DHBW Ravensburg

Projektbericht - Sortieralgorithmen

Fortgeschrittene Programmierung

Felix Keller
20.5.2024

1. Einleitung

i. Vorwort

Das gesamte Projekt wurde auf einem Github Repository dokumentiert und kann unter folgendem Link eingesehen werden: felixkxlr/Projektbericht (github.com).

ii. Einleitung

Meine Aufgabe ist es, einen Datensatz zu erstellen und basierend auf diesem einen Sortieralgorithmus zu implementieren. Meinen Datensatz habe ich aus Informationen über verschiedene Autos bezüglich der Frage: „Welches Auto ist der beste Daily Driver?“ erstellt.

Um den Datensatz zu erstellen, habe ich mir einige Autos von verschiedenen Herstellern angesehen und unterschiedliche Daten der Autos herausgeschrieben. Bei der Frage nach dem besten Daily Driver spielen für mich verschiedenste Faktoren eine Rolle. Das Auto sollte auf jeden Fall ein 5-Türer sein mit ausreichend Platz im Kofferraum, damit man auch keine Probleme bekommt, wenn man zum Beispiel in den Urlaub fahren will oder größere Dinge transportieren möchte. Allerdings sollte das Auto auch nicht zu groß sein, damit man komfortabel in Parkhäusern einen Parkplatz findet oder durch enge Gassen, wie zum Beispiel in einer kleinen italienischen Stadt im Urlaub, fahren kann. Außerdem sollte das Auto Spaß am Fahren bereiten, das heißt es sollte eine gute Straßenlage haben, eine angemessene Höchstgeschwindigkeit und auch dementsprechend auch nicht zu wenig PS. Da das Auto als Daily Driver täglich im Einsatz ist, sollte der Durchschnittsverbrauch auch nicht zu hoch sein.

Meine Problemstellung für die Implementierung eines Sortieralgorithmus ist es, den Datensatz absteigend nach PS und gruppiert nach Marke zu sortieren. Genauer bedeutet das, die Autos mit den meisten PS pro Marke zu finden und diese anschließend absteigend nach PS zu sortieren.

Für diese Aufgabe soll einer der im folgenden Kapitel beschriebenen Sortieralgorithmen ausgewählt werden, welcher den nachfolgenden Anforderungen gerecht wird. Es sollen Datensätze mit 1 Milliarden Zeilen aggregiert werden können und der Algorithmus soll in der Lage sein, die Sortierung parallel zu verarbeiten.

2. Sortieralgorithmen

i. Selection Sort

Bei dem Selection Sort haben wir zu Beginn eine unsortierte Liste. Aus dieser wird das kleinste Element ausgewählt und in den sortierten Teil der Liste verschoben. Dieser Prozess wird so oft wiederholt, bis alle Elemente der Liste sortiert wurden.

Das theoretische Konzept sieht wie folgt aus. Als Erstes wird der Index auf 0 gesetzt. Der Index beschreibt hierbei die Stelle, an die das kleinste Element der unsortierten Liste verschoben werden soll. Anschließend durchsucht man den unsortierten Teil der Liste nach dem kleinsten Element. Nachdem die Liste fertig durchsucht wurde, wird das kleinste Element mit dem Element am Index getauscht. Abschließend wir der Index um 1 erhöht. Dieser Ablauf wird so lange wiederholt, bis alle Elemente der Liste sortiert sind. Für das absteigende Sortieren wird nicht nach dem kleinsten, sondern nach dem größten Element gesucht. Dieses wird dann an die Stelle des Index verschoben.

Die Performance des Selection Sort liegt bei $O(N^2)$ und die Speicherkomplexität bei $O(1)$. Zudem ist der Selection Sort Algorithmus instabil. Das bedeutet, dass die Reihenfolge gleicher Elemente nicht garantiert beibehalten wird.

ii. Bubble Sort

Bei dem Bubble Sort wird bei jedem Durchlauf das größte Element der unsortierten Liste in den sortierten Teil der Liste verschoben. Dieser Ablauf wird so lange wiederholt, bis alle Elemente sortiert sind.

Das theoretische Konzept des Bubble Sort ist wie folgt. Es wird am Index 0 gestartet, dem ersten Element der Liste. Dieses wird mit dem Nachbarelement verglichen. Das Nachbarelement ist in diesem Fall immer das Element rechts des Elements am Index. Sind diese beiden nicht in der richtigen Reihenfolge, werden sie getauscht. Anschließend wird der Index um 1 erhöht und auch dieses Element wird mit seinem Nachbarelement verglichen und getauscht, falls diese nicht in der richtigen Reihenfolge sind. Dieser Prozess wird so lange durchgeführt, bis das letzte Element erreicht ist. Nun ist das hinterste Element der Liste sortiert und wir starten wieder am Index 0. Dieser Vorgang wird

so lange wiederholt, bis alle Elemente der Liste sortiert sind, also das Element am Index 0 und das Nachbarelement in der richtigen Reihenfolge sind.

Auch hier liegt die Performance, wie beim Selection Sort, bei $O(N^2)$ und die Speicherkomplexität bei $O(1)$. Allerdings ist der Bubble Sort stabil. Die Reihenfolge gleicher Elemente bleibt also bestehen.

iii. Insertion Sort

Bei dem Insertion Sort wird nachfolgend immer ein Element aus dem unsortierten Teil der Liste genommen und an die richtige Stelle im sortierten Teil der Liste eingeordnet.

Das theoretische Konzept dahinter lautet wie folgt. Man geht davon aus, dass das erste Element bereits sortiert ist und somit zum sortierten Teil der Liste gehört. Der sortedHighIndex wird also zu Beginn auf 1 gesetzt. Man nimmt das erste Element aus dem unsortierten Teil der Liste. Dieses hat den Index Low + 1. Nun werden alle Elemente im bereits sortierten Bereich nacheinander mit dem Element am aktuellen Index verglichen. Ist der Ort für das Element gefunden, wo es in die Reihe der sortierten Elemente passt, wird es dort einsortiert und der sortedHighIndex um 1 erhöht. Dieser Ablauf wird so lange wiederholt, bis alle Elemente sortiert wurden.

Der Insertion Sort hat ebenfalls eine Performance von $O(N^2)$ und eine Speicherkomplexität von $O(1)$. Vorteil des Insertion Sort ist, dass im Nachhinein angefügte Elemente einfach in die bereits sortierte Liste einsortiert werden können. Zudem ist der Algorithmus stabil.

iv. Quick Sort

Bei dem Quick Sort handelt es sich um einen rekursiven Algorithmus. Er basiert auf dem Paradigma Divide and Conquer, was besagt, dass ein komplexes Problem in mehrere kleine Probleme aufgeteilt wird.

Es wird ein beliebiges Pivotelement gewählt. An diesem wird die zu sortierende Liste aufgeteilt. Anschließend wird jedes Element mit dem Pivotelement verglichen. Ist dieses Element kleiner als das Pivotelement, wird es auf die linke Seite des Pivotelements

verschoben, wenn es größer ist in den rechten Teil. Danach werden beide Teile nacheinander sortiert, wobei mit dem linken Teil angefangen wird.

Das theoretische Konzept dahinter sieht wie folgt aus. Wenn nur noch 1 Element übrig ist, erreichen wir den Base Case und rufen “return“ auf. Der Pre Recurse beim Quick Sort ist das Aufteilen der Liste in den linken und rechten Teil. Beim Recurse wird zuerst die linke und dann die rechte Seite bearbeitet.

Der Quick Sort hat eine Speicherkomplexität von $O(1)$. Die Zeitkomplexität liegt im Worst Case auch bei $O(N^2)$, allerdings ist der Quick Sort effizienter bei großen Datenmengen. Im Average bzw. Best Case hat er eine Zeitkomplexität von $O(N \log N)$. Zudem ist der Quick Sort nicht stabil.

v. Merge Sort

Auch der Merge Sort ist ein rekursiver Algorithmus. Gleich wie der Quick Sort bedient er sich auch an dem Paradigma Divide and Conquer. Beim Merge Sort wird der Input wiederholt in der Mitte aufgeteilt, bis nur noch Listen mit einem Element vorliegen. Anschließend werden immer 2 Listen wieder miteinander verbunden und dabei in sich sortiert. Dies wird so lange wiederholt, bis am Ende wieder eine Liste entstanden ist. Diese ist nun sortiert.

Das theoretische Konzept hinter dem Merge Sort sieht wie folgt aus. Wenn nur noch ein Element übrig ist, erreichen wir den Base Case und rufen “return“ auf. Der Pre Recurse ist wie beim Quick Sort das Aufteilen der Liste in einen linken Teil und rechten Teil. Beim Recurse wird zuerst die linke und dann die rechte Seite bearbeitet. Der abschließende Post Recurse führt die beiden sortierten Seiten wieder zusammen.

Die Zeitkomplexität beim Merge Sort beträgt $O(N \log N)$ und die Speicherkomplexität $O(N)$. Zudem ist der Algorithmus stabil. Allerdings ist der Speicherbedarf beim Merge Sort hoch, da es sich nicht um einen In-Place Sort handelt und somit Kopien erstellt, welche anschließend wieder überschrieben werden. Der Merge Sort kann auch parallel ausgeführt werden, indem Teillisten aus dem Teilvorgang auf verschiedene Rechner aufgeteilt werden. Auf den Rechnern können die Listen dann weiter geteilt und wieder zusammengeführt werden.

vi. Auswahl des Sortieralgorithmus

Wie in der Einleitung bereits beschrieben, muss der Algorithmus folgende Anforderungen erfüllen. Es sollen Datensätze mit 1 Milliarden Zeilen aggregiert werden und der Algorithmus soll in der Lage sein, die Sortierung parallel zu verarbeiten.

Um eine Datenmenge von 1 Milliarde Datensätze möglichst effizient zu bearbeiten, muss die Zeitkomplexität des Algorithmus möglichst gering sein. Hierbei kommen der Quick Sort und Merge Sort in Frage, da beide eine Zeitkomplexität von $O(N \log N)$ haben. Zudem soll die Sortierung parallel verarbeitet werden können. Dies ist mit dem Quick Sort nicht möglich. Der Merge Sort hingegen ist in der Lage die Teillisten auf mehrere Rechner aufzuteilen. Dadurch kann die Sortierung parallel verarbeitet werden.

Zu den Anforderungen passt folglich der Merge Sort am besten. Diesen werde ich im Folgenden auf Basis meiner Problemstellung implementieren.

3. Implementierung

i. Klasse Row

```
public class Row {

    public String brand;
    public String modell;
    public int hp;

    public Row(String brand, String modell, int hp){
        this.brand = brand;
        this.modell = modell;
        this.hp = hp;
    }
}
```

ii. Klasse Aggregated Row

```
public class AggregatedRow {

    public String brand;
    public String modell;
    public int hp;

    public AggregatedRow (String brand, String modell, int hp){
        this.brand = brand;
        this.modell = modell;
        this.hp = hp;
    }
}
```

iii. Klasse Aggregator

```
public class Aggregator {

    public ArrayList<AggregatedRow> aggregate (ArrayList<Row> rows){
        ArrayList<AggregatedRow> aggregatedRows = new ArrayList<>();

        for (String brand : getBrands(rows)) {
            aggregatedRows.add(new AggregatedRow(brand, null, 0));
        }

        for(int i = 0; i < aggregatedRows.size(); i++){
            for (Row row : rows) {
                if (aggregatedRows.get(i).brand == row.brand) {
                    if(aggregatedRows.get(i).hp < row.hp){
                        aggregatedRows.get(i).modell = row.modell;
                        aggregatedRows.get(i).hp = row.hp;
                    }
                }
            }
        }
        return aggregatedRows;
    }

    private ArrayList<String> getBrands(ArrayList<Row> rows){
        ArrayList<String> brands = new ArrayList<>();

         for (Row row : rows) {
            if(brands.contains(row.brand) == false){
                brands.add(row.brand);
            }
        }
        return brands;
    }
}
```

iv. Klasse Sorter

```
public class Sorter {

    public void sort(ArrayList<AggregatedRow> rows){
        mergeSort(rows);
    }

    private ArrayList<AggregatedRow> mergeSort(ArrayList<AggregatedRow> rows) {

        int length = rows.size();
        if (length < 2) {
            return rows;
        }

        int middleIndex = length / 2;
        ArrayList<AggregatedRow> leftSide = new ArrayList<>(rows.subList(0, middleIndex));
        ArrayList<AggregatedRow> rightSide = new ArrayList<>(rows.subList(middleIndex, length));

        mergeSort(leftSide);
        mergeSort(rightSide);
        merge(rows, leftSide, rightSide);

        return rows;
    }

    private void merge(ArrayList<AggregatedRow> rows, ArrayList<AggregatedRow> leftSite, ArrayList<AggregatedRow>
rightSide) {

        int leftIndex = 0;
        int rightIndex = 0;
        int index = 0;
        int leftSize = leftSite.size();
        int rightSize = rightSide.size();

        while (leftIndex < leftSize && rightIndex < rightSize) {
            if (leftSite.get(leftIndex).hp <= rightSide.get(rightIndex).hp) {
                rows.set(index, rightSide.get(rightIndex));
                rightIndex++;
            } else {
                rows.set(index, leftSite.get(leftIndex));
                leftIndex++;
            }
            index++;
        }

        while (leftIndex < leftSize) {
            rows.set(index, leftSite.get(leftIndex));
            leftIndex++;
            index++;
        }

        while (rightIndex < rightSize) {
            rows.set(index, rightSide.get(rightIndex));
            rightIndex++;
            index++;
        }
    }
}
```

4. Fazit

Die Problemstellung war, einen Sortieralgorithmus zu implementieren, der Autos absteigend und gruppiert nach PS pro Marke sortiert. Dabei soll der Sortieralgorithmus folgenden Anforderungen gerecht werden: Er soll 1 Milliarde Zeilen aggregieren können und in der Lage sein, die Sortierung parallel zu verarbeiten.

Zwischen den Sortieralgorithmen Selection Sort, Bubble Sort, Insertion Sort, Quick Sort und Merge Sort habe ich mich für den Merge Sort entschieden. Dieser erfüllt die Anforderungen an den Sortieralgorithmus am besten.

Als Erstes habe ich die Klassen Row und AggregatedRow erstellt. Die Klasse Row bildet ein Auto mit den Attributen Marke, Modellname und PS ab. Die Klasse AggregatedRow bildet ebenfalls ein Auto mit den Attributen Marke, Modellname und PS ab.

Anschließend habe ich die Klasse Aggregator implementiert. Diese verarbeitet eine Liste von Objekten der Klasse Row und erstellt eine Liste von Objekten der Klasse AggregatedRow. Hierbei wird aus der Liste von Rows pro Marke ein Objekt der Klasse AggregatedRow generiert. Darin wird das Auto der Marke mit den meisten PS gespeichert.

Abschließend habe ich die Klasse Sorter erstellt. Diese sortiert eine Liste von Objekten der Klasse AggregatedRow. Für die Sortierung habe ich den Merge Sort Algorithmus implementiert.

Erzeugt man nun eine Liste von Rows und übergibt diese an den Aggregator, bekommt man eine Liste von AggregatedRows zurück. Diese Liste wird nach dem Merge Sort Algorithmus sortiert, indem man die Klasse Sorter aufruft und dieser Klasse die Liste an AggregatedRows übergibt.

Erklärung über die eigenständige Erstellung der Projektarbeit

Hiermit versichere ich, dass ich die vorliegende Projektarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Schlaitdorf, den 20.05.2024

Felix Keller