Inclusion Dependencies Identification using Spark

Department of Computer Science and Mathematics
Philipps-Universität Marburg

Course: Big Data Systems
Instructor: Prof. Dr. Thorsten Papenbrock

submitted by Nikita Galaganov (3873245) and Erik Maiterth (3695656)

February 10, 2025

Contents

1 Introduction

Ensuring data integrity in large-scale relational databases is crucial. This report presents a system designed to discover Inclusion Dependencies (INDs) using **Apache Spark**, a distributed computing framework. The system processes large CSV datasets to efficiently identify column dependencies.

2 System Design and Architecture

The system follows a structured pipeline for IND discovery:

1. **Data Reading:** CSV files are parsed into Spark DataFrames.
2. **Data Transformation:** Cell values are converted into tuples of the form `(cellValue, Set(columnName))`.
3. **Dependency Generation:** Grouping by cell value merges the column sets, generating candidate dependencies.
4. **Dependency Reduction:** For each column, the intersection of candidate dependency sets is computed.
5. **Result Output:** The final INDs are collected, sorted, and displayed.

3 Sindy Algorithm Overview

The Sindy algorithm is a scalable method for discovering inclusion dependencies in large datasets. It leverages Apache Spark's distributed processing capabilities to identify relationships between columns based on overlapping cell values. In essence, the algorithm operates as follows:

- **Extraction:** Each cell value is paired with its corresponding column name, forming tuples of the type `(cellValue, Set(columnName))`.
- **Candidate Generation:** By grouping these tuples by cell value, the algorithm determines the set of columns in which a particular value appears. For each value, candidate dependency pairs are generated by subtracting the current column from the set.
- **Dependency Reduction:** The algorithm computes the intersection of candidate dependency sets for each column. A non-empty intersection indicates that every value in a column is also present in the intersected columns, establishing an inclusion dependency.

This approach efficiently uncovers dependencies such as `Column_A < Column_B, Column_C`, meaning every value in `Column_A` is also found in `Column_B` and `Column_C`.

4 Implementation Details

The core function `discoverINDs` implements the IND discovery pipeline. The following code listing shows the main steps of the implementation.

```scala
package de.ddm

import org.apache.spark.sql.{Dataset, Row, SparkSession, Encoder, Encoders}
import org.apache.spark.sql.functions._

object Sindy {
  implicit val tupleEncoder: Encoder[(String, Set[String])] =
    Encoders.tuple(Encoders.STRING, Encoders.kryo[Set[String]])

  def discoverINDs(inputs: List[String], spark: SparkSession): Unit = {
    import spark.implicits._

    // Step 1: Read CSV files and transform rows into (cellValue, Set(columnName)) tuples.
    val extractedData: Dataset[(String, Set[String])] = inputs
      .map { input =>
        spark.read
          .option("inferSchema", "false")
          .option("header", "true")
          .option("quote", "\"")
          .option("delimiter", ";")
          .csv(input)
          .flatMap { row =>
            row.schema.fieldNames.map { col =>
              (row.getAs[Any](col).toString, Set(col))
            }
          }
      }
      .reduce(_ union _)

    // Step 2: Group by cell value and generate dependency candidates.
    val dependencyCandidates = extractedData
      .groupByKey { case (cellValue, _) => cellValue }
      .mapGroups { case (_, tuples) =>
        val mergedColumns = tuples.flatMap(_._2).toSet
        mergedColumns.map(col => (col, mergedColumns - col))
      }
      .flatMap(identity)

    // Step 3: For each column, compute the intersection of dependency candidates.
    val indCandidates = dependencyCandidates
      .groupByKey { case (col, _) => col }
      .mapGroups { case (col, depIter) =>
        (col, depIter.map(_._2).reduce(_ intersect _))
      }
      .filter { case (_, deps) => deps.nonEmpty }

    // Step 4: Collect, sort, and output the final INDs.
    val finalINDs = indCandidates
      .collect()
      .map { case (col, deps) => (col, deps.toList.sorted) }
```

```
      .sortBy(_._1)

    finalINDs.foreach { case (col, deps) =>
      println(s"$col < ${deps.mkString(", ")}")
    }
  }
}
```

Listing 1: Sindy Implementation

5 Advantages Compared to Alternative Approaches

While techniques such as the Longest Common Substring algorithm are useful for string similarity tasks, the Sindy approach offers several advantages in the context of inclusion dependency discovery:

- **Scalability:** Sindy is designed to run on Apache Spark, enabling efficient distributed processing of very large datasets.
- **Applicability:** No like the Longest Common Substring algorithm—which focuses on character-level similarity between two strings—Sindy operates on cell values across multiple columns to identify inclusion relationships.
- **Efficiency:** By leveraging grouping and set intersection operations, Sindy quickly filters out irrelevant candidates and hones in on true inclusion dependencies.
- **Robustness:** The algorithm is tailored for structured data in relational databases, making it more robust to the diverse and noisy data often encountered in real-world datasets.
- **Flexibility:** Sindy's design allows for easy extension to additional data formats (e.g., JSON) and integration with further optimization strategies such as caching and indexing.

6 Future Scope

Possible Enhancements:

- Extend support to additional data formats (e.g., JSON).
- Optimize performance through caching and improved indexing strategies.
- Explore real-time IND detection for streaming data.