The Document Title

Example Author Another Author

2017-02-20

Contents

1 Beschreibung

1.1 Projekt

1.2 Auftraggeber

1.3 Qualitätsanforderungen

- (siehe DDM folien)

2 Zusammenfassung

3 Erfahrungsbericht

4 Entwicklungsprozess

5 Team

6 Risikomanagement

7 Zeitplan

8 Definition des Zielsystems

8.1 Hardware

8.2 Software

9 Funktionale Anforderungen

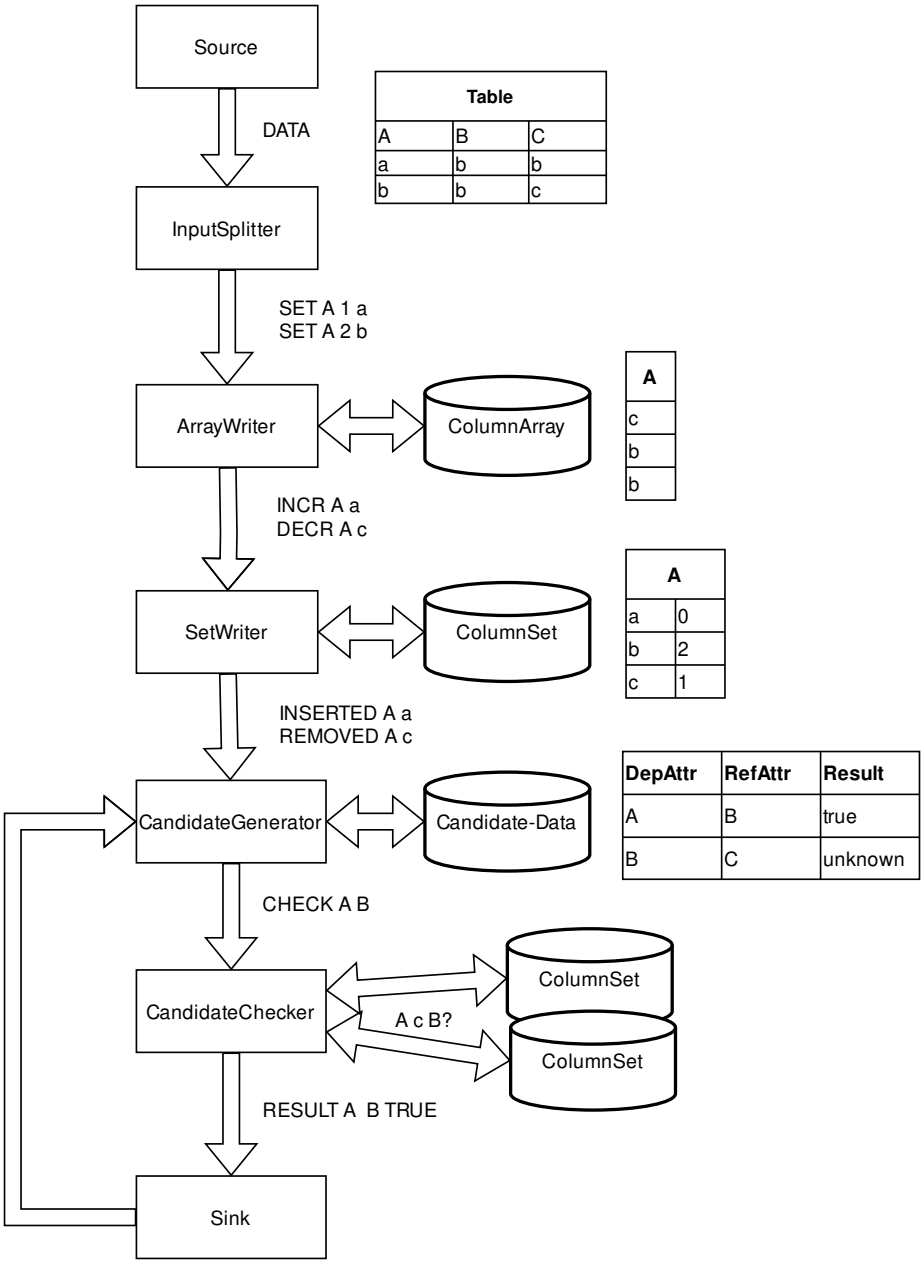
9.1 Inkrementelle Entdeckung von Inclusion-Dependencies

10 Nicht-funktionale Anforderungen

- für Benchmarks: Observability oder Modularer Aufbau (inversion of control)

11 System-Entwurf

11.1 Überblick



11.2 Value Representation

11.2.1 Hashing Long Values

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

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

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

11.2.2 Faster Hash Algorithm

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

Neben Algorithmen der SHA-Familie könnten wir auch xxHash oder MurmurHash verwenden.

11.2.3 Byte Array Values

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

11.3 Smart Candidate Generation

11.3.1 Elimination-by-Implication

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

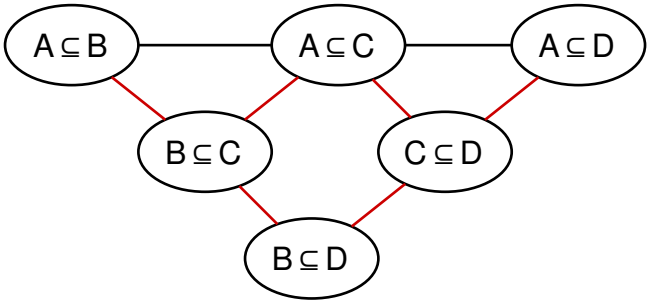
```
A c B  /\    B c D  ->   A c D
A c B  /\  !(A c D) -> !(B c D)
```

11.3.2 Candidate Picking

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

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

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



11.3.3 Candidate Flagging

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

- Counterexamples

11.4 Single-Column-Analysis Prechecking

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

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

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

11.5 Optimierte Subset-Checks

11.5.1 Dirty-Ranges

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

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

11.5.2 Early-Return

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

11.5.3 Bidirectional Check

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

12 Technologien

- **Java**: für Hauptsystem
- **Apache Spark**:
- **Apache Parquet**: Spaltenorientiertes Übertragungsformat
- **Cassandra**: Verteiltes, spaltenorientiertes Datenbanksystem
- **DataStax Spark Cassandra Connector**: Spark-Datasets für Cassandra Datenbanken
- **ScalarDB**: Transaction-Manager (MVCC) für Cassandra
- **Python**: für Datengenerator
- **Faker**: Python-Bibliothek für Generaton von Testdaten
- **collectd**: Linux-Daemon für system performance monitoring, erlaubt verteiltes Benchmarking von CPU/RAM-Nutzung, Netzwerklast, etc

13 Komponenten-Entwurf

14 Beweisidee

15 Algorithmenentwurf

16 Datenbankentwurf

17 Testplan

18 Testprotokoll

19 Beispielanwendungen

20 Benutzerdokumentation

21 Entwicklerdokumentation