Master Thesis

Distributed Unique Column Combinations Discovery

Verteiltes Finden Einmaliger Spaltenkombinationen

Benjamin Feldmann
benjamin.feldmann@student.hpi.de

Handed in on 15.01.2020

Master of Science
IT-Systems Engineering

Information Systems Group
Digital Engineering Faculty
University of Potsdam

Supervisor: Dr. Thorsten Papenbrock
Advisors: Dr. Thomas Bläsius, Martin Schirneck

Abstract

Finding minimal unique column combinations (UCCs) is a data profiling task that is important for applications such as data deduplication, data cleansing, and database query optimization. In this thesis, we introduce the highly scalable distributed algorithm HitUCC, that discovers unique column combinations.

HitUCC uses the actor model to detect unique column combinations in parallel. We use a two-phase approach that first compares each row with every other row in a data set in the first phase and uses the results of the comparisons to prune the search space and discover the unique column combinations in the second phase. HitUCC outperforms modern algorithms on datasets with many columns and complex unique column combinations. We measure the scalability of HitUCC on a computing cluster, and show that the algorithm is highly scalable and that it can achieve a nearly linear speedup.

Zusammenfassung

Das Finden von minimalen eindeutigen Spaltenkombinationen (UCCs) ist eine Data-Profiling-Aufgabe, die für Anwendungen wie Datendeduplizierung, Datenbereinigung und Datenbankabfrageoptimierung wichtig ist. In dieser Arbeit stellen wir den hochskalierbaren verteilten Algorithmus HitUCC vor, der einzigartige Spaltenkombinationen findet.

HitUCC nutzt das Aktormodell, um parallel eindeutige Spaltenkombinationen zu erkennen. Wir verwenden einen zweistufigen Ansatz, der in der ersten Phase jede Zeile mit jeder anderen Zeile in einem Datensatz vergleicht und die Ergebnisse der Vergleiche nutzt, um in der zweiten Phase den Suchraum zu beschneiden und die eindeutigen Spaltenkombinationen zu entdecken. HitUCC übertrifft moderne Algorithmen bei Datensätzen mit vielen Spalten und komplexen eindeutigen Spaltenkombinationen. Wir messen die Skalierbarkeit von HitUCC auf einem Rechencluster und zeigen, dass der Algorithmus hoch skalierbar ist und einen nahezu linearen Geschwindigkeitszuwachs aufweisen kann.

Contents

1 Introduction

Nowadays, many applications generate a lot of relational data (e.g., sensor data, social network data). The resulting datasets do have not only many rows but also many columns. The tables are so large that the data becomes difficult to understand, and most often, structural information or documentation of the data is missing [1]. *Data profiling* is the field of discovering metadata for databases [1]. A fundamental task of data profiling is the discovery of *unique column combinations* (UCCs) from relational data. UCCs are combinations of columns in which no row is a duplicate of another row. If at least one row of a column combination contains the same values as another, we speak of a *non-unique column combination*. Figure 1.1 shows a sample table with three UCCs.

First	Last	Age	Zip Code
Anna	Johnson	21	1234
Lisa	Miller	23	5678
Max	Smith	21	1234
Joel	Miller	24	9876

Figure 1.1: Sample table with {First}, {Last, Age} and {Last, Zip Code} as minimal UCCs

Unique column combinations are essential when it comes to identifying key candidates in a data set. They help to understand the structure of the data. Furthermore, identifying UCCs supports other data management tasks such as duplicate detection, query optimizations, and data cleansing [17, 1, 22]. Knowing these unique combinations helps in other data profiling areas such as dependency detection. Most applications only utilize minimal UCCs. The discovery of all unique column combination is a computationally expensive task. The solution space, i.e., all possible column combinations is exponentially large. A brute-force algorithm would need to test every column combination, to discover all solutions. The problem of the UCCs discovery is both NP-hard and W[2]-hard [8].

Existing state-of-the-art algorithms like HyUCC [22] and Ducc [17] mainly work on one CPU and are therefore limited to the computing power of its machine. The Ducc technique is an exception with a distributed version. In some cases, the discovery of unique column combinations takes so long that the runtime is no longer affordable. For instance, HyUCC, which is the most efficient solution at the moment [1], took more than 8 hours to find all unique column combinations from the *ncvoter* (4,167 GB) dataset and takes around 5,8 hours with parallel uniqueness validation. The Ducc algorithm takes over 8 hours to calculate all uniques on the same machine [22]. The authors used a Dell PowerEdge R620 with 128 GB RAM and 32 cores [22] for the experiments. These long runtimes are increased even further with larger data sets. At the same time, the storage on only one machine may be no longer sufficient. A solution is needed to solve

the problem for larger tables.

One option to improve the performance of the discovery is to distribute an algorithm by splitting the algorithm into several smaller parts and execute the resulting tasks in parallel on multiple machines. A group of all computers that are working together is a *cluster*. A single machine is named a *node* of the cluster. Most of the traditional UCC discovery approaches use a *lattice* representation of the search space, which is an interconnected graph consisting of all possible column combinations. Every approach relies heavily on pruning the search space to avoid enumerating every combination. An example is that after identifying column A as a unique column, the algorithm does not need to check any superset of the column A, since it would not be minimal.
Since we are aiming for a more efficient solution, we could attempt to distribute an algorithm that is based on a lattice represented search space. Unfortunately, a lattice is very strongly interconnected and is therefore difficult to divide into different parts that can be executed in parallel. Additionally, these algorithms rely heavily on the pruning of the search space, which would result in much network communication. Bläsius et al. as introduced an undistributed algorithm that first compares each row with every other in a dataset as a preprocessing, and executes a tree search to afterward. The comparisons can be well partitioned into subtasks, and a tree is more distributable than a lattice because the subtrees are distinct and not connected to each other. Therefore, we use this approach as a baseline to better benefit from parallel processing.

1.1 Contributions and Structure

This master thesis presents the scalable *unique column combination* discovery algorithm HitUCC. We distribute the algorithm across multiple machines and compare the result with non-distributed state-of-the-art algorithms. The specific contributions we present are:

1. We propose a distributed, scalable two-phase algorithm for the discovery of all unique column combinations based on the actor model and the introduced approach from Bläsius et al.

2. We present the division of the first phase into subtasks and partition the data accordingly to achieve optimized use of all resources on the cluster. We describe a peer-to-peer based merge process of all data and provide an evaluation on different partition sizes.

3. We propose the reactive data transfer strategy, which optimizes the timing of the transfer of individual batches and ensures that not all data is transferred over the network at the same time using peer-to-peer based communication.

4. We propose a distributed tree search and describe the task partitioning that includes a peer-to-peer task redistribution strategy using work pulling.

For this purpose, we define the context of this work and provide a historical outline of the discovery of unique column combinations in Section 2. We then describe all the foundations necessary for this work and explain the baseline algorithm we are distributing in Section 3. Afterward, we present the architecture of HitUCC and clarify our technical assumptions of the cluster that executes our algorithm. Furthermore, we describe how to start the cluster and get all nodes to work together in Section 4. Section 5 focuses on the first part of the algorithm. We explain the used data distribution model and how we synchronize all involved nodes. Additionally, the chapter describes the reactive data transfer strategy in detail. Section 6 concentrates on the second part of HitUCC. The section describes the distributed tree search with minimal network overhead. Section 7 provides our evaluation of the algorithm. At the end, we summarize this thesis and offer a few possibilities for future work in Section 8.

2 Related Work

This section provides an overview of related work. We roughly describe five unique column combination detection algorithms with a higher emphasis on the HyUCC algorithm, our main competitor. Additionally, we divide all presented algorithms into three different classes: Column-based, Row-based, and hybrid approaches [1]. Finally, we describe a non-traditional approach that reduces the discovery of unique column combinations to an enumeration problem on hypergraphs.

2.1 Different Approaches

Colum-based: Most algorithms that are column-based, use a lattice as their representation of the search space. A lattice is an interconnected graph consisting of all possible column combinations is often generated with the *apriori* approach [4]. Figure 2.1 shows a lattice search space of 4 columns $\{A, B, C, D\}$ visualized in a *Hasse diagram* [6]. The HCA [2], Ducc [17] and the Swan [3] algorithms traverse the lattice to collect all minimal unique column combinations. The column-oriented algorithms are all using pruning techniques to minimize the search space. These three algorithms can be distinguished by each other in a different order of the lattice traversal. Colum-based algorithms are usually slower for datasets with many columns and faster for datasets with many rows.

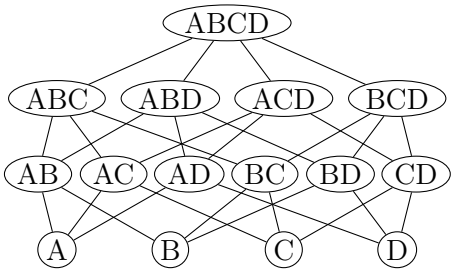


Figure 2.1: Lattice search space with 4 columns

Row-based: These algorithms compare all rows in pairs in a dataset to create non-unique column combinations and derive all minimal UCCs from the non-unique UCCs [1]. Row-based approaches are efficient on datasets with few rows because of less non-unique candidates. In this section, we focus on the Gordian [26] algorithm.

Hybrid: The HyUCC [22] algorithm combines both approaches to utilize the strengths of each by alternating between testing column combination candidates and comparing rows in a dataset.

2.2 UCC Discovery Algorithms

Sismanis et al. introduced in 2006 the row-based UCC discovery algorithm Gordian [26] that first recognizes all non-unique column combinations and then forms uniques from them. When testing whether a column combination contains duplicates, the algorithm can stop after finding the first duplicate value. Therefore, the process of finding an actual unique column combination is usually slower. Afterward, the Gordian derives all UCCs from the maximal non-uniques.
The HCA algorithm (introduced in 2011) is a column-based approach that traverses the lattice from bottom to top as it starts by checking all combinations with only one column. The algorithm removes all unique column combinations from the current candidate set and generates the next level of candidates from the current column combinations and repeats until the candidate set is empty or the traversal completes.
In 2013, Heise et al. presented the column-based Ducc [17] Algorithm. The approach uses a depth-first random walk to traverse the lattice that allows the algorithm to prune upwards and downwards. The minimized search space results in higher performance than Gordian and HCA [17]. The authors also proposed a distributed version of the algorithm with in parallel executed random walks.
Abedjan et al. introduced the Swan [3] algorithm in 2014. Swan is an incremental algorithm that monitors changes in a dataset. Inserting rows can invalidate the original unique column combinations. On the other hand, a row deletion could change a non-unique combination into a UCC.

The HyUCC [22] algorithm uses a hybrid approach (2017) to alternate between a row- and a column-based strategy. In the row-based phase, HyUCC compares rows pairwise to form *agree sets*. An agree set specifies the positions in which two rows have the same value and corresponds to a non-unique column combination. The column-based strategy is similar to the HCA algorithm and starts the lattice traversal from bottom to top. The hybrid strategy combines both approaches by using the row-based search as a sampling phase and validate UCCs via the column-based phase before returning to the sampling phase. The result of each phase is used to narrow the search space of the others. Due to the superior pruning of the HyUCC, the algorithm outperforms any other of the presented algorithms [22].

The HyUCC transforms all records to *position list indexes* (Plis [18]) and compresses all columns with the dictionary encoding using these Plis. HitUCC also encodes all columns with the dictionary encoding but does not need to calculate Plis first, because our proposed algorithm does not need these indices for the discovering of UCCs.

Additionally, the HyUCC has a distributed implementation [24] based on Apache Spark [16].

2.3 Enumerating Hitting Sets to Discover Unique Column Combinations

Bläsius et al. introduced a way to enumerate Hitting Sets of hypergraphs efficiently in 2019 [7]. They reduce this problem to the discovery of unique column combinations. We are basing our distributed algorithm on their undistributed approach. Section 3.2 of the foundation chapter describes the basics of the Hitting Set Problem and explains the enumeration algorithm in more detail.

3 Foundations

In this section, we introduce the theoretical foundation of unique column combinations. Furthermore, we present the undistributed algorithm of Bläsius et al., on which this master's thesis is built. Additionally, we introduce the actor model, which describes one approach to implement concurrent behavior and compare two different communication strategies on top of the actor model.

3.1 Unique Column Combinations

We use the following notation: A relational schema R consists of an ordered set of attributes. $|R|$ is the cardinality of R. An instance r of R is an ordered set of records (also called rows). $|r|$ is the number of rows in a dataset. X_i denotes the ith attribute in $X \subseteq R$, and we use the word column for a set consisting of the ith value of every record. t_i denotes the ith row of an instance r. We denote individual attributes by capital letters from the start of the alphabet, i.e., A, B, C and sets of columns with upper-case letters starting from the end of the alphabet, i.e., X, Y, Z.
We denote by $t[X]$ the projection on the attribute set X and use the term *column combination* for the projection. A projection on a single attribute A is the set of values from all rows in the column A. In conclusion, a projection of the attribute set X is a union from all projections of all columns in X. A column combination $X \subseteq R$ is *unique* (UCC) if the projection of X contains no duplicated rows. A unique column combination $X \subseteq R$ is *minimal* if every subset $B \subset X$ is not a unique column combination.

Furthermore, for UCCs X and Y, X is a *specialization* of Y if $X \supset Y$ and a *generalization* of Y if $X \subset Y$. Every generalization of a unique column combination is also a unique column combination. Thus it suffices to find all minimal UCCs from a dataset and afterward generating all generalizations when discovering all UCCs.

3.2 Discover Minimal UCCs by Enumerating Hitting Sets

Bläsius et al. have introduced a two-step algorithm that finds all unique column combinations in a dataset [7]. In the first step, the algorithm creates the Hitting Set instance from the dataset. The second step is a validation phase that finds all minimal results, which directly relate to minimal unique column combinations.

Given a *hypergraph* $H = (V, E)$ with vertex set V and set of *hyperedges* E. A *hyperedge* is an edge between any number of vertices of V. A vertex set $S \subseteq V$ is a Hitting Set of H exactly when the intersection of S with each hyperedge is not empty.

3.2.1 Creating the Hitting Set Instance

To discover unique column combinations with the presented approach on relational data, we first need to create a minimal Hitting Set instance. We accomplish this by forming *difference sets* from the data and then minimizing them. A difference set specifies at which positions two rows have different values. We compare each pair of rows to build all sets. Figure 3.1 shows an example of a difference set creation.

A	B	C	D	E
1	2	3	4	5
5	2	4	4	2

$\Longrightarrow$

A	B	C	D	E
1	0	1	0	1

Figure 3.1: Create a difference set from two table rows

Afterward, we need to identify all minimal difference sets from the resulting sets. The minimal sets are sufficient to solve the Hitting Set problem [7], and the enumeration is faster if the amount of all difference sets is as small as possible. Given a list of difference sets L, a difference set $X \in L$ is minimal if no other set $B \in L$ exists, such that $B \subset X$. Thus we need to compare each difference set pairwise to filter out non-minimal difference sets. Figure 3.2 shows a Hitting Set instance in non-minimized and minimal form.

A	B	C	D	E
1	0	1	0	1
1	1	0	0	0
1	1	1	0	1
1	1	0	1	0
0	0	1	0	1

$\Longrightarrow$

A	B	C	D	E
1	1	0	0	0
0	0	1	0	1

Figure 3.2: Minimize all Difference Sets

3.2.2 Enumeration of Unique Column Combinations

Bläsius et al. describe how to identify all minimal unique column combinations from the resulting difference sets. They base the solution validation on an *extension oracle* [7].

The oracle decides whether a column combination $X \subset R$ is unique or if the column combination is extendable to a UCC while excluding the column combination $Y \subset R$ ($X \cap Y = \emptyset$). Extendable means that X can become a unique column combination by adding one or more columns to X as long as they are not in Y. We test the smallest column combinations first and add more and more columns. This approach eliminates the need to test if a found UCC is minimal since a found UCC always has the smallest

possible number of columns. The oracle uses the column combinations X and Y and all minimal difference sets H as input.

Bläsius et al. show that the HITTING SET problem and another cover problem are equal under parameterized reduction k. They call the new cover problem MULTICOLOURED INDEPENDENT FAMILY that formulates the following task: Given k lists of sets, each collection representing a color, and a list of forbidden sets T. The MULTICOLOURED INDEPENDENT FAMILY problem has a solution if there exists a set of each color such that their union does not completely cover any of the forbidden sets. [7]. The oracle creates the MULTICOLOURED INDEPENDENT FAMILY instance from X, Y, and H. If T is empty, the instance is trivial: if none of the k lists contains an empty set, the oracle solved the problem, and X is a minimal UCC. If one of the lists contains an empty set, there is no solution, and X is not extendable. If the oracle found no solution, it solves the problem with a brute force approach. We describe the exact functionality of the oracle in Section 6.2, which explains our implementation of the oracle.

We use a decision tree to list all column combinations that the oracle tests. We start with two empty column combinations X and Y. In each tree traversing step, the oracle checks whether the first column combination is extendable to a UCC while excluding Y. If we find a unique column combination or if an extension is not possible, we can prune the search tree and do not have to enumerate any further column combinations in this path. If the column combination is extendable to a UCC, we add two new decision points to the tree: First, we add the next unused column to the first set of columns X, and second, we add the column to the column combination Y. Figure 3.3 shows the unpruned decision tree for three columns, while Figure 3.4 presents a variant of the tree where the oracle prunes subtrees. A node in the tree of both figures consists of the column combinations X and Y.

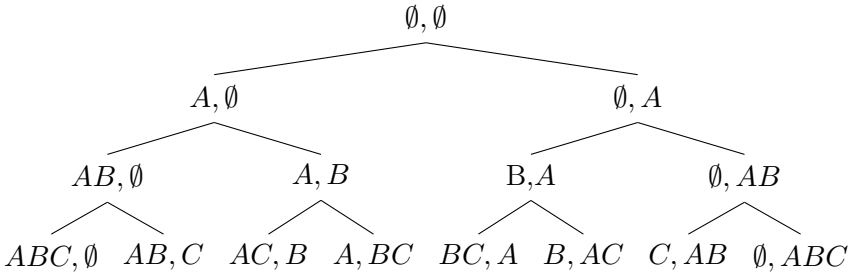


Figure 3.3: Decision tree search space with columns A, B and C.

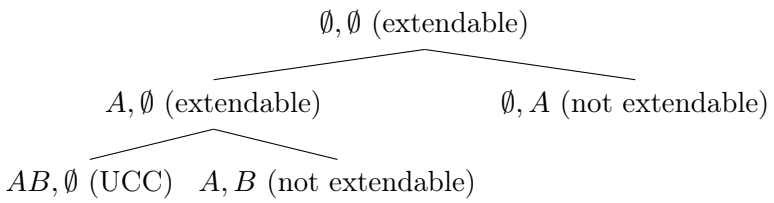


Figure 3.4: Example decision tree with oracle.

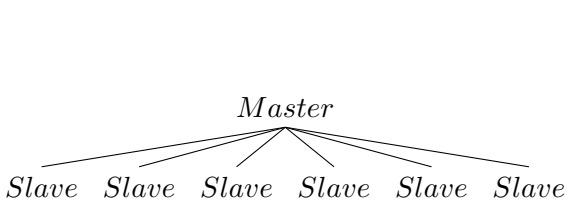
3.3 Distribution Model

In this section, we describe the concurrent computation model named actor model. Finally, we compare two different communication models and explain why we chose the *peer-to-peer* communication over the *master-slave* communication pattern.

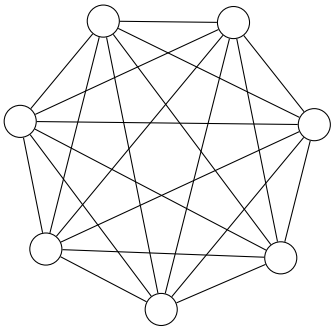
3.3.1 Actor Model

The actor model is a model of concurrent computation and consists of operational units called actors that can send asynchronous messages to each other [28]. Every actor has a mailbox and each message is stored in it when the actor receives it. An actor is an independent computation unit with a private state and is only allowed to act and change th state when the actor reads a message from their mailbox. The mailbox is a message queue with a fixed capacity that caches messages until their actor has the capacity the read the next message. The structure of the system avoids shared memory issues like write-locks because no actor has access to the capsulated state of another actor. A semantic group of actors on a physical machine is an actor-system, and all actors in the system are connected locally. Different systems may be physically located on different machines. Actors are ordered hierarchically in tree structures where parent actors supervise their children and thus handles their failures. The standard approach for error-handling is that the parent restarts a child actor.

We use the actor model as our distribution model since the control of the individual actors is very high. It is possible to decide precisely which actor will handle which tasks and how the data will be distributed. This fact allows us to design and implement a very well-scaled algorithm. Other approaches like batch processing (i.e., Apache Spark [16]) or stream processing (i.e., Apache Flink [14]) allow less control but require less implementation effort [20]. Our implementation of choice of the actor model is Akka [19].



(a) Master — Slave Communication



(b) Decentralized Communication

Figure 3.5: Communication Models

3.3.2 Communication Model

Various communication strategies can be implemented on top of the actor model. Two more prominent are the *master-slave* paradigm and the *peer-to-peer* communication model (Figure 3.5).

Master-Slave The model consists of two logical types of actors: the master actor and the slave actor. The master knows all the necessary tasks and organizes the computation by delegating tasks to the slave actors. Slave actors only compute their tasks and return the results to the master. The model introduces a *single point of failure* by having a master actor. The algorithm cannot recover from a crashing master because the slaves do not know the master state. On the other hand, if a slave crashes, the master knows which task he delegated to the slave and can reassign the task to another slave actor. Another problem is that the system depends on the performance of the master actor. Since all communication passes through the master, the load on the master also increases as the size of the system increases. From a certain point, the master actor may organize the work slower than the slave actors need for their task, and a more extensive distribution is no longer worthwhile.

Peer-to-Peer Differently, the *peer-to-peer* communication model uses a decentralized approach. Every actor is a master and a slave at the same time and can delegate tasks or do work for other actors. Consent is much more challenging to achieve because every actor must agree with each other at some point to coordinate the tasks. On the other hand, the model removes a *single point of failure* by eliminating the centralized state. A system with a *peer-to-peer* approach is highly scalable as no communication actor exists that can be a bottleneck. Due to the high scalability of this strategy, we decided to use the *peer-to-peer* approach instead of the *master-slave* approach.

Independent of both models, we can transfer work packages and data via a *work-pulling* or *work-pushing* strategy. With work-pushing, an actor sends messages to other actors and expects them to complete the task. The advantage of this method is that the receiving actors do not have to wait for messages and always have some work in their message queue and can continue working immediately. But we have to take care that the actor doesn't have too many messages in his message queue; otherwise, we overload the worker, and it can crash. The work-pulling mentality avoids this problem because an actor always asks another actor for messages and therefore has only very few tasks in the message queue. The significant disadvantage of this strategy is that more communication is involved, and the actors always have to ask for new messages before they can take care of the work packages themselves.

4 Architecture and Cluster Start

This section provides an overview of the architecture of the distributed algorithm HitUCC. We describe the input and output of the algorithm and clarify our technical assumptions about the cluster that executes our algorithm.

The algorithm of Bläsius et al. finds all unique column combinations in one data set. We reimplement this algorithm and distribute the execution on a cluster. First, we have to create all minimal difference sets from which we can derive all UCCs. Afterward, we use the decision tree search to find all unique column combinations from the difference sets. We divide the algorithm into phases 1 and 2; Section 5 describes how to distribute the first phase, which is finding all minimal difference sets. To create all sets, we need to compare each row in the data set in pairs and minimize the resulting list if difference sets using a suitable strategy. The intelligent distribution of the data to all nodes, the task distribution, and the subsequent synchronization of the results play a particularly important role in this phase. We describe in Section 6 how we distribute the second phase over the whole cluster. We explain the distribution of the tree search and show how we merge all found unique column combinations on one node for the algorithm output.

4.1 Conceptional Core Components

HitUCC consists of two subsequent two algorithm phases. We can, however, identify six conceptual components that are critical parts of the algorithm phases:

Phase 1:

1. *Initial data partitioning* to send as little data over the network as possible and replicate it to as few nodes as possible.

2. *Reactive distribution and preloading* of the required data to the individual nodes to distribute the network load over the entire runtime of the first phase of HitUCC, which results in little or no waiting time for data.

Phase 2:

3. *Creating and minimize all difference sets* from the data individually on actors.

4. Distributed *merging* of all results after each worker created all minimal difference sets individually.

5. The distributed *tree-search*, consisting of local pruning of subtrees.

6. The *extension oracle*, which uses the minimum difference sets to decide whether a column combination X is extendable to a unique column combination or is already a UCC when we exclude the column combination Y.

Figure 4.1 shows the rough interaction of all components as a flow chart, which we explain in detail in Section 5 and 6. While running, HitUCC does not wait until it found all minimal difference sets (3) before merging all results (4). The two steps run partly in parallel because the algorithm can start merging the results when two actors finished discovering all sets. The reactive data transfer runs in parallel to finding the difference sets. The distributed tree search (5) includes the extension oracle (6).

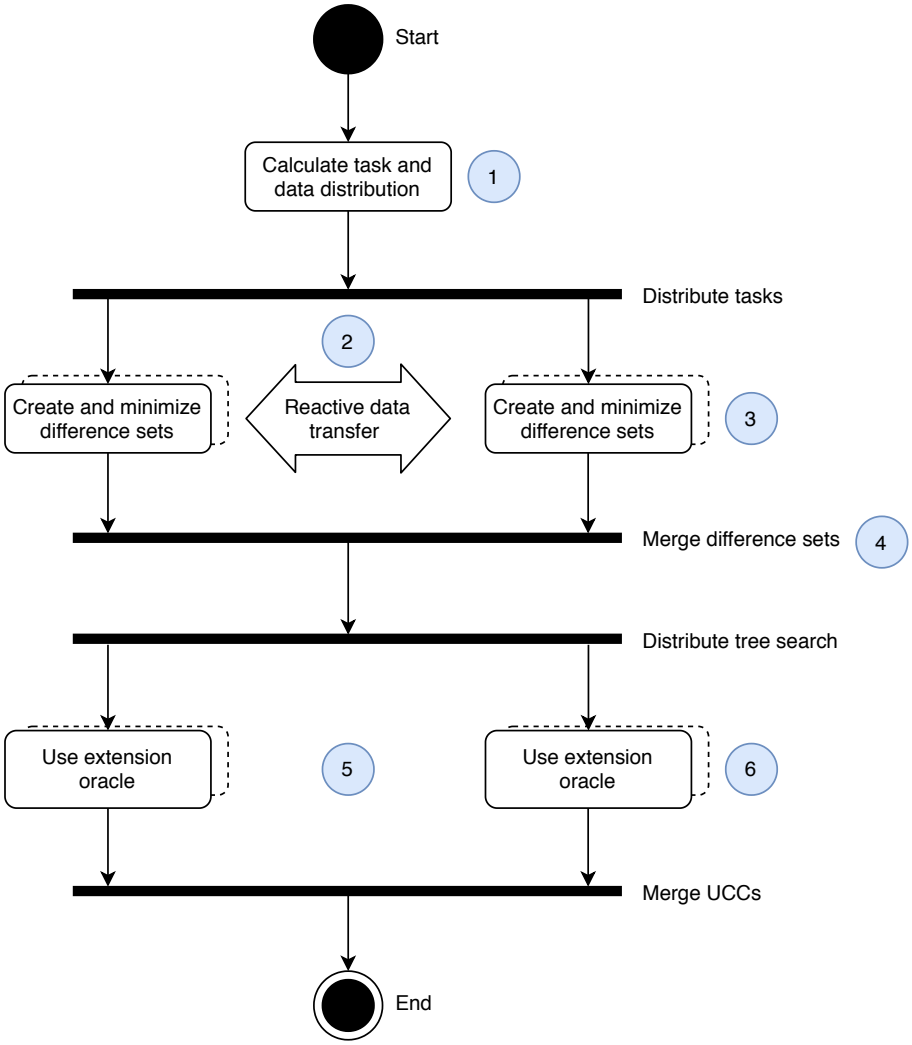


Figure 4.1: Component interaction of HitUCC

4.2 Technical Assumptions

HitUCC runs on a cluster. It uses all available threads on a single node to achieve maximum parallelization without being slowed down by the network. Our algorithm reads a data table in the form of a CSV file on the start node and writes the results to a JSON file on the same node at the end of the algorithm. The start node must have enough memory to read the complete table once and then compress it. The working memory on the other nodes can be much smaller, as they have to process only a part of the compressed data. Our implementation can be executed as a Java Jar application or via Docker [12].

4.3 Actor-System Registration

Before the individual nodes (and the different actors on the nodes) can begin with the algorithm, the nodes must first form a cluster. To do this, we select a node as the *start* node when we start the algorithm and pass the network address to the other nodes.

For each node, there is precisely one actor-system with several actors. We use two different actor types: A *worker* actor who runs the actual creation of the difference sets and an actor we named *data-bouncer* who is responsible for data management and data transfer. Each actor-system consists of precisely one data-bouncer and at least one worker actor. The number of workers depends on the physical machine on which the algorithm is running because each worker needs an own thread to run in parallel to the other actors. We name the worker actors *worker_ i*, where i is the index of the worker of the respective actor-system. The first worker has the name *worker_ 0*. In this section, we describe how to start the cluster. We explain more about the further functionality of the two actor types in Section 5.

First, we start the actor-system on the start node with the information on how many actor-systems should be in the cluster before we can begin finding all unique column combinations. Afterward, we start the actor-systems on all other nodes with the information about the network address of the start node. In each actor-system, we start one data-bouncer and several worker actors independently from other systems. Each worker sends a registration message to the local data-bouncer that waits until it received a registration from all workers in the actor-system. If this is the case, the data-bouncer reports the address information of all local workers to all other local workers, such that every actor in the actor-system can communicate with every other actor.

When the local registration of all workers on a non-start node is complete, the actor-system registers with the start node. To do this, the data-bouncer of that system sends the address information about itself and worker_0 to the start node, which is read by the

data-bouncer of this node. The worker sent along is responsible for the communication between the actor-systems in phase 1. When the data-bouncer on the start system has all information from all other actor-systems, it sends the accumulated address information back to all involved data-bouncers that forward this data to each local worker. The result is that every data-bouncer knows every other data-bouncer in the cluster, and we have created a network for the data transfer between the actor-systems. Furthermore, the worker_0 from an actor-system can communicate with every worker_0 from all other systems. Within a single actor-system, all actors know the address of each other. After the cluster registration, we can start to read the data and discover the unique column combinations.

5 Discovering Difference Sets

This section discusses the construction of minimal difference sets. To create these, we have to compare every row in the table with all others. From the resulting difference sets, we then build the minimal sets. We explain the preprocessing, the individual subtasks of each worker, and how we distribute the required data to all actors. Furthermore, we describe the procedure of a single actor and how it can optimize the process of minimizing the difference sets. Finally, we explain how to synchronize all results for their use in the second phase.

5.1 Task and Data Distribution

In a distributed setting, we have to send data over the network. We first describe a compression technique that reduces the size of the required data and, at the same time, reduces the runtime of creating all difference sets without the need to decompress the data. To create all difference sets, we need to compare every row of a data set with all others. To solve this task in a distributed setting, we have to divide the comparison of rows into sub-tasks that we can run in parallel on different nodes. We describe the distribution strategy of HitUCC, which takes the number of all available actors into account and minimizes the waiting time for the network transfer.

5.1.1 Data Compression

Before the algorithm starts sending the data to all other workers, it first compress the data. Due to the reduced size of the compressed table, we can send the data faster over the network. One technique to compress a table is *dictionary encoding*, which encodes each column individually. A dictionary encoded column consists of a sorted list of all unique elements of the original column (*dictionary*) and a list that contains references to the position of values in the dictionary (*mapping vector*) [23]. The largest number in the mapping vector is therefore equal to the length of the dictionary minus one. Figure 5.1 shows an example of the dictionary compression. The data-bouncer actor of the actor-system, which reads the data and starts the algorithm, compresses the data.

The dictionary encoding is excellent for our application, because we need only the mapping vector to compare two rows. Each value in the mapping vector is a number that points to the actual value in the dictionary. It is sufficient to compare the positions from the mapping vector to determine if two values are equal or not. This process not only saves memory space by compressing the data but also avoids the need for decompressing it afterward. Storage optimizations are always crucial in a distributed setting, because

A	B
Tree	Apple
House	Apple
Tree	Banana
Car	Apple
Bicycle	Peach
House	Apple

$\Longrightarrow$

	A
Bicycle	3
Car	2
House	3
Tree	1
	0
	2

	B
Apple	0
Banana	0
Peach	1
	0
	2
	0

Figure 5.1: Compress two columns with the dictionary encoding

we need to send the data over the network, and smaller data sizes indicate a faster transfer. On top of the storage space optimizations, the comparison of two numbers is much faster than the comparison of two strings. Thus, we also accelerate the building of the difference sets.

To execute a dictionary encoding on a column, we first need all the unique values from that column to build the dictionary. To achieve this, we duplicate the column, sort it, and then remove all duplicates. Afterward, we create a new list with length equal to the length of the original column. For each element in the column, we search the dictionary for the position of the element. Subsequently, we insert the found index into the new list. Since the dictionary is sorted, we can use binary search.

5.1.2 Task Distribution

After compressing the data, we have to distribute it. Xu Chu et al. introduced a method to partition data for their distributed data deduplication algorithm (Triangle Distribution Strategy) [10]. This method is suitable for us, because they compare all rows pairwise too. Therefore, we base our implementation on their approach. The triangle distribution strategy partitions the table into multiple batches and distributes the different parts to different nodes.

The proposed strategy partitions the task of creating all difference sets into several subtasks and distributes them to different nodes. A sub-task consists of two data packages A and B, from which we compare each row from A with every row from B. The number of subtasks is derived from the number of batches in which we divide the compressed data set. HitUCC compares each data batch with itself and with every other data set. We can perform these comparisons independently and in parallel. Figure 5.2 shows a subdivision of the data in five batches and consequently, in 15 sub-tasks.

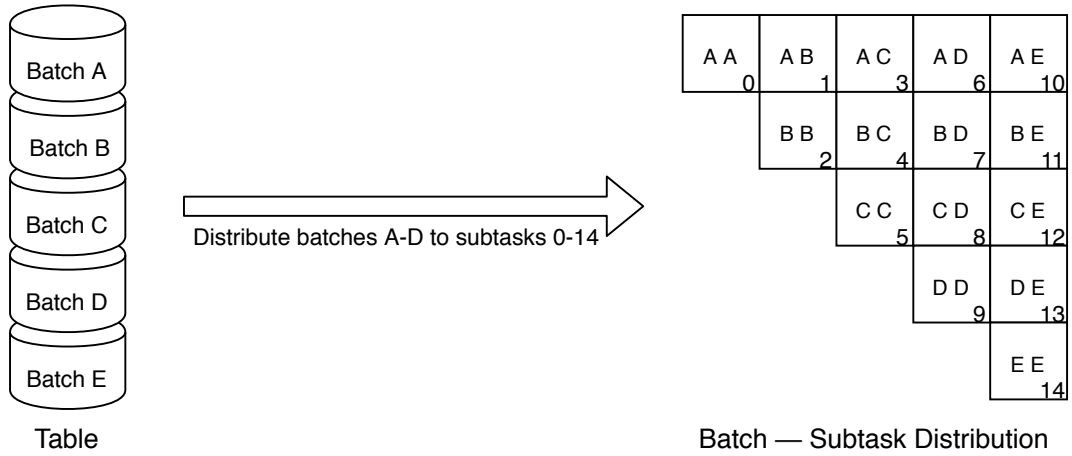


Figure 5.2: Batch distribution to 15 subtasks

The advantage of this strategy is that most of the sub-tasks are of equal size. Each of the data batches has approximately the same number of rows, and therefore the complexity of comparing two batches is always the same. Given two data batches with the number of rows n and m, the algorithm must perform $n \cdot m$ comparisons. The only exception is when comparing rows within the same data batch. In this case, the algorithm needs to compare each row from one batch with every other row from the same batch, resulting in $k \cdot (k-1) \div 2$ comparisons for a batch with k rows, which is slightly less than half of the $n \cdot m$ comparisons. Although, with a large number of subtasks, there are comparatively only a few subtasks that require fewer comparisons. The sequence of the possible numbers of sub-tasks is the sequence of the triangular numbers. A triangular number is the sum of all subsequent natural numbers from 1 to n and the equation of the nth triangular number is $n \cdot (n+1) \div 2$. Given k data batches we have to make $k \cdot (k+1) \div 2$ batch comparisons, which is the same formular. In the example in Figure 5.2 we created 15 tasks out of 5 batches and $5 \cdot (5+1) \div 2 = 15$ confirms this.

At the same time, the used strategy limits us in distribution opportunities. Since the number of subtasks is always a triangular number, and the sequence of these numbers *(1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, ...)* is not equal to the natural numbers, we cannot guarantee that the number of sub-tasks is equal to the number of available worker actors. Given 13 workers and 15 sub-tasks, 11 workers have to do one sub-task each, whereas 2 of the workers have to work on two sub-tasks, resulting in 11 idle workers that have to wait until the other workers finished their tasks. If we divide the compressed data into 12 data batches, we get $12 \cdot (12+1) \div 2 = 78$ sub-tasks that we can distribute evenly among the 13 workers, such that each worker has to do exactly six tasks, while ensuring that each of the workers gets two tasks that require less comparisons than the others. We want to have more tasks than workers, as this reduces the runtime considerably. We will discuss the reason for this in section 5.2 show the evaluation on this optimization in

Section 7.4.2.

The data-bouncer on the start node creates the batches after compressing the data. For this, the actor uses a *brute-force* approach by creating subsequently triangular numbers and test if the resulting number is a multiple of the number of all worker actors in the cluster. The data-bouncer ensures that the number of sub-tasks is not equal to the number of all workers by skipping that triangular number. After the actor has calculated the number of batches, it loops over the compressed data and randomly assigns each row to a batch. Afterward, the data-bouncer creates a list of all sub-tasks, whereby the actor does not store a batch but the index of the batch in a subtask.

For the sub-task distribution, we use the locality of an actor-system. Within one actor-system, there is no network transfer cost since all actors can access the same memory. This means that if several actors from the same actor-system need the same data, we do not have to transfer the data to the actor-system again. We distribute the sub-tasks to the individual workers in a manner that a data batch needs to be distributed to as few actor-systems as possible to reduce the data that we need to transfer over the network. To realize this idea, HitUCC uses a *greedy distribution strategy*: The data-bouncer creates a list of subtasks for each other actor-system in the cluster. Furthermore, the actor finds the best possible sub-task for each actor-system and adds them *round-robin* to the respective actor-system list. The best possible sub-task is a task that increases the number of required data stacks the least. The actor discovers this sub-task by first selecting a random not-distributed task and scoring it to determine whether that subtask requires none, one, or two data batches that are not yet in the list for the actor-system. Then the actor iterates over the list of all not-distributed sub-tasks and attempts to find another task with a lower score. The data-bouncer adds the sub-task with the lowest score to the list for the actor-system. The actor repeats these steps until it distributed all subtasks minus the number of tasks the start node should work on. All data batches are already stored on that node, and thus a data distribution optimization is not possible. The data-bouncer adds all undistributed sub-tasks to the list of the start node and transfers the tasks *round-robin* to each worker of an actor-system from the respective actor-system list. Algorithm 5.1 shows the greedy distribution strategy for all actor-systems except the start one in pseudo-code.

Algorithm 5.1: Greedy Distribution Strategy

Input: List of lists of sub-tasks for each actor-system except the start actor-system $tasksPerActorSystem$, List of all sub-tasks $taskList$, task count for the start actor-system $leftOutCount$

```
1  index ← 0;
2  while taskList.length > leftOutCount do
3      currentTask ← random task fom taskList;
4      currentScore ← calculateScore(currentTask, tasksPerActorSystem[index]);
5      foreach task ∈ taskList do
6          testScore ← calculateScore(task, tasksPerActorSystem[index]);
7          if testScore < currentScore then
8              currentTask ← task;
9              currentScore ← testScore;
10     taskList.Remove(currentTask);
11     tasksPerActorSystem[index].Add(currentTask);
12     index ← (index + 1) mod tasksPerActorSystem.length;
```

5.2 Reactive Data Transfer Strategy

The data-bouncer is responsible for data management and data transfer. The actor stores all for the actor-system necessary batches and can be requested for data by another worker actor from the same actor-system. If the data-bouncer has already stored a required batch, it returns this batch to the worker. If not, the actor asks one of the other data-bouncers (randomly selected) and then forwards the batch asynchronously back to the worker. If the data-bouncer has received a subtask from another data-bouncer, it broadcasts the information to all data-bouncers in the cluster. Thus, every data-bouncer knows where it can request which DataBatch. The transfer of the data initially occurs via the start actor-system but is distributed over the whole cluster afterward. According to the peer-to-peer approach, all actor-systems are involved in the data transfer.

As discussed in the last section, every worker has a list of sub-tasks with ids of data batches. We handle the processing of a sub-task as *black box* in this subsection to better focus on the communication and data transfer. A worker receives a list of sub-tasks and processes the list sequentially afterward. The worker checks if it has already stored the required data batches for the current task. If not, the worker requests the batches it currently needs for the task by asking the local data-bouncer with the ids from the task and waits asynchronously for the necessary data. If the worker has already cached the data or received the batches from the local data-bounce, the actor starts to process the subtask. Afterward, the worker begins to process the next task if it has unprocessed sub-tasks left.

To prevent a worker from having to wait for the required data before each task, we preload the needed batches to the actor-system beforehand. If a worker starts with a subtask, it tells the local data-bouncer which batches the actor needs for the next task. If the data-bouncer does not have stored the data, it requests the required batch from other data-bouncers while the worker is still working on a sub-task. With this strategy, we distribute the network load over the entire first phase of the algorithm, and none of the workers have to wait for data for further tasks. Each worker must only wait for the data transfer before processing their first task. We only benefit from this strategy if each worker does not only processes one task but several. Figure 7.10 in the evaluation shows that we can more than halve the runtime with a larger number of batches. Figure 5.3 visualizes the communication between worker and data-bouncer.

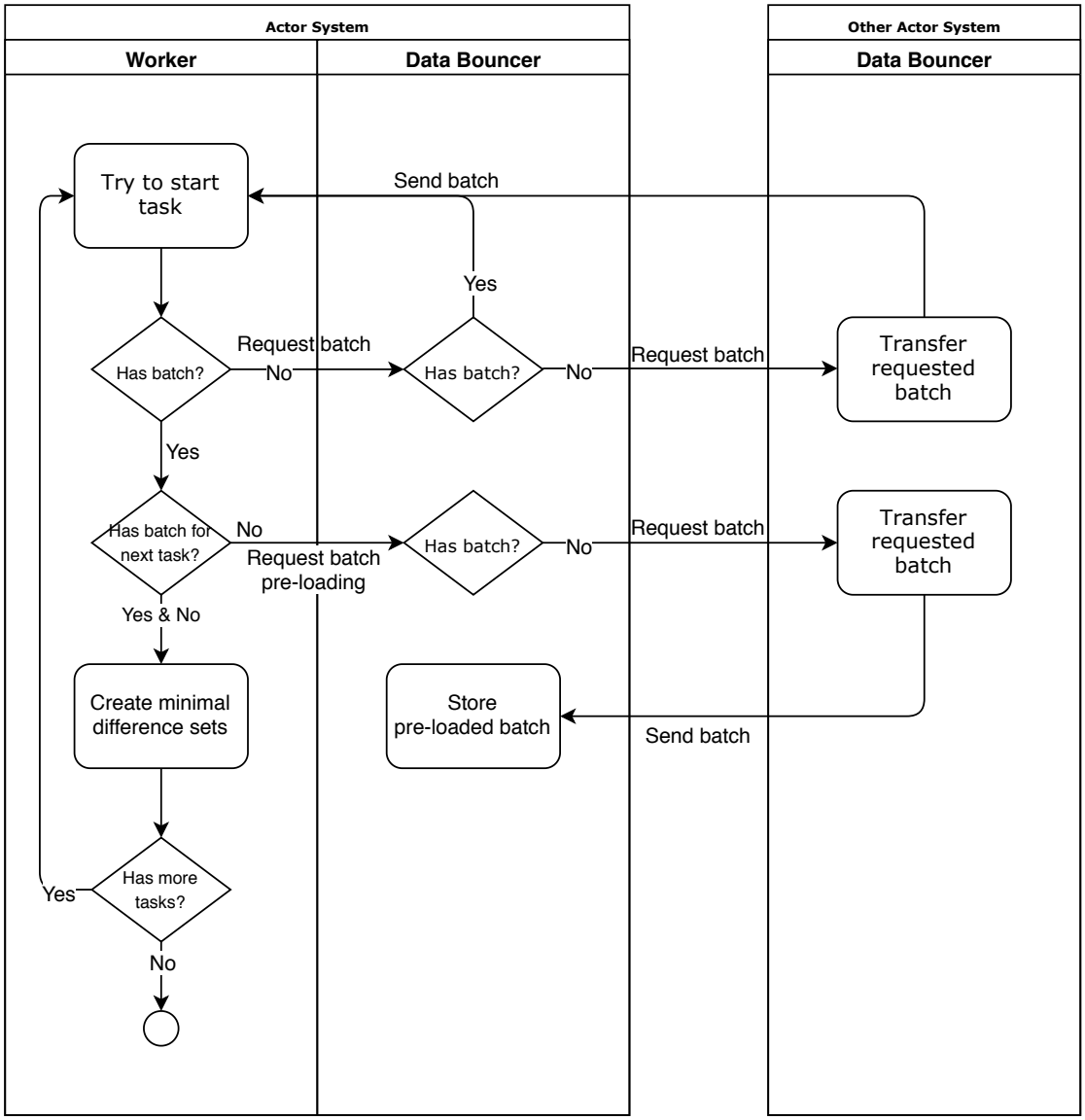


Figure 5.3: Worker — Data Bouncer Communication

5.3 Creating and Minimizing Difference Sets on a Single Actor

In this subsection, we describe how a worker creates minimal difference sets from two data batches after receiving tasks the data-bouncer of the start system. Besides, we explain how a worker merges two lists of minimal difference sets and removes all non-minimal sets. These calculations are all done locally on a worker, and no communication is involved.

5.3.1 Build Difference Sets

When creating the difference sets, we distinguish between two different cases: both data batches in a sub-task are different, and both batches are the same.

If both batches are different, the worker compares each row from the first batch with every row from the second batch. If there are n rows in the first batch and m rows in the second batch, the worker has to make $n * m$ comparisons. If both datasets are equal, it is sufficient to compare all rows within one batch, resulting in only $n*(n-1)/2$ comparisons, which is much faster to calculate.

When comparing two rows, the worker creates a bitmap that has a set bit at each position if the rows have two different values at that position. If this is not the case, the bit at that position is unset. We refer to the example in Chapter 3.2.1, Figure 3.1.

We have noticed that the resulting collection of difference sets often contain more duplicates than unique values. These occupy memory space and generate unnecessary comparisons in the following minimization step. The worker avoids keeping copies by saving all difference sets in a hash-based set.

Alternative Storage Strategies As further storage strategies, we tested an array bases list with and without removing duplicates. To remove all duplicates from a list, the worker sorts the list once and remove all duplicates in the subsequent pass minimizing. Another option to saving the difference sets is a *prefix tree*. A prefix tree (trie) is an index structure mainly used for storing character strings. The trie only saves every value exactly once and allows fast reading access. Every node of the prefix tree consists of one character of the string and going downwards to a leaf node results in a stored string. Instead of using strings, we use columns. We implemented a trie based on the implementation of Sedgewick et al. that does not index strings but column combinations. [25]. Figure 5.4 shows a symbolical example how we treat a column combination in a trie.

A	B	C	D	E
1	0	1	0	1
1	1	0	0	1
0	1	0	1	0

$\Longrightarrow$

ACE
ABE
BD

Figure 5.4: Representation of difference sets for usage in a trie

We compared the runtimes of both of the array-based approaches, our trie implementation, an external *Patricia* tree implementation [15] and the hash-based approach against each other. The hash-based strategy outperforms all the other approaches as evaluated in Section 7.4.1.

5.3.2 Minimize Difference Sets

After we calculated the list of all difference sets without duplicates, the worker has to minimize them. A naive strategy is to compare all difference sets and check if one of them is a subset of the other. However, we can take advantage of the fact that a difference set can only be a subset of another set if its cardinality (number of ones in the set) is smaller than the cardinality of the other. This particular case implies that if we divide all difference sets into buckets according to their cardinality, then no difference set can be a subset of another difference set in a bucket with greater cardinality. Additionally, we do not have to test any difference sets against each other in the same bucket. For two difference sets with unequal cardinality, we only have to check whether the set with smaller cardinality is a subset of the other and not the other way around. Thus, the difference sets in the bucket with the smallest cardinality are automatically minimal. Furthermore, we observed that the minimal difference sets with the smallest cardinality are subsets of many other sets.

Algorithm 5.2: Bucketing Approach to Minimize Difference Sets

Input: $uniqueDifferenceSets, numberOfColumns$
Output: List of all minimal difference sets

```
1  bucketList ← initialize array of lists of difference sets;
2  smallestCardinality ← numberOfColumns;
3  foreach set ∈ uniqueDifferenceSets do
4      bucketList[set.cardinality()].Add(set);
5      smallestCardinality ← min(smallestCardinality, set.cardinality());
6  minimalDifferenceSets ← [];
7  newMinimalSets ← [];
8  foreach set ∈ bucketList[smallestCardinality] do
9      minimalDifferenceSets.Add(set);
10 for i ← smallestCardinality + 1 to numberOfColumns do
11     foreach set ∈ bucketList[i] do
12         if isNoSuperset(set, minimalDifferenceSets) then
13             newMinimalSets.Add(set);
14         minimalDifferenceSets.AddAll(newMinimalSets);
15         newMinimalSets.Clear();
16 return minimalDifferenceSets
```

The Algorithm 5.2 reads as follows. First, the worker initializes the data structure and divides all difference sets into buckets according to their cardinality, while caching the smallest used cardinality in lines 1 to 5. Then, in lines 6 to 9, the actor adds all sets from the bucket with the smallest cardinality to a new list. This list contains all minimal difference sets. The worker performs the next step (lines 11 to 15) for all buckets in the

order from lowest to highest cardinality: For each difference set from the bucket, the worker tests whether the set is not a superset of one of the minimal sets in line 12. If this is the case, the actor adds the difference set to the list of minimal sets after testing the complete bucket in line 14. After checking all sets from all buckets, the worker successfully returns all minimal difference sets.

A categorization of n difference sets into buckets is faster than sorting the difference sets by their cardinality because the bucketing process is in $\mathcal{O}(n)$ and sorting a list in $\mathcal{O}(n \log n)$. Additionally, the worker does not compare difference sets with the same cardinality against each other with the bucketing strategy. This optimization is unfortunately not possible with a list without comparing the cardinality in each test.

It is possible that the worker has already processed one or more sub-tasks and has already stored difference sets that were minimal in those tasks. Unfortunately, difference sets that were minimal before may be no longer minimal after adding new difference sets. Instead of merging both lists of locally minimal difference sets, we use the already stored sets in the minimization process for the current difference sets. The worker does this by a list of buckets filled with all previously stored minimal difference instead of creating a list of empty buckets in line 1 of Algorithm 5.2. Since minimal difference sets often have a small cardinality, the worker can exclude non-minimal sets that would have been minimal without the previously stored difference sets early on and, thus, skip some comparisons.

5.4 Merging of Results

We discussed how HitUCC divides the creation of difference sets into sub-tasks and distributes them in the cluster. Every worker actor processes their tasks individually from each other and stored a list of minimal difference sets as a result. In this section, we describe how the workers communicate with each other in the peer-to-peer approach to merge all results and store them on a single worker. We divide the merge process into two phases — first, the merging of the results from a single actor-system, and second, the combination of the results from all actor-systems. We firstly want to combine all minimal difference sets locally, because there is no network overhead involved in the communication. Afterward, only one worker from each actor-system needs to communicate with other nodes. Thus, there is only little communication over the network, and the network overhead remains as small as possible. The actual merging process remains the same, with the only difference that we first merge on a local system before the synchronization takes place globally. Figure 5.5 visualizes the two-step merge process between two actor-systems in a sequence diagram. We only show the actual merging, but not the communication messages in this diagram to focus on the understanding of the two phases. We first explain the actual merging of two lists of minimal difference sets on a worker actor and refer to that functionality as “merge into” in the rest of this section.

Afterward, we describe the involved communication between all nodes in an actor-system to merge all results locally. At the end of this section, we demonstrate the second step of the merge process and how all actor-systems communicate with each other to achieve globally minimal difference sets.

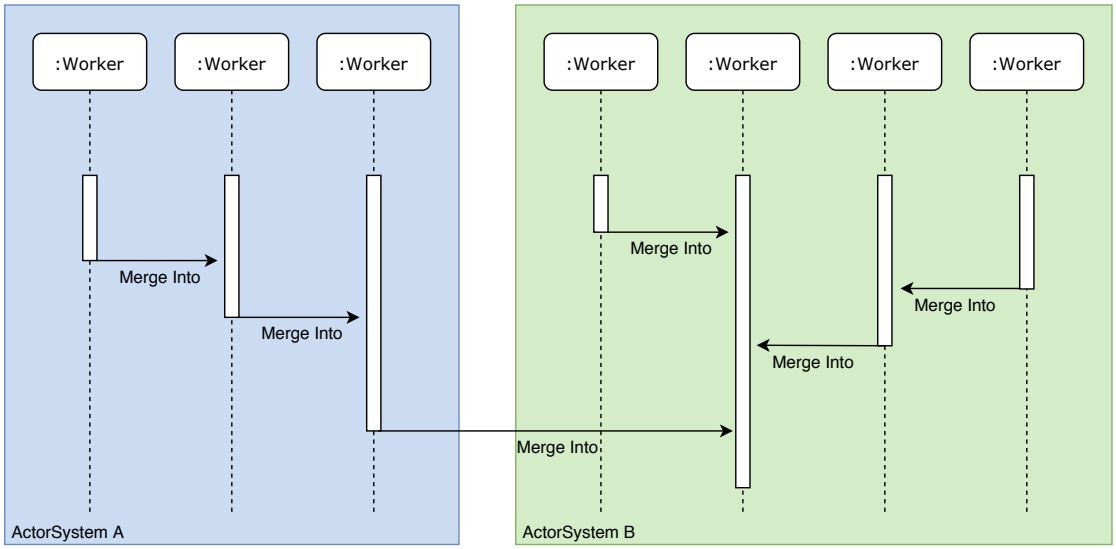


Figure 5.5: Two-Step Merge Process

5.4.1 Merge Minimal Difference Sets on a Single Actor

A worker has previously created a list of minimal difference sets and has to merge the list with difference sets from another worker from the same or a remote actor-system. Both lists of difference sets are sorted in ascending order by cardinality. We can now proceed in a similar way to MergeSort. Algorithm 5.3 describes the procedure.

The worker initializes the necessary variables in lines 1 to 3. Afterward, the worker repeats the next steps until it checked every difference set once (line 4). First, the worker tests the special cases that it has already checked all elements from one of the two lists (line 5 and 9). This check includes the event that one of the two collections is empty. If this is the case, then the algorithm only checks the difference sets from the other lists. The test if a difference set is minimal is the same as in Algorithm 5.2. The worker examines if the set is not a superset of any of the minimal sets and if it is not, the worker adds the difference set to the collection of minimal sets (lines 6 to 7, 10 to 11, 14 to 15 and 18 to 19). If both lists have sets that the worker did not test the worker examines which of the leftmost set of both collections have the smallest cardinality in line 13. If the leftmost set of the locally stored list has the smaller cardinality, the worker tests if

Algorithm 5.3: Merging of Minimal Difference Sets

Input: $lhsSets, rhsSets$

Output: List of all minimal difference sets

```
1  minimalDifferenceSets ← [];
2  lhsIndex ← 0;
3  rhsIndex ← 0;
4  while lhsIndex < lhsSets.length or rhsIndex < rhsSets.length do
5      if lhsIndex = lhsSets.length then
6          if isNoSuperset(rhsSets[rhsIndex], minimalDifferenceSets) then
7              minimalDifferenceSets.Add(rhsSets[rhsIndex]);
8          b ← b + 1;
9      else if rhsIndex = rhsSets.length then
10         if isNoSuperset(lhsSets[lhsIndex], minimalDifferenceSets) then
11             minimalDifferenceSets.Add(lhsSets[lhsIndex]);
12         a ← a + 1;
13     else if lhsSets[lhsIndex].cardinality() <= rhsSets[rhsIndex].cardinality()
        then
14         if isNoSuperset(lhsSets[lhsIndex], minimalDifferenceSets) then
15             minimalDifferenceSets.Add(lhsSets[lhsIndex]);
16         a ← a + 1;
17     else
18         if isNoSuperset(rhsSets[rhsIndex], minimalDifferenceSets) then
19             minimalDifferenceSets.Add(rhsSets[rhsIndex]);
20         b ← b + 1;
21 return minimalDifferenceSets
```

this set is minimal (lines 14 and 15). If the cardinality of the local set is the same or greater than the cardinality of the other set, the worker tests if the other difference set is minimal (lines 18 and 19). At the end of the algorithm, the worker returns a list of merged minimal difference sets.

5.4.2 Communiation in an Actor-System

The communication for the merging of the difference sets in one actor-system proceeds as follows: We sort all actors of that actor-system into a logical order ascending by name. A worker can send the following messages to other worker actors of the same system:

- A *ready-to-merge* message to signal that the worker finished discovering all difference sets and can merge them now.
- An *ask-for-merge* message to requests a merge operation with the message receiver.
- An *accept-merge* message to accept a merge request.
- A *decline-merge* message to decline a merge request.
- A *merge* message to transfer all stored difference sets to the receiving worker.
- A *finished* message to signal the receiver that the worker has transferred all data to another worker and does not participate in the merge process anymore.

Every worker only requests a merge with an actor with a lower position of the ordering because it is sufficient for two actors that only one of them asks for a merge and not both. That way, we keep the communication to a minimum while handling all communication with the peer-to-peer approach. With this strategy, the first worker from that order (worker_0) stores all minimal difference sets from the actor-system in the end.

If one worker has finished enumerating all minimal difference sets, it broadcasts a *ready-to-merge* to all workers with a higher position from the defined order and waits until the worker receives a *ready-to-merge* or a *request-merge* message from another worker. If a worker receives one or multiple *ready-to-merge* messages, it sends a *request-merge* message to a randomly chosen one of the transmitters. Because a worker can only receive a *ready-to-merge* from a worker with a lower position in the order, the worker can only send a *request-merge* message to a worker with a lower position in the previously defined order. The handling of a single merge request is similar to a TCP three-way handshake [27]. When a worker receives a *request-merge* message, it can respond with two options: It can either accept or decline the request. If the worker is already involved in a merge (requested or accepted a merge request), it replies with a *decline-merge* message. Otherwise, the

worker responds with a *accept-merge* message. Upon receiving a *decline-merge* message, the requested merge is aborted, and the worker requests a merge by a randomly chosen worker that is ready to merge. When a worker finally receives an *accept-merge* message, it responds transfers all stored difference sets with a *merge* message to the transmitter of that message. Additionally, the worker sends a *finished* message to all workers with a higher position from the defined order and worker_0. That way, the receiving workers know that this worker does not participate in the merge process anymore and will not request a merge from that particular worker. When a worker receives a *merge* message, it merges the received difference sets into the list of locally stored sets. If worker_0 received a *finished* message from all other workers in the actor-system, the merge process is complete.

The merge process runs partially in parallel with the minimization and the record comparisons because we already start the process when two workers finished minimizing their difference sets. Therefore we do not have to wait until all workers have found all results.

Figure 5.6 shows an example of communication in an actor-system with three workers. In the illustration, the worker_0 is the first to finish discovering all minimal difference sets and sends a *ready-to-merge* message to the other two workers. Afterward, the worker_1 completes finding all minimal difference sets and sends the message to worker_2. Worker_1 requests a merge from worker_0 that the other worker accepts. In the meantime, worker_2 also discovers all difference sets and requests a merge from worker_1, which declines the message because it is already involved in a merge process. Worker_1 then sends a merge message to worker_0 and broadcasts a *finish* message to both workers. Because worker_1 does not participate in the merge process anymore, worker_2 can only request a merge from worker_0, which the other worker accepts. After transferring a *merge* message to worker_0, worker_2 sends a *finished* message to worker_0, which received a *finished* message from all other workers, and thus, the merge process is complete.

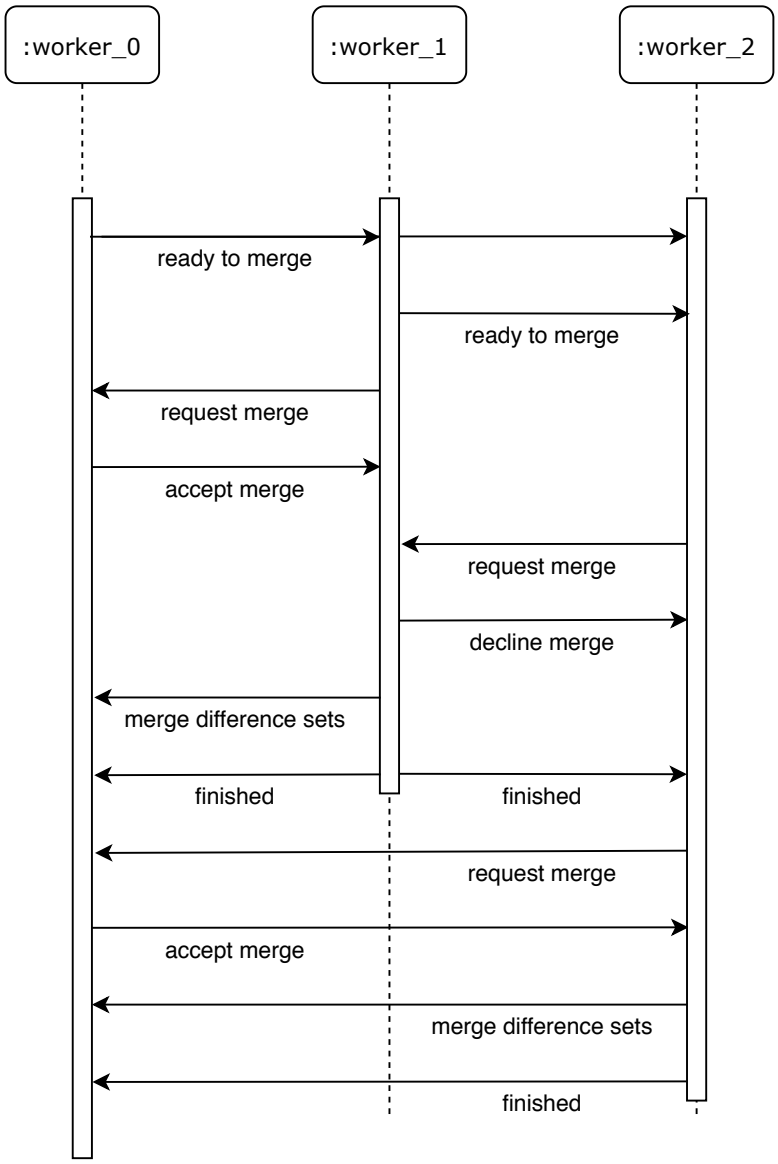


Figure 5.6: Merge Process Example

5.4.3 Communiation between Actor-Systems

We perform the same merge process between all actor-system as within an actor-system, except that only the worker_0 from each actor-system is involved. We define the order of the workers by using the network address of each respective actor-system, with the feature that the worker_0 from the start actor-system is the first actor in the order. If one of the workers has accumulated all minimal difference sets from their actor-system, the worker sends *ready-to-merge* to all other workers with a higher position from the newly defined order. From there, the merge process continues until all difference sets are stored on the worker_0 from the start system. This process overlaps with the local merge processes and potentially with the creation of all difference sets on a single worker as we start to merge all results as early as possible.

6 Discovering Minimal Unique Column Combinations

The worker_0 from the start actor-system has accumulated all minimal difference sets from each worker of the cluster. We must now implement and distribute the tree search introduced in Section 3. We first describe how to prepare the data. Then, we describe the tree search as if we would not distribute it, to focus on its functionality, and handle the extension oracle as a black box. Then, we explain how to distribute the task throughout the entire cluster and how we balance the load on the system during the tree search. Afterward, we describe how the actors merge all results to the node that started the algorithm. At the end of this section, we explain the extension oracle in detail.

6.1 Distributed Tree Search

In this subsection, we describe the distributed tree search and how all workers communicate with each other. We describe what actor types we are using and our preprocessing strategy. We cover the initial distribution, the rebalancing of the tasks, and the final merging of all found unique column combinations.

In the first phase of HitUCC, which is the creation of the minimal difference sets, we used two types of actors: the *data-bouncer*, who was responsible for the data management in the local actor-system and the whole cluster and the *worker* actor who processed the tasks and created all minimal difference sets. We only use a single actor type in the second algorithm phase. Since we already have several worker actors, we reuse them for the tree search and change their behavior via the strategy pattern. We do not need the data-bouncers, so we are terminating them. To realize this behavior, the worker that accumulated all minimal difference sets sends every other worker in the cluster the message that they should change their behavior for the current algorithm phase. Additionally, the worker transfers the information that the algorithm completed the first phase to all data-bouncers, who shut themselves down. The worker waits for an acknowledgment of the other workers before it starts the distributed tree search.

6.1.1 Non-Distributed Decision Tree Traversal

For the sake of simplicity, we explain everything in this subsection as if the tree search would only run locally on one actor. Instead of using a tree data structure, we use a *stack*, a list-like data structure that allows reading, adding, and removing the last element of the stack. The operations to add an element at the to the stack is called *push*, while the retrieving of the last element is called *pop*. A task in the stack represents a node in the decision tree and consists of two column combinations X and Y and the length of the

two column combinations. The decision oracle introduced in Section 3 introduced finds a solution to the Hitting Set problem, and thereby unique column combinations. It has the included column combination X, the excluded column combination Y, and all minimal difference sets as an input. The result of the *extendable* function of the oracle is "UCC" if X is a minimal UCC, "EXTENDABLE" if X without Y is extendable to a minimal unique column combination and "NOT_EXTENDABLE" if X is not extendable to a minimal UCC. Algorithm 6.1 shows the tree search and uses the *extendable* function of the oracle. We use this function as a black box and explain it in detail in Section 6.2.

Algorithm 6.1: Non-Distributed Tree Search

Input: $differenceSets$
Output: collection of UCCs

```
/* A task is a data structure with included columns X, excluded columns
   Y and the current length of both column combinations as values.  We
   store column combinations as bitmaps.                              */
1  taskStack ← initialize collection of tasks;
   /* add a task with two empty column combination and length equals 0  */
2  taskStack.Add(Task({}, {}, 0));
3  discoveredUCCs ← [];
4  while taskStack.length > 0 do
5      currentTask ← taskStack.pop();
6      result ← extendable(currentTask.X, currentTask.Y);
7      if result = UCC then
8          discoveredUCCs.Add(currentTask.X);
9      else if result = EXTENDABLE then
10         newX ← currentTask.X;
11         newX.setBit(currentTask.length);
12         taskStack.push(Task(newX, currentTask.Y, currentTask.length + 1));
13         newY ← currentTask.Y;
14         newY.setBit(currentTask.length);
15         taskStack.push(Task(currentTask.X, newY, currentTask.length + 1));
16 return discoveredUCCs
```

Algorithm 6.1 is non-distributed, so only one worker can run this version of the tree search. The worker actor starts the search with two empty column combinations X and Y and the length of 0 (line 2). The worker repeats the steps in lines 5 to 15 as long as the stack is not empty, and there are tasks to process (line 4). The worker reads from the oracle if the column combination X of the current task is extendable to a UCC while excluding the column combination Y or is a minimal UCC in lines 5 and 6. If X is a minimal unique column combination, the worker adds the combination X to the collection of discovered unique column combinations (lines 7 and 8). If the oracle returns

an "EXTENDABLE", the worker needs to add more columns to the combinations and adds two new tasks to the stack. For the first task, the worker adds the column next column (column at the index of the current length) to the included column combination X. The worker adds the next column to the excluded column combination Y for the second task. Additionally, the worker increases the length by 1 in both cases. If X is not extendable to a unique column combination, the worker does not add new tasks to the stack because those tasks can not be a UCC.

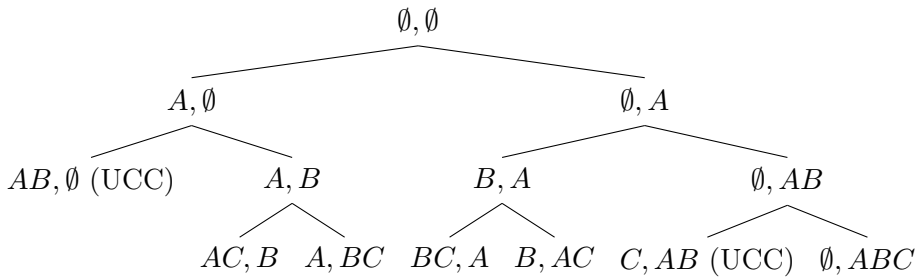
6.1.2 Preprocessing

Experiments have shown (Section 7, Figure 7.9) that the tree search is faster if we first test the column combinations that are most likely unique. If a column combination is a UCC, all supersets of this combination are unique as well and not minimal. Thus, we do not have to test them anymore. By avoiding these combinations, we reduce the search space and the tree search progresses faster. As presented in Algorithm 6.1, if the oracle discovers a UCC, we do not add new tasks to the stack.

A column combination is probably a UCC if most of the difference sets have a set bit at the position of these columns. A set bit in a difference set indicates that the values of two rows in the table at that position are not equal. The proposed tree search starts adding columns to the column combinations if the order of the columns in the difference sets. We can reorder the columns to manipulate the tree search. By sorting the columns descending by the number of set bits at that position, we ensure that the tree search tests columns (and column combinations) that are likely unique early. Figure 6.1 shows in an example that sorting the columns by the number of different values (i.e., the number of set bits) can significantly reduce the size of the tree, and therefore, decrease the runtime. On the other hand, if we sort the columns in ascending order by the number of set bits in all difference sets, we increase the size of the three, and thus, the runtime.

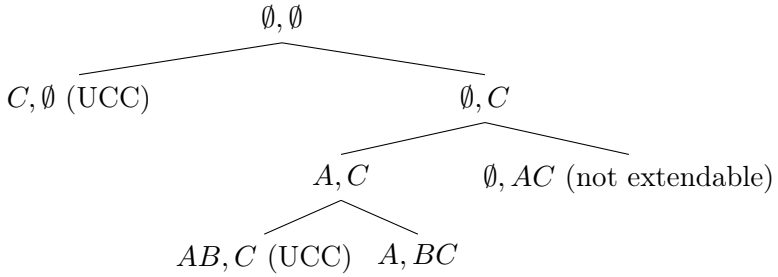
The worker that accumulated all difference sets in the first phase of HitUCC executes this optimization before starting and distributing the tree search by counting the number of set bits in every column and sorting the columns afterward. The actor remembers the old order of the columns to set the columns back in the correct order for the output of the algorithm. There is a similar performance impact if we sort all columns in descending order by the number of different values in a column in phase one before dividing the data into batches (Figure 7.9).

A	B	C
1	0	1
0	1	1



(a) Tree Search with unsorted Difference Sets

C	A	B
1	1	0
1	0	1



(b) Tree Search with sorted Difference Sets

Figure 6.1: Decision Tree with unsorted and sorted columns A, B and C.

6.1.3 Distribution and Workload Rebalancing

This sub-section describes how HitUCC distributes the tree search and balances the workload of each worker actor. We first describe the intuition before explaining specific details.

In a distributed setting, a worker does not process the complete tree, but handles 100 tasks before communicating with other workers. Every worker uses a *deque* (double-ended queue) to store the tasks. A deque is a list-like data structure that allows reading, adding, and removing the first *and* the last element of the structure. We replace the stack from Algorithm 6.1 with the deque because we need to access the first element when distributing tasks. If a workers empties its deque, this worker requests a task from another worker using work-pulling. If the other worker has tasks left in their deque, it transfers the first task from their deque to the requester. Since an actor otherwise treats the deque as a stack, the first element is the one that was the longest time in the deque. Therefore, this element is the work package with the smallest column combinations, and the oracle would likely decide that the column combination is extendable. Consequently, this element has the highest potential to open up a large sub-tree under itself. A worker sends this element so that the requesting worker is busy with processing the tree as long as possible to keep the communication overhead as small as possible. If a worker would send the last task from the deque, it will be processed much faster, and the worker

would have to send more messages over the network. Algorithm 6.2 shows how a worker processes 100 tasks. The difference to Algorithm 6.1 is that a worker stores a task deque and a collection of all discovered unique column combinations outside of the tree search function and can access the necessary data every time it starts processing the next 100 tasks.

Algorithm 6.2: Processing of 100 Tasks in the Tree Search

Input: $taskDeque, differenceSets, discoveredUCCs$

```
1  for i ← 0 to 100 do
2      if taskDeque.length = 0) then
3          break;
4      currentDeque ← taskDeque.pop();
5      result ← extendable(currentTask.X, currentTask.Y);
6      if result = UCC then
7          discoveredUCCs.Add(currentTask.X);
8      else if result = EXTENDABLE then
9          newX ← currentTask.X;
10         newX.setBit(currentTask.length);
11         taskDeque.push(Task(newX, currentTask.Y, currentTask.length + 1));
12         newY ← currentTask.Y;
13         newY.setBit(currentTask.length);
14         taskDeque.push(Task(currentTask.X, newY, currentTask.length + 1));
```

After presenting the intuition of the three search, we explain the communication between every worker in detail. Every worker has the following private state:

- A deque for storing tasks
- A randomized list of all other worker actors in the cluster
- Address to worker_0 from the start actor-system
- A *dirty-flag*
- A *shutdown-flag*

Every worker can communicate with every other worker in the cluster and requests tasks from the other actors in a *round-robin* way. Every worker shuffles the list of other actors to ensure that every worker requests tasks from different workers and that the workload is evenly distributed. We explain the dirty-flag and the shutdown-flag while explaining the worker's behavior when receiving messages. A worker can get six different messages in this phase:

- A *task-message* from another worker
- A *task-request-message* from another worker
- A *no-work-message* that an approached worker has an empty deque
- The *shutdown-preparation-message* that the tree search is almost complete
- A *finished-message* that an actor finished the tree search. Only the worker_0 from the start actor-system can receive this message
- A *shutdown-message* that the worker should stop itself

The worker_0 that sorted the columns creates and processes the first task consisting of two empty column combinations. The worker continues to process tasks from the deque until there are as many tasks in the deque as they are workers in the cluster. Afterward, the worker sends a *task-message* to every worker in the cluster with one of the created tasks for each actor. The other workers start to process tasks individually. A worker only processes up to 100 tasks. Then it sends itself an empty *task-message* to respond to messages from other actors. When the worker reads the empty *task-message*, it continues to process tasks from the deque. At the end of this phase, all minimal UCCs are stored in the worker_0 from the start system. We choose this actor for this role because every worker in the cluster knows the address of it, and we do not have to perform a new quorum-based round of consensus to agree on the result sink. The behavior of the worker when receiving messages is as follows:

Receiving a *task-message* A worker can receive two kinds of *task-messages* — a message with a task and an empty *task-message*. In the case that the task is not empty, the worker adds the task to the worker's deque and begins to process the next 100 tasks. Additionally, the worker sets the *dirty-flag*, which states that the worker got work from another worker. If the worker receives an empty work message from itself, it continues to process up to 100 tasks. If the worker receives an empty *task-message* from another actor, the current worker has requested a task from the other worker, which has only one task in the deque and can not transfer the only task. If this is the case, the local worker requests a task from the same actor again.

Receiving a *task-request-message* If the worker has set the *shutdown-flag*, we finished the tree search, and the worker replies with a *shutdown-preparation-message*. If no *shutdown-flag* is set, the worker processes the message normally. If the worker has more than one work package, the worker answers with a *task-message* from the first position of the deque. If the worker has only one work package left in the deque, the worker returns

an empty *task-message*. In the last case, if the deque is empty, the worker answers with a *no-work-message*.

Receiving a *no-work-message* When the actor receives a message that another worker has no more work left, the actor requested a task from that worker previously and, asks the next worker in his list for a task. When the actor has asked all other workers on the list, and the last worker has an empty deque, the worker checks its *dirty-flag*. If the flag is set, the worker has received tasks from another actor and cannot be sure that the algorithm processed all required tasks. So the actor removes the dirty flag and starts querying the list of workers from the beginning. If the flag is not set, the actor knows that no other actor can have work and sends itself a *shutdown-preparation-message* message.

Receiving a *shutdown-preparation-message* If an actor receives this message, the algorithm successfully completed the tree search. The actor sends the worker_0 from the start actor-system a *finished-message* and transfers all minimal UCCs it discovered. Afterward, the worker sets the *shutdown-flag* for itself.

Receiving a *finished-message* When the worker_0 from the start actor-system receives this message, at least one actor requested tasks from every other actor and was rejected. Now all actors prepare for the cluster shutdown. The worker is waiting for all other actors to transfer this message to it, and therefore all UCCs. As soon as everyone has done this, the worker sends a shutdown message to all workers in the cluster and stops himself.

Reading a *shutdown-message* If a worker gets a shutdown message, the actor stops itself, and we finished the algorithm.

The worker that accumulated all minimal unique column combinations resets all columns back in the right order and writes all found UCCs into a file. When all actors in an actor-system stopped themselves, we also shut the actor-system down, and thus, the process on the respective CPU node. We completed the algorithm, and all required resources on all physical systems are available again.

6.2 Extension Oracle

The extension oracle runs locally on each worker. Given a set of all columns V. The oracle uses the column combinations $X \subset V$ and $Y \subset V$ and all minimal difference sets H as input, $X \cap Y = \emptyset$. It decides whether X is extendable to a UCC excluding Y or whether X is already a unique column combination.

Algorithm 6.3: Decision Oracle

Input: included columns X, excluded columns Y, differenceSets H
Output: UCC, EXTENDABLE or NOT_EXTENDABLE

```
1  if X.cardinality() = 0 then
2      if isHittingSet(X, Y) then
3          return EXTENDABLE
4      return NOT_EXTENDABLE
5  t ← initialize collection of forbidden sets;
6  s ← initialize collection of colored lists;
7  foreach e ∈ H do
8      if cardinality of e ∩ X = 0 then
9          add e \ Y to T;
10     else if cardinality of e ∩ X = 1 then
11         add e \ Y to S_i;
12 if one of the lists of S is empty then
13     return NOT_EXTENDABLE
14 if T is empty then
15     return UCC
16 if ∀t ∈ T : t is not a subset of the union of S then
17     return EXTENDABLE
18 return NOT_EXTENDABLE
```

In the particular case that the cardinality of X is 0, we first check whether $V \setminus Y$ is a hitting set (lines 1 and 2). To accomplish this test, we check if each intersection of $V \setminus Y$ with each minimal difference set is not empty. If this is the case, X is extendable to a UCC (line 3). If one of the intersections is empty, $V \setminus Y$ is not a hitting set, and therefore there is no solution for this instance, and as a result, is X not extendable (lines 4).

If there is no such trivial solution we create the Multicoloured Independent Family instance (see Section 3.2.2) from X, Y and H. We create the empty set T (line 5), and the empty list of collections of sets S (line 6). T is the list of forbidden sets, and S is the collection of colored lists. We can access any list from S with S_i. Afterwards, we calculate the intersection of X with each difference set $e \in H$. If $e \cap X = \emptyset$, we add

$e \setminus Y$ to the collection of forbidden difference sets T (lines 8 and 9). If the cardinality of the intersection is equal to 1, we add $e \setminus Y$ to S_i, where i is equal to the index of set bit from the bitset $e \setminus Y$ (lines 10 and 11). After we have created the instance for the Multicoloured Independent Family problem, we can consider two trivial cases: If one of the lists in S is empty, there can be no solution for the cover problem and X is not extendable (lines 12 and 13). If the forbidden collection T is empty and no collection from S is empty, no set of S can completely cover a set from T, resulting in the solution that X is minimal (lines 14 and 15). If there is no trivial solution, we have to solve the Multicoloured Independent Family problem with a brute force approach.

To solve the cover problem, we need to check if the union of all sets from S completely covers any set of T. We determine this by testing if any difference set from T is a subset of the union. If a difference set of T is a subset of the union, there is no solution to the problem because we completely covered one of the forbidden sets. Thus, X is not extendable. If no difference set of T is entirely covered, there is a solution, and X is extendable (lines 16 and 17). We optimize the subset check by forming a union from the first bit of the difference sets of S first. We may determine that no set of T can be a subset of the union with only one bit. If we do not find a solution for the first bit, we add the next bit to the union, test again, and continue until we tested every set of T against the complete union of S. If we have not found a solution until now, X is not extendable (line 18).

7 Evaluation

In this chapter, we comprehensively evaluate the HitUCC algorithm for the UCC discovery. Our algorithm differs from other similar algorithms in two main features. Firstly, HitUCC is implemented with the actor model and can be executed in parallel on a cluster. Secondly, the algorithm does not traverse a lattice representation of the search space, as many other state-of-the-art algorithms do. HitUCC distributes the comparisons of every row in a dataset as a distributed preprocessing for the candidate checks. The distributed tree search uses the preprocessing to prune the search tree and to find minimal UCCs. The pruning strategy is suitable in a distributed setting because HitUCC does not need to communicate pruned candidates to other nodes. In this evaluation, we ask ourselves the following questions:

- How does the presented algorithm compare to state-of-the-art UCC discovery algorithms?
- How well can HitUCC be distributed across the cluster, and what is the behavior of the algorithm with data sets of different sizes and structures?
- How do we best distribute the required data across the cluster so that we reduce the overall runtime?
- Which of the presented strategies for finding the minimal difference sets and the tree search is the fastest?

To answer these questions, we first describe the hardware and software on which we conduct our experiments. Afterward, we give the results of our tests on one node and one actor to compare individual strategies without the network overhead in Section 7.4.1. Then we present the results of the strategy comparisons on the whole cluster in Section 7.4.2. Afterward, we show how well we can distribute the HitUCC algorithm on the entire cluster and what difference the number of used nodes makes in Section 7.2. At the end of the evaluation, we compare the proposed algorithm with other distributed and non-distributed state-of-the-art UCC discovery algorithms.

7.1 Experimental Setup

In this subsection, we describe our test setup. We cover the used cluster, all used data sets, and all used algorithms. The data sets may contain *null* fields. Our experiments use $null = null$ semantic since this is how null values are handled in related work [1]. However, the presented algorithm HitUCC can also consider $null \neq null$.

7.1.1 Cluster

We utilize a cluster consisting of eight Dell PowerEdge R430 rack server computers for our experiments. Each node is equipped with Intel Xeon E5-2360 v4 CPUs running ten physical cores at 2.2 GHz with Intel HyperThreading support, each of which can run two threads simultaneously (Figure 7.1). The nodes are interconnected with a Gigabit LAN connection and each of the eight cluster nodes is outfitted with 64 GB main memory and 1 TB disk storage. The operation system of the cluster nodes is the 64-Bit Ubuntu Server Linux distribution at version 18.04.2. The proposed HitUCC implementation uses the Akka Framework at version 2.5.26, and we are running all tests on the Java HotSpot 64-Bit Server VM for Java 1.8.0_211 on the cluster. We test the HyUCC algorithm with the metanome toolkit [21] version 1.2 and the Apache Spark-based implementation of the HyUCC with Apache Spark at version 2.44 on the cluster.

The cluster specification allows us to run a maximum of 20 Akka Actors on a single node at the same time. With eight nodes, the cluster provides us with 160 actors that we can run in parallel in theory. We use two kinds of actors in the first phase of our algorithm: the data-bouncer and worker actors. We reserve one thread per node for the data-bouncer and use 16 threads for the worker resulting in 16 used worker actors. Our actor model implementation of choice Akka needs free threads to send heartbeats between the actors to check that every actor is online. If we used 17 ore more threads, there would be no more free threads for the heartbeats, and the nodes would not reach each other via the heartbeats. This results in Akka sending more heartbeats, which blocks threads. In the worst case, the algorithm can abort because it thinks that the other nodes are no longer available.

Nodes	Cores per Socket	Threads per Core	Main Memory	Disk Storage
8	10	2	64 GB	1 TB

Figure 7.1: Cluster specifications

7.1.2 Data Sets

We use multiple data sets for our experiments. Figure 7.2 shows a numerical overview over the used data sets.

The **ncvoter10000** dataset is an extract of 10000 lines from the *ncvoter* dataset [13]. It contains information about voters from North Carolina. Among them are name, first name, and ZIP code of the persons. The **uniprot** dataset [11] is a public data set consisting of protein functions and sequences. We test this data set in different sizes: 100000 rows and 50 columns, 100000 rows and 80 columns, 100000 rows and

100 columns, and 539166 rows and 100 columns. The **STRUCT_SHEET_RANGE** dataset [5] from the Protein Data Bank consists of 3D shapes of proteins, nucleic acids, and complex assemblies. The **IMAGE** dataset has been crawled from the Wikipedia website.

Name	Size	Rows	Columns	Minimal UCCs
ncvoter10000	5 MB	10000	73	2281
uniprot100000_50c	139 MB	100000	50	20
uniprot100000_80c	167 MB	100000	80	23
uniprot100000_100c	185 MB	100000	100	31
uniprot539166_100c	993 MB	539166	100	269
STRUCT_SHEET_RANGE	150 MB	664128	32	167
CE4HI01	648 MB	1678782	65	25
IMAGE	100 MB	777676	12	3
artist	106 MB	1157142	19	3

Figure 7.2: List of the data sets we have used to evaluate HitUCC

7.1.3 Algorithms used for the Evaluation

We compare our implementation of HitUCC against the HyUCC [22] algorithm and a Apache Spark-based implementation of the HyUCC [24]. The non-spark HyUCC implementation uses all threads on one node to validate candidates.

Unfortunately, the Spark implementation of the HyUCC could not execute on data sets with *null* values. We had to edit the source code, to replace *null* values with empty strings when reading the data. Sadly the count of the returned minimal UCCs on the data sets *ncvoter10000*, *STRUCT_SHEET_RANGE* and *artist* was wrong. The reason for this could have been our change in the source code. We also tested another string as a replacement for *null* values and could not produce correct results on these three data sets. Nevertheless, we show the runtimes on all datasets.

7.2 Runtime Experiments

We present the results of our runtime experiments for all our datasets in Figure 7.3. We first notice a split in the results. Our algorithm outperforms the HyUCC algorithms by orders of magnitudes on all variants of the *uniprot* data set. Additionally, HitUCC was the fastest algorithm on the *ncvoter10000* data set. On the other hand, all other algorithms outperform our proposed approach on the other four data sets. What distinguishes

Data sets	HitUCC	HyUCC	Spark-HyUCC
ncvoter10000	**1.597 s**	3.219 s	5.819 s
uniprot100000_50c	**28.85 s**	63.380 s	51.931 s
uniprot100000_80c	**54.972 s**	4090.756 s	> 2 h
uniprot100000_100c	**68.418 s**	> 2 h	> 2 h
uniprot539166_100c	**1935.036 s**	> 2 h	> 2 h
STRUCT_SHEET_RANGE	352.474 s	29.499 s	**26.953 s**
CE4HI01	3546.064 s	**52.876 s**	77.560 s
IMAGE	328.831 s	**6.25 s**	10.141 s
artist	843.826 s	**9.522 s**	15.677

Figure 7.3: Runtime of different UCC discovery algorithm on multiple data sets

the data sets is the number of columns and the complexity of the unique column combinations. The *uniprot* datasets and the *ncvoter* dataset all have more than 50 columns and very complex UCCs, i.e., unique column combinations consisting of many columns. The other four datasets contain only low-level UCCs, i.e., unique column combinations consisting of very few columns, often only one or two columns. The CE4HI01 also has many columns, but only a few non-complex UCCs.

The HyUCC algorithm is using a bottom-up lattice traversal approach in the column-based strategy [22]. The more columns a data set has, the more candidates can be created in the lattice. At the same time, the more complex the minimum UCCs, the more candidates must be generated and tested in this strategy. This approach indicates, that the runtime of the HyUCC algorithm increases exponentially with the number of columns and the complexity of the UCCs. On the other hand, the algorithm is heavily based on pruning techniques, and can massively reduce the search space when they find unique column combinations.

In contrast to this is our proposed approach, which must always first compare every line in the data set with every other line in it. During this time, we cannot prune and reduce the search space. The runtime of the comparisons increases exponentially with the number of rows in the data set. The comparisons also scale with the number of columns, but not as much as with the number of rows. The tree search scales with the number of columns but is fast compared to the first phase of the algorithm and is not very important for the runtime.

The reason for the differences in runtime is, therefore, the nature of the data records. HitUCC has about the same runtime for the *STRUCT_SHEET_RANGE* and the *IMAGE* data set because both data sets have similar numbers of rows and columns. The runtime is much higher for the *artist* and the *CE4HI01* data set, because the number of rows is much higher for these two. The runtime differences between the *uniprot* data

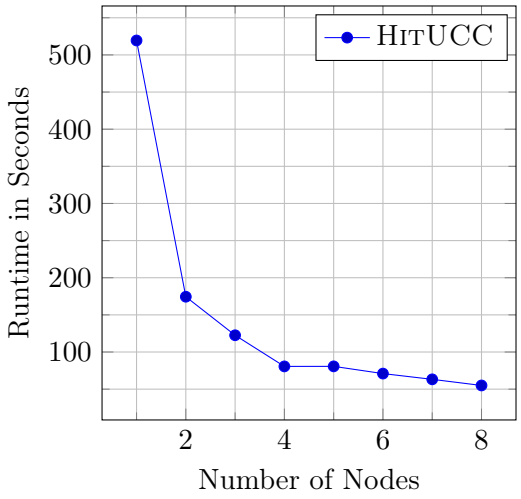
sets with 100000 rows is not very big, but it increases with the number of columns. The runtime of the largest *uniprot* data set is much higher than the runtime of the other *uniprot* data sets, but not as long as the runtime on the *CE4HI01* data set, which has the most rows of all tested data sets.

7.3 Scale-Out Experiments

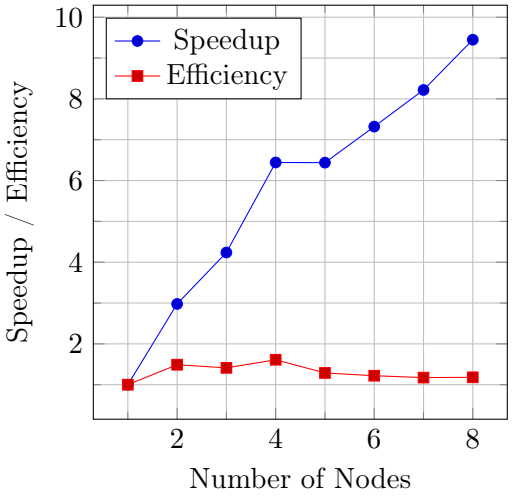
In this sub-section, we provide three scale-out experiments. We test HitUCC with the *uniprot100000_80c*, *uniprot100000_100c*, and the *ncvoter10000* dataset on the cluster with node counts from 1 to 8 (Figure 7.4 and 7.5). In each case, we specify the speedup and the efficiency of the algorithm to understand scale-out behavior better. The speedup is calculated from the runtime of the algorithm with one node divided by the runtime with the respective node count. This chart makes it clear how much faster the algorithm is with multiple nodes than with one node. The efficiency is the speedup divided by the number of nodes used. This chart describes how much the distribution yields.

We notice that HitUCC scales very well on the two large data sets. With each added node, the algorithm gets faster. There is a big jump between one and two nodes. We have verified that it is not because one node has less available memory and therefore has difficulties with the amount of data. We assume that the runtime reduction on one node comes from the used HashSet because we put more data into when we are using only one node, and there are many more collisions. The curve is jagged because we use a different number of data batches for each node as the number of worker actors changes, and we want to allow the highest possible parallelism. The figure shows that the speedup and efficiency decrease with the number of nodes, but that the speedup is almost linear to the number of nodes. Also note that HitUCC outperforms the HyUCC algorithm with a single node on both data sets.

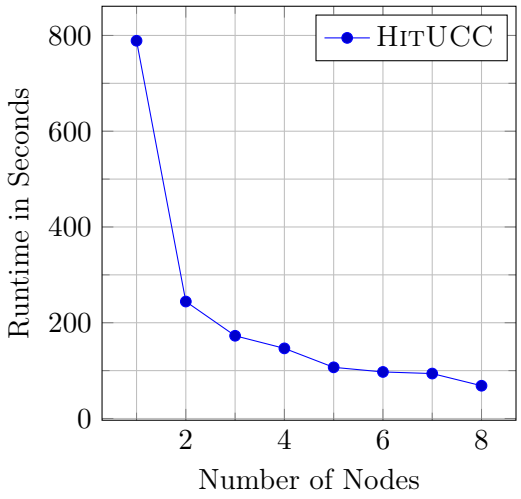
On the other hand, we notice that the parallelism does not always decrease the runtime for the small *ncvoter* data set (Figure 7.5). The different tasks each worker has to do are so small that the network overhead gets more significant than the comparisons. We note that the speedup is at its peak at four nodes and that the efficiency drops continuously.



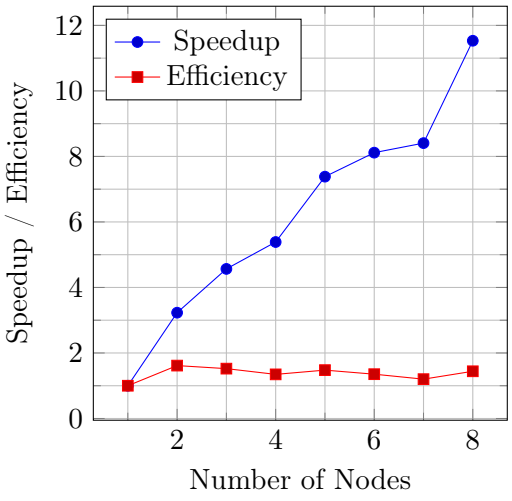
(a) Algorithm runtime on the *uniprot100000_80c* data set with different node counts.



(b) Speedup and Efficiency of HitUCC with different node counts on the *uniprot100000_80c* data set.

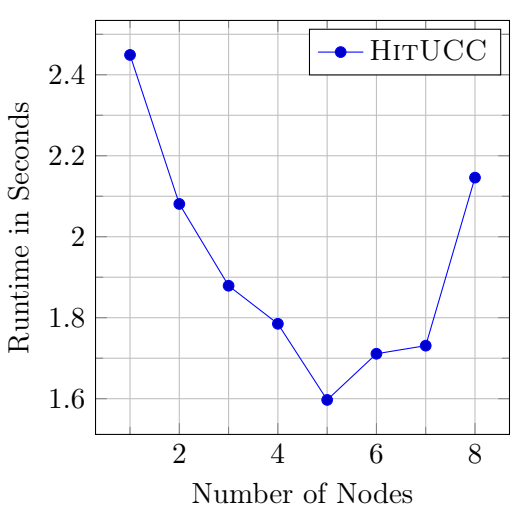


(c) Algorithm runtime on the *uniprot100000_100c* data set with different node counts.

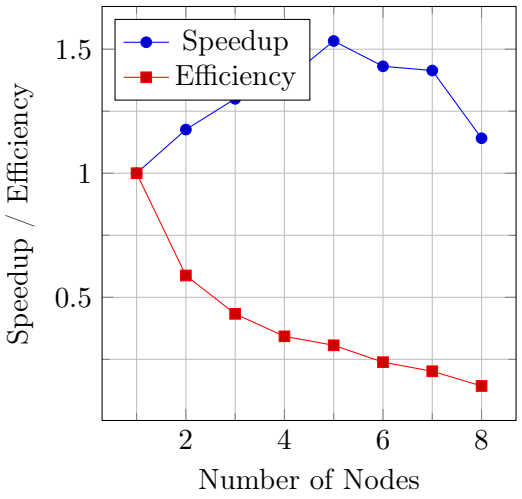


(d) Speedup and Efficiency of HitUCC with different node counts on the *uniprot100000_100c* data set.

Figure 7.4: Scale-out behavior on *uniprot* data sets.



(a) Algorithm runtime on the *ncvoter10000* data set with different node counts



(b) Speedup and Efficiency of HitUCC with different node counts on the *ncvoter10000* data set

Figure 7.5: Scale-out behavior on *ncvoter10000* data sets.

7.4 Comparing Strategies

We compare different strategies of our algorithm against each other in this subsection. First, we test methods that impact the performance of a single actor locally on a node with one worker. Afterward we test different strategies on the whole cluster. We do not include other algorithms in this subsection.

7.4.1 Single Node Strategy Comparison

We use the small *ncvoter10000* dataset because we only work with one worker on one node. Thus in this subsection, one worker has a batch of 10000 rows and compares each row to each of the other rows in the batch and then minimizes the results. First, we compare strategies for creating all difference sets without minimizing them. We test which data structure is most efficient for storing all difference sets, while possibly removing all duplicates at the same time. The minimize step is faster if the enumerated difference sets do not contain any copies. We used five data structures to enumerate all difference sets:

(a) A naive approach with a list in which we store all difference sets and do not remove duplicates.

(b) A list in which we insert all difference sets. After enumerating, we sort the list and remove all duplicates.

(c) The Apache implementation [15] of a trie based on strings.

(d) A prefix tree implemented by us which indexes column combinations.

(e) The enumeration using a Hashmap.

The tables in Figure 7.6 each consist of two columns: The *build difference sets* column and the *minimize difference sets* column. The first column contains the duration of the creation of all difference sets and the storing into the used data structure in seconds. The second column contains the reading of all formed difference sets from the data structure and the minimizing of those sets. The runtime of the minimize step is different for each approach, even though the input is the same (except for strategy (b)). The reason is that the runtime of the retrieval of all difference sets from each data structure is different — the random access on a list is faster than on a hash set or a prefix tree. The worst performing method is the list-based strategy with sorting the difference sets and removing all duplicates. By comparison, the other list-based variant is much faster. The reason for this is that the sorting of all difference sets takes so much time because we need to

sort not only by the cardinality of the difference sets but by the index of each bit. On the other hand, the minimize step on the slowest strategy is faster than in any other strategy. The second slowest strategy is the external Patricia Trie implementation of Apache [15] because the implementation is based on strings and we need to encode a column combination to a string each time we insert it to the trie and decode it when reading from the trie. Our trie implementation operates directly on column combinations, does not need to encode or decode them, and stores all formed difference sets much faster into the data structure. However, the Apache implementation can read all unique difference sets faster. The second fastest strategy is the naive implementation with a list without removing any duplicates or sorting of the list, and the approach with the smallest runtime is the hash-based variant.

The results of this experiment are that the removal of duplicates is not that important for the minimizing step. The bucketing process requires very few checks compared to creating the difference set and is only a bit slower if we do not remove all duplicates. Secondly, we proved that the hash-based strategy is the fasted of our tested ones. We use this approach for the next experiments in this section.

Operation	Hash Set	Naive	List + Remove Duplicates	Patricia tree	Trie
Create Difference Sets	**10.530 s**	15.370 s	80.753 s	60.111 s	23.504 s
Minimize Difference Sets	0.087 s	0.033 s	**0.004 s**	0.11 s	0.233 s

Figure 7.6: Single Node Experiments with various difference set enumeration strategies

We have developed three different strategies for minimizing difference sets. First, we use the bucketing approach presented in Section 5.3.2. The second strategy is similar to the bucketing approach. Instead of organizing every difference sets into buckets, we sort the sets ascending by cardinality. The third method is a naive variant where we compare all found difference sets in pairs to find the minimal difference sets. Figure 7.7 shows that the bucketing process is the fastest strategy. The reason for this is that we can organize all difference-set into buckets in $\mathcal{O}(n)$, while the sorting by cardinality is in $\mathcal{O}(n \log n)$. Additionally, we do not compare difference sets with the same cardinality against each other in the bucketing strategy because we do not compare sets from the same bucket. This optimization is unfortunately not possible with a list. The naive method is by far the slowest method of all because we have to make much more comparisons, which increases the runtime.

Operation	Bucketing Method	Sorting Method	Naive
Minimize Difference Sets	**0.087 s**	0.141 s	1882.64 s

Figure 7.7: Single Node Experiments with various difference set minimization strategies

In the next experiment, we evaluate the impact of the number of data batches in phase one on the runtime. The number of data batches on a single worker has no positive effect

on the runtime (Figure 7.8a). If we use 100 data batches, the runtime gets even worse than with only one batch. The advantage of multiple data batches (and thus numerous work packages) is that the data batches that we have to transfer over the network to other nodes become smaller and to improve parallelism if we have more work packages than workers. Since we do not need networking with a single worker, this optimization does nothing positive and increases the runtime because of the unneeded overhead. We do not focus on the best possible number of data batches; we want to show that there is a difference between different data batch counts. See Figure 7.10 for more in-depth tests on the data batches.

For comparison, the number of batches changes the runtime slightly when we use more actors in a system as the parallelism becomes higher (Figure 7.8b). The runtime with one batch is the same in both experiments because we cannot utilize more than one actor with only a single data batch. The runtime of the test increases if we use too many data batches.

1 Data Batch	10 Data Batches	100 Data Batches
10.883 s	**10.383 s**	15.853 s

(a) Experiment with a single actor

1 Data Batch	10 Data Batches	100 Data Batches
10.817 s	**1.579 s**	2.541 s

(b) Experiment with 16 worker actors

Figure 7.8: Single Node Experiments with various data batch counts on the *ncvoter10000* data set

The next experiment evaluates the approaches that sort the columns to improve the performance of the tree search. We can sort the columns in two different steps of the algorithm: sorting the columns descending or ascending by the number of distinct values in a column before distributing the data in the first step and sorting the columns descending or ascending by the number of set bits in the difference sets in the second phase. Figure 7.9 proves that sorting in an ascending order increases the runtime by orders of magnitude. There is no significant difference between sorting the columns descending in phase 1 or phase 2. Sorting the columns descending in the second phase is slightly faster, and therefore we use this approach for all next experiments.

Sort desc (Phase 1)	Sort asc (Phase 1)	Sort desc (Phase 2)	Sort asc (Phase 2)
0.492 s	2.587 s	**0.433 s**	2.564 s

Figure 7.9: Sort columns to manipulate tree search runtime

We have tested all possible configurations of the algorithm on one node and one worker. The result is that we enumerate all difference sets with a hashmap, minimize all difference sets with the bucketing approach and sort the columns of the table in descending order by the number of set bits in the difference sets before the tree search.

7.4.2 Multi Node Strategy Comparison

In this subsection, we run all experiments on the whole cluster and find the best algorithm configuration for 16 workers on eight nodes each. Thereby we test the performance impact of the number of data batches used in phase one and if a greedy redistribution of work packages is worthwhile. The second experiment is the number of work packages that an actor locally processes in the tree search before reading and processing messages from other actors.

We test how the runtime of creating difference sets changes when we change the number of data batches on the *uniprot100000_ 50c* data set in Figure 7.10. The number of tested data batches are: 20, 50, 100, 200, 255, and 300. With 16 workers on eight nodes, we have $16 * 8 = 128$ worker actors in the cluster. With 16 data batches, we have the 16th triangular number $(16^2 + 16)/2 = 136$ as the number of work packages that the cluster needs to process. 136 is the smallest number of work packages that allow us to have at least one work package per worker. The $255th$ triangular number is a special triangular number because it creates $(255^2 + 255)/2 = 32640$ work packages and 32640 is a multiple of the number of workers in the cluster, such that each worker actor processes exactly $32640/128 = 255$ work packages and we achieved the highest possible parallelism. Figure 7.10a shows the experiment with and Figure 7.10b without greedy task redistribution. We note that a greedy redistribution of the tasks to the individual actor-systems decreases the runtime on every test. The reason is that we reduce the amount of data packages that we need to send to the different nodes and thus reduce the time a worker needs to wait before the worker can begin working on a task. We also note that the runtime decreases until we reach the 255th triangle number and then increases again. With 255 data batches, each worker has to process the same amount of tasks, and no actor has to wait long for the other worker, which is why this number causes the best performance.

16 Batches	50 Batches	100 Batches	200 Batches	255 Batches	300 Batches
39.297 s	19.781 s	17.221 s	15.642 s	**14.936 s**	15.192 s

(a) Without greedy task redistribution

16 Batches	50 Batches	100 Batches	200 Batches	255 Batches	300 Batches
35.319 s	19.287 s	16.963 s	15.211 s	**14.779 s**	14.919 s

(b) With greedy task redistribution

Figure 7.10: Multi Node Experiments with various data batch counts on the *uniprot100000_ 50c* data set. We show the runtime of the creation of all minimal difference sets.

The last experiment in the subsection evaluates the number of local tree search tests on

a worker before he starts communicating again. We use the *ncvoter10000* data set for this experiment since it contains most UCCs, and the tree search takes the longest time on this data set. We use 10, 100 and 1000 as possible numbers (Figure 7.11).

Tree depth 10	Tree depth 100	Tree depth 1000
0.871 s	**0.76 s**	1.397 s

Figure 7.11: Test different local tree depth before communicating again

To understand the runtimes of HitUCC better, we show the runtimes on all data sets divided by the runtime of encoding the data and calculating a greedy distribution, the creation of all difference sets, and the runtime of the tree search (Figure 7.12). Reading and encoding the data takes significantly longer than the runtime of the tree search. On the other hand the creation of the difference sets can take orders of magnitudes longer than the tree search, the encoding, and the calculation of the greedy distribution.

Data sets	Encode + Greedy	Create Difference Sets	Tree Search
ncvoter10000	1.902 s	0.596 s	0.482 s
uniprot100000_50c	11.687 s	14.714 s	0.269 s
uniprot100000_80c	13.838 s	38.314 s	0.209 s
uniprot100000_100c	15.099 s	51.761 s	0.176 s
uniprot100000_100c	60.052 s	1846.72 s	0.141 s
STRUCT_SHEET_RANGE	18.582 s	329.3 s	0.245 s
CE4HI01	47.845 s	3460.403 s	0.263 s
IMAGE	18.751 s	328.831 s	0.204 s
artist	18.683 s	820.742 s	0.225 s

Figure 7.12: Runtime of HitUCC divided in encoding plus greedy distribution, creating all minimal difference sets and the tree search runtimes

8 Conclusion

In this master's thesis, we investigated the use of the actor model to find unique column combinations. We investigated the possibility of parallelizing the discovery and efficiently using all available resources. We have studied the subdivision of the algorithm into small subtasks that we can perform in parallel. Our focus was mainly on the balance of the tasks of all nodes and the minimization of data transfer over the network. We tested the peer-to-peer approach for communication and work schedules.

At the end of this thesis, we want to summarize our results and our contributions. We will focus on the proposed HitUCC algorithm and show our perspective on how the algorithm can be improved and changed.

8.1 Summary

We implemented and evaluated the distributed algorithm HitUCC. We demonstrated that HitUCC scales well on a cluster when discovering unique column combinations on big data sets and that we achieve an excellent speedup and a great efficiency. On the other hand, we discovered that HitUCC does not scale well on small data sets and that the efficiency over multiple nodes decreases faster than on a bigger data set. Furthermore, we showed that the proposed algorithm outperforms a distributed and non-distributed HyUCC implementation on data sets with many columns and complex UCCs. Additionally, we discovered that HitUCC is slower than the other tested algorithms on data sets with fewer columns.
We implemented and tested an approach to compare every row in a data set in a distributed setting, which is a task used in other applications, such as data deduplication. We tested different strategies to enumerate and minimize the resulting difference sets. We investigated how to balance the workload on all involved nodes. As part of the balancing and the distribution, we need to send the necessary data to each node in the cluster. We investigated how to keep transfer as few as possible data over the network and how to transfer the data just before a node needs this data. We implemented a peer-to-peer communication strategy to merge the results of all nodes. We discussed how to distribute a tree search and how to collect all results. Additionally, we introduced a peer-to-peer strategy that uses work-pulling to redistribute tasks if the tree search is unbalanced.

We discovered that the preprocessing of our approach is the slowest step. Sometimes, it is a lot faster to test small column combinations to find all minimal unique column combinations, instead of comparing each row with every other in a data set.

8.2 Future Work

This section describes approaches to extend HitUCC and what a productive execution would require from the algorithm.

Incremental Algorithm

One possibility to extend the algorithm is to store intermediate results to avoid having to run through the complete algorithm again when changing the underlying data set. After the first phase of the algorithm, it is possible to save the minimal difference sets in a file. If a new row is added to the data set, we only need to compare the new row with every other row in the data set. Afterward, we merge the resulting difference sets with the previously stored difference sets. Unfortunately, we need to execute the search, but the runtime is not as high as the runtime of finding all minimal difference sets.

Fault Tolerance

Many errors can occur in a distributed setting. Messages between the nodes in the cluster may arrive malformed or may not arrive at all. The network can fail briefly or even collapse altogether. It can also happen that a node produces an error and stops. Before we can use a distributed algorithm productively, we must ensure that this algorithm is network fault-tolerant.

In the first phase of HitUCC, each actor-system works autonomously before a global merge occurs. To make this phase fault-tolerant, we can send the information in which nodes compare which data batches to all other nodes. So if a node fails or is no longer available, the list of all missing tasks can be distributed over all remaining nodes. As long as the start node is still available, it is ensured that the other nodes can get the necessary data. To cover the case that the start node fails, the node can distribute the data batches in the cluster such that each batch is at least stored on another node.

The tree search from the second phase is very fast. Instead of making the tree search fault-tolerant, it is possible to redo the entire phase. To do this, we can store all minimal difference sets in a file on each node. If one or more nodes fail, we read all difference sets from the file, and we abort the tree search and restart it on the remaining nodes.

Better Data Transfer with Compression

An option to reduce the time of the data transfer between nodes would be to compress our data further. One option is to use *bit-compression* [29] to reduce the size of a dictionary encoded column. We store every column combination, and every difference set as a bitmap. An option to reduce the size of a bitmap is the *roaring bitmap* approach [9] that is implemented by Apache Spark.

9 References

[1] Abedjan, Ziawasch ; Golab, Lukasz ; Naumann, Felix ; Papenbrock, Thorsten: Data profiling. In: *Synthesis Lectures on Data Management* 10 (2018), Nr. 4, S. 1–154

[2] Abedjan, Ziawasch ; Naumann, Felix: Advancing the discovery of unique column combinations. In: *Proceedings of the 20th ACM international conference on Information and knowledge management* ACM, 2011, S. 1565–1570

[3] Abedjan, Ziawasch ; Quiané-Ruiz, Jorge-Arnulfo ; Naumann, Felix: Detecting unique column combinations on dynamic data. In: *2014 IEEE 30th International Conference on Data Engineering* IEEE, 2014, S. 1036–1047

[4] Agrawal, Rakesh ; Srikant, Ramakrishnan u. a.: Fast algorithms for mining association rules. In: *Proc. 20th int. conf. very large data bases, VLDB* Bd. 1215, 1994, S. 487–499

[5] Bank, Protein D.: *STRUCT_ SHEET_ RANGE Dataset*. `https://www.rcsb.org/`

[6] Birkhoff, Garrett: *Lattice theory*. American Mathematical Soc., 1940

[7] Bläsius, Thomas ; Friedrich, Tobias ; Lischeid, Julius ; Meeks, Kitty ; Schirneck, Martin: Efficiently Enumerating Hitting Sets of Hypergraphs Arising in Data Profiling. In: *2019 Proceedings of the Twenty-First Workshop on Algorithm Engineering and Experiments (ALENEX)* SIAM, 2019

[8] Bläsius, Thomas ; Friedrich, Tobias ; Schirneck, Martin: The parameterized complexity of dependency detection in relational databases. In: *11th International Symposium on Parameterized and Exact Computation (IPEC 2016)* Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017

[9] Chambi, Samy ; Lemire, Daniel ; Kaser, Owen ; Godin, Robert: Better bitmap performance with roaring bitmaps. In: *Software: practice and experience* 46 (2016), Nr. 5, S. 709–719

[10] Chu, Xu ; Ilyas, Ihab F. ; Koutris, Paraschos: Distributed data deduplication. In: *Proceedings of the VLDB Endowment* 9 (2016), Nr. 11, S. 864–875

[11] Consortium, UniProt: *Uniprot Dataset*. `https://www.uniprot.org`

[12] Docker, Inc.: *Docker*. `https://www.docker.com/`

[13] Elections, North Carolina State B.: *NCVoter Dataset*. https://www.ncsbe.gov/ncsbe/data-statistics

[14] Foundation, Apache S.: *Apache Flink*. https://flink.apache.org/

[15] Foundation, Apache S.: *Apache Flink*. https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/trie/PatriciaTrie.html

[16] Foundation, Apache S.: *Apache Spark*. https://spark.apache.org/

[17] Heise, Arvid ; Quiané-Ruiz, Jorge-Arnulfo ; Abedjan, Ziawasch ; Jentzsch, Anja ; Naumann, Felix: Scalable discovery of unique column combinations. In: *Proceedings of the VLDB Endowment* 7 (2013), Nr. 4, S. 301–312

[18] Huhtala, Yka ; Kärkkäinen, Juha ; Porkka, Pasi ; Toivonen, Hannu: TANE: An efficient algorithm for discovering functional and approximate dependencies. In: *The computer journal* 42 (1999), Nr. 2, S. 100–111

[19] Inc., Lightbend: *akka*. https://akka.io/

[20] Kleppmann, Martin: *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems*. " O'Reilly Media, Inc.", 2017

[21] Papenbrock, Thorsten ; Bergmann, Tanja ; Finke, Moritz ; Zwiener, Jakob ; Naumann, Felix: Data profiling with metanome. In: *Proceedings of the VLDB Endowment* 8 (2015), Nr. 12, S. 1860–1863

[22] Papenbrock, Thorsten ; Naumann, Felix: A hybrid approach for efficient unique column combination discovery. In: *Datenbanksysteme für Business, Technologie und Web (BTW 2017)* (2017)

[23] Plattner, Hasso u. a.: *A course in in-memory data management*. Springer, 2013

[24] Saxena, Hemant ; Golab, Lukasz ; Ilyas, Ihab F.: Distributed implementations of dependency discovery algorithms. In: *Proceedings of the VLDB Endowment* 12 (2019), Nr. 11, S. 1624–1636

[25] Sedgewick, Robert ; Wayne, Kevin: Algorithms and data structures. In: *Princeton University, COS* 226 (2007)

[26] Sismanis, Yannis ; Brown, Paul ; Haas, Peter J. ; Reinwald, Berthold: GORDIAN: efficient and scalable discovery of composite keys. In: *Proceedings of the*

32nd international conference on Very large data bases VLDB Endowment, 2006, S. 691–702

[27] Stevens, W R.: *TCP/IP illustrated vol. I: the protocols*. Pearson Education India, 1994

[28] Vernon, Vaughn: *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. Addison-Wesley Professional, 2015

[29] Wang, Jianguo ; Lin, Chunbin ; He, Ruining ; Chae, Moojin ; Papakonstantinou, Yannis ; Swanson, Steven: MILC: inverted list compression in memory. In: *Proceedings of the VLDB Endowment* 10 (2017), Nr. 8, S. 853–864

Declaration of authorship

With these words, I assure, that this master's thesis was written independently. No other resources other than those specified were used, and quotations were explicitly marked.

Berlin, January 15th, 2020