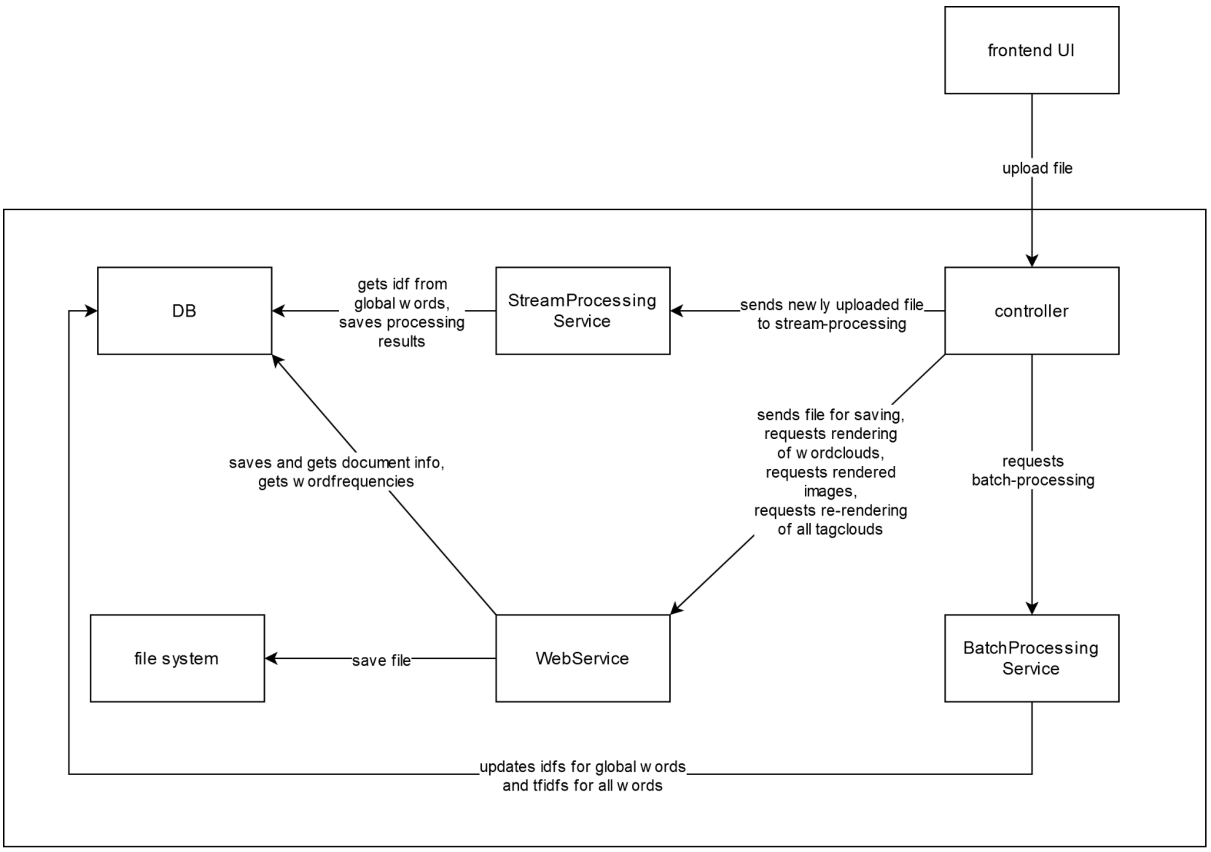
Architektur und Probleme

Gruppe C, Jannis Seefeld, 2160016

GitHub-Repository: https://github.com/HexadimensionalerAlp/BDEA_CA1

Architekturskizze:



Diese Skizze stellt vereinfacht die Komponenten und deren Kommunikation dar.

Entitäten:

```
create table global_words
(
    word  varchar(255) not null
        primary key,
    idf   double       not null,
    tf    bigint       not null,
    tfidf double       not null
);
```

```
create table words
(
    document_id bigint       not null,
    word        varchar(255) not null,
    tf          bigint       not null,
    tfidf       double       not null,
    primary key (document_id, word)
);

create table documents
(
    id   bigint       not null
        primary key,
    name varchar(255) not null
);
```

In *global_words* steht jedes Wort einmal, mit dem Vorkommen in allen Dokumenten. In der Tabelle *words* stehen alle Wörter in Zuordnung zu ihren Dokumenten, die Eindeutigkeit eines Eintrags ergibt sich hier deshalb nicht nur aus dem Wort, sondern aus der Kombination der Id des Dokuments und dem Wort. In *documents* wird den Textdateien eine eindeutige Id zugewiesen.

Die hier genutzte Architektur folgt der Lambda-Architektur. Textdateien können hochgeladen werden, diese werden im Dateisystem hinterlegt, eingelesen und die Worte jeder Datei werden mit ihrer Häufigkeit und ihrer tfidf gespeichert (Speed Layer). Ist in der Tabelle der globalen Wörter noch kein Eintrag für das Wort vorhanden, wird bei der Berechnung der tfidf eine df von 1 angenommen. Anschließend wird eine Tag Cloud für die Textdatei erstellt. Die Einträge in der Tabelle der globalen Wörter werden durch das manuelle Anstoßen des Batch Jobs ausgelöst. Dieser berechnet die globale Häufigkeit jedes Wortes und die idf und tfidf. Außerdem aktualisiert er die tfidf der Einträge in *words*, sowie deren Tag Clouds und generiert die globale Tag Cloud.

Eine fertig einsetzbare Datenbank kann über Docker mit dem Befehl

> docker run --name mariadb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=bdea -p 3306:3306 -d mariadb:latest

heruntergeladen und gestartet werden. Es sind keine weiteren Anpassungen notwendig. Anschließend kann die Anwendung einfach gebaut und gestartet werden. Sie ist dann unter http://localhost:8080/ erreichbar. Im Repository finden sich unter ./testfiles einige Beispieldateien. Sie stammen aus dem Project Guttenberg (https://www.gutenberg.org/) und dem Canterbury Corpus (https://corpus.canterbury.ac.nz/descriptions/).

Probleme:

Bei der Umsetzung sind viele Probleme aufgetreten.
Erstes Mal mit Maven, Spring Boot und Spark, dementsprechend fehlende Kenntnisse bei JavaRDD, RDD, Datasets und der Konvertierung von Rows zu Entitäten.
Größere Kopfschmerzen hat auch ein Fehler im MariaDB-Treiber bereitet: Das Speichern in der DB war mit Spark nicht direkt möglich, die URL musste von `"jdbc:mariadb://localhost:3306"` auf `"jdbc:mysql://localhost:3306"` geändert werden. Hierdurch sind mehrer Stunden verloren gegangen.
Generell war die Verbindung von Spark direkt auf die DB oft etwas schwieriger. Ein weiteres Problem im Bereich DB war die unterschiedliche Handhabung von Spark und JPA beim Mapping von Entitätennamen. JPA hat es zugelassen, dass die Id von Dokumenten in der Datenbank als document_id gespeichert wird, in der Entitätenklasse jedoch als documentId verfügbar ist. Spark hatte damit Probleme und konnte damit nicht umgehen. Folglich wurde das Attribut documentId in document_id umbenannt.
Kumo wirft beim ersten Erstellen einer Tag Cloud einen Fehler (Width (0) and height (0) cannot be <= 0), obwohl beide Parameter gesetzt wurden. Dieser Fehler ist schon länger bekannt und wurde wohl auch mal behoben, ist jedoch wieder aufgetaucht. Alle weiteren Versuche nach dem ersten Versuch eine Word Cloud zu erstellen funktionieren aber problemlos.
Auch für Verwirrung gesorgt hat die Tatsache, dass man bei Entitäten mit mehreren Feldern als Primärschlüssel eine zusätzliche Id-Klasse erstellen muss.
Was mich am Tag vor der Abgabe noch bis spät in die Nacht wachgehalten hat, ist ein Problem mit Spark Dataset.write() mit Mode “Overwrite”. In diesem Modus wurde zwar der Inhalt der Tabelle gelöscht, jedoch kein neuer Inhalt reingeschrieben, obwohl der Inhalt vorhanden war. Da ich nach über zwei Stunden keine Lösung dafür gefunden habe und die Zeit langsam knapp wurde, habe ich dann das Schreiben in die Datenbank an dieser Stelle mit JPA umgesetzt. Das ist jedoch bedeutend langsamer und führt zu längeren Zeiten beim Batch Job. Dieses Problem ist auch nur an dieser Stelle aufgetreten
```
transformedWords.write().mode("Overwrite").jdbc("jdbc:mysql://localhost:3306", "bdea.words", DBUtils.getConnectionProperties());
```
(BatchProcessingService, Zeile 83), an dieser Stelle
```
transformedWords.write().mode("Overwrite").jdbc("jdbc:mysql://localhost:3306", "bdea.global_words",
DBUtils.getConnectionProperties());
```
(BatchProcessingService, Zeile 60) funktioniert es einwandfrei.

Ausblick:

Aktuell verwendet die Anwendung nicht das HDFS, da es ohne dieses einfacher zu Entwickeln und zu Testen ist. Um das HDFS zu verwenden muss man nur Pfad in der Zeile 35 der Klasse *StreamProcessingService* zu einem relationalen Pfad ändern und die Anwendung zum Beispiel in einem Container laufen lassen, in dem das HDFS installiert ist.
Eine weiter Sache, die man mit mehr Zeit verbessern könnte, ist das Warten auf den Response im Frontend. Stattdessen könnte man direkt einen Response bekommen, mit der Nachricht, dass die Datei hochgeladen oder der Batch Job gestartet wurde. Sobald der Upload oder der Batch Job fertig sind, würde das Frontend dann vom Backend benachrichtigt werden.