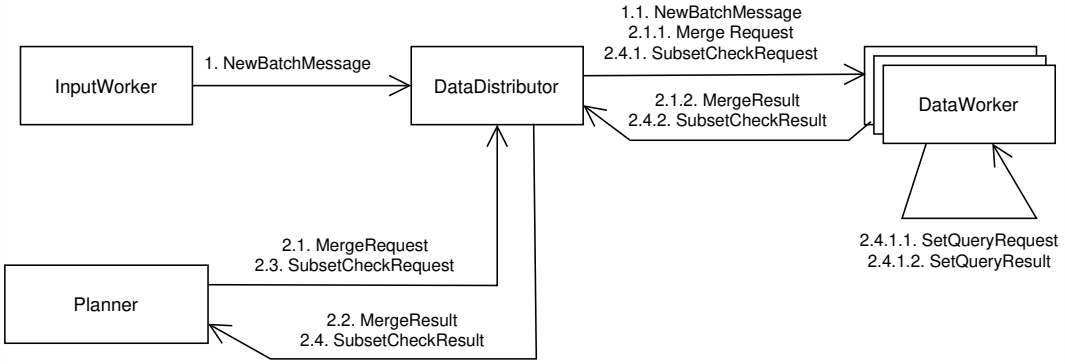
Das versimpelte Single-Host Akka System dient dazu, ein erstes funktionsfähiges MVP (Minimum Viable Product) zu liefern, ohne Rücksicht auf Daten- oder Taskverteilung. Es kann dazu genutzt werden, die Korrektheit des dynamischen Algorithmus zu prüfen und die Funktionsweise bestimmter Aktoren zu testen.



Der `InputWorker` spawnt 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` dieses Batch speichert, wartet der `InputWorker` für eine konfigurierbare Zeitdauer (`AdjustPollFrequency`) bevor er versucht neue Batches zu pollen.



11.3 Multi-Host Akka System



Kommunikationsdiagramm für das verteilte Multi-Hosts Akka System

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

11.4 Optimierungen

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 $B \subset C$, => $A \subset C$.

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

11.4.2 Parallelisierung

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 Unäre IND's geprüft. Man könnte den Algorithmus dahingehend erweitern, dass man auch N-Äre IND's findet. Also wenn mehrere Spalten und Zeilen in mehreren anderen Spalten und Zeilen enthalten sind.

12 Benutzerdokumentation

(später, Readme.md(

13 Entwicklerdokumentation

verlinken! github repo

14 Projektdokumentation

Generierung des PDF Dokuments:

```
pandoc *.md  -t html --pdf-engine-opt=--enable-local-file-access -o full.pdf  -
```