Divide & Conquer-based Inclusion Dependency Discovery

Thorsten Papenbrock♦ Sebastian Kruse♦
Jorge-Arnulfo Quiané-Ruiz⋆ Felix Naumann♦

♦Hasso Plattner Institute (HPI)
Potsdam, Germany

⋆Qatar Computing Research Institute (QCRI)
Doha, Qatar

{thorsten.papenbrock, sebastian.kruse, felix.naumann}@hpi.de
jquianeruiz@qf.org.qa

ABSTRACT

The discovery of all inclusion dependencies (INDs) in a dataset is an important part of any data profiling effort. Apart from the detection of foreign key relationships, INDs can help to perform data integration, query optimization, integrity checking, or schema (re-)design. However, the detection of INDs gets harder as datasets become larger in terms of number of tuples as well as attributes.

To this end, we propose Binder, an IND detection system that is capable of detecting both unary and n-ary INDs. It is based on a divide & conquer approach, which allows to handle very large datasets – an important property on the face of the ever increasing size of today's data. In contrast to most related works, we do not rely on existing database functionality nor assume that inspected datasets fit into main memory. This renders Binder an efficient and scalable competitor. Our exhaustive experimental evaluation shows the high superiority of Binder over the state-of-the-art in both unary (Spider) and n-ary (Mind) IND discovery. Binder is up to 26x faster than Spider and more than 2500x faster than Mind.

1. INTRODUCTION

Current applications produce large amounts of data at fast rates. Understanding such datasets before querying them is crucial for both high quality results and query performance. In this regard, *data profiling* aims at discovering and understanding relevant basic properties of datasets, such as column statistics and dependencies between attributes [15]. In practice, data profiling is needed whenever the metadata of a dataset is (or became) unknown. This happens frequently due to today's ever increasing amounts of data.

Inclusion dependencies (INDs) are one of the most important properties of relational datasets [4]. An IND states that all tuples of some attribute-combination in one relation are also contained in some other attribute-combination in the same or (usually) a different relation [12]. This makes INDs important for many tasks, such as data integration [14], query optimization [6], integrity checking [5], or schema (re-)design [9]. In particular, INDs are useful to discover foreign-primary key relationships, which are a necessity for suggesting join paths, data linkage, and data normalization [17].

Proceedings of the VLDB Endowment, Vol. 8, No. 7

However, INDs are unspecified for most real-world datasets, as typically there is simply no schema information of the datasets – or if there is, foreign keys are very often not given. Furthermore, datasets produced by current applications are constantly changing (with data being modified or added) over time so that metadata become quickly out-of-date. Thus, there is a high demand for efficient and automatic IND discovery techniques. The research community has paid relatively little attention to this problem, possibly because of its complexity: finding all unary INDs (i.e., INDs between individual attributes) is quadratic and finding all n-ary INDs (INDs between lists of attributes) requires exponential time in the number of attributes [10]. Besides this complexity, efficiently discovering INDs is also challenging because each IND candidate check potentially needs to read input datasets. This indeed incurs a high I/O cost, which in turn significantly decreases performance.

Existing approaches addressing the IND detection problem already provided some first contributions. For example, Bell and Brockhausen make use of SQL `join`-statements to check the validity of IND candidates [3]; DeMarchi et al. propose the use of inverted indexes to speed up the IND discovery process by simply intersecting such inverted indexes [12]; and Bauckmann et al. present Spider, which basically finds INDs with an adapted sort-merge join approach [2]. Nevertheless, none of these approaches scale with the size of the input dataset and they only solve the IND discovery problem for either unary or n-ary INDs.

We present Binder, a novel IND discovery system that efficiently detects *both* unary and n-ary INDs. Unlike all previous systems, Binder's scalability is not bounded to any technical restrictions, such as main memory size or file handle limits. It uses a divide & conquer approach to apply additional pruning concepts and to process datasets of any size. In particular, we make the following major contributions:

(1) We propose an approach to efficiently divide datasets into smaller partitions that fit in main memory, lazily refining too large partitions. This lazy refinement strategy piggybacks on the actual IND validation process, saving many expensive I/O operations.

(2) We propose a fast IND validation strategy that is based on two different indexes (inverted and dense) instead of a single index or sorting. Additionally, our IND validation strategy applies two new non-statistics-based pruning techniques to speed up the process.

(3) We present a robust IND candidate generation technique that allows Binder to apply the same strategy to discover both all unary and all n-ary INDs. This makes Binder easy to maintain.

(4) We present an exhaustive validation of Binder on many real-world datasets as well as on two synthetic datasets. We experimentally compare it with two other state-of-the-art approaches: Spi-

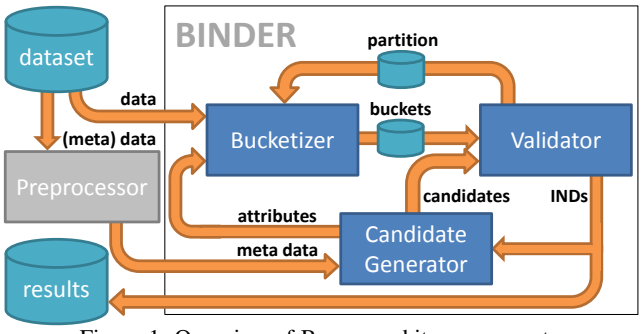

Figure 1: Overview of Binder and its components.

der [2] and Mind [12]. The results show the high superiority of Binder. It is up to more than one order of magnitude (26 times) faster than Spider and up to more than three orders of magnitude (more than 2500 times) faster than Mind.

Next, we give an overview of Binder in Section 2. We then present the three major components of Binder: the `Bucketizer` (Section 3), the `Validator` (Section 4), and the `CandidateGenerator` (Section 5). We present our experimental results in Section 6. We finally discuss related work in Section 7 and conclude with Section 8.

2. BINDER OVERVIEW

We begin this section by formally defining *inclusion dependencies* (INDs). We then give an overview on Binder (short for ***B**ucketing **IN**clusion **D**ependency **E**xtracto**R***).

Preliminaries. We define an IND between two lists of attributes X and Y over a relational schema with relations R_i as $R_j[X] \subseteq R_k[Y]$ ($X \subseteq Y$ for short). In other words, an IND expresses that within a database instance each record r in $R_j[X]$ is also contained in $R_k[Y]$. We refer to the left-hand side X of an IND as *dependent* attribute(s) and the right-hand side as *referenced* attribute(s). Both of these attribute sequences must be of the same size $n = |X| = |Y|$. An IND is said to be *unary* if $n = 1$, otherwise it is *n-ary*. Notice that we do not consider any semantics for *null* values as they do not contribute to INDs: we simply ignore them.

Binder. Our Binder system efficiently discovers all INDs in large datasets. The algorithm, which has been developed in the context of the Metanome data profiling project (`www.metanome.de`), uses the divide & conquer paradigm in order to discover INDs even in very large datasets. The main idea is to partition an input dataset into smaller buckets that can better be mapped into main memory and, then, check these buckets successively for INDs.

Figure 1 depicts the architecture of Binder, which consists of three major components: `Bucketizer`, `Validator`, and `CandidateGenerator`. The additional `Preprocessor` is required by Binder to find some structural properties needed for the IND discovery, such as table identifiers, attribute identifiers, and attribute types. Note that every IND detection system requires such a preprocessing phase for unknown datasets [2, 3, 12]. Like all these works, Binder uses standard techniques for the extraction of structural properties. It is also worth noting that, in terms of performance, the preprocessing step is negligible. Therefore, we do not consider it as a major component of Binder. Below, we briefly discuss each of Binder's three major components. For clarity, we use the dataset in Figure 2, which is an instance of a relational table with four attributes and a universe of six String values.

(1) `Bucketizer`. Given an input dataset, Binder starts by splitting the datasets into several smaller parts (*buckets*) that allow for efficient IND discovery. A bucket is a (potentially deduplicated) subset

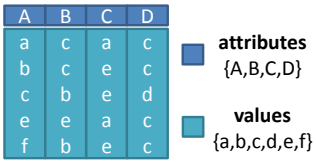

Figure 2: An example dataset that contains one relational table.

of values from a certain single attribute. The `Bucketizer` splits a dataset using hash-partitioning on the attributes values. More precisely, it sequentially reads an input dataset and places each value into a specific bucket according to a given hash-function. The `Bucketizer` uses hash-partitioning instead of a range- or list-partitioning, because it satisfies two important criteria: it puts the values that are equal in the same bucket and, by choosing a good hash-function, it can evenly distribute the values across all buckets. At the end of this process, it writes the generated buckets to disk.

Let us illustrate this process with our example dataset from Figure 2. Figure 3 shows an output produced by the `Bucketizer` for this dataset. Each box of attribute values represents one bucket. We denote a bucket using the tuple (a, n), where a is the attribute of the bucket and n is the bucket hash-number. For instance, the bucket $(A, 2)$ is the second bucket of attribute A. Then, a *partition* is the collection of all buckets with the same hash-number, $p_i = \{(a, n) \mid n = i\}$. Each row in Figure 3 represents a different partition. Note that in the following validation process, an entire partition needs to fit in main memory. If this is not the case, Binder again calls the `Bucketizer` to dynamically refine a partition into smaller sub-partitions that each fit in main memory. Section 3 explains the `Bucketizer` in more detail.

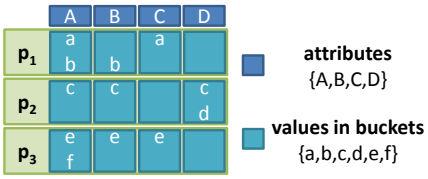

Figure 3: The example dataset bucketized into 12 buckets.

(2) `Validator`. Having divided the input dataset into a set of buckets, Binder starts to successively validate all possible unary IND candidates against this set. The `Validator` component, which is illustrated in Figure 4, is in charge of this validation process and proceeds partition-wise: First, it loads the current partition (i.e., all buckets with the same hash-number) into main memory. If too few partitions have been created so that a partition does not fit into main memory entirely, the `Validator` instructs the `Bucketizer` to refine that partition; then, the `Validator` continues with the sub-partitions.

Once a partition or sub-partition has been loaded, the `Validator` creates two indexes per partition: an inverted index and a dense index. The inverted index allows the efficient checking of candidates, whereas the dense index is used to prune irrelevant candidate checks for a current partition. When moving to the next partition, the `Validator` also prunes entire attributes as *inactive* if all their IND candidates have been falsified (gray buckets in Figure 4). In this way, subsequent partitions become smaller during the validation process, which can reduce the number of lazily executed partition refinements. The `Validator` returns all valid INDs, which are the candidates that "survived" all checks. In Figure 4, this is the IND $F \subseteq A$. We further discuss the `Validator` in Section 4.

(3) `CandidateGenerator`. This is the driver component that defines the set of IND candidates and calls the `Bucketizer` and `Validator` components. Initially, it generates all unary IND candidates from the dataset's metadata and sends them into the IND

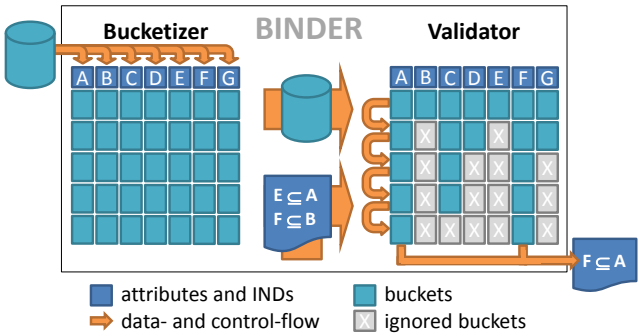

Figure 4: Illustration of the bucketing and validation processes.

discovery process. If only unary INDs shall be detected, the component stops the detection process when it retrieves the valid unary INDs. Otherwise, it uses them to generate and process all binary IND candidates, then ternary IND candidates, etc. The `CandidateGenerator` keeps generating and processing (n+1)-ary IND candidates from discovered n-ary INDs until no more candidates can be generated. We present the `CandidateGenerator` component in detail in Section 5.

3. EFFICIENTLY DIVIDING DATASETS

As discussed in the previous section, BINDER first uses the `Bucketizer` component to split an input dataset into a fixed number of partitions, which are ideally of equal size. Each partition contains one bucket per attribute. A bucket is an unordered collection of attribute values whose hashes lie within the same range.

Algorithm 1 depicts the bucketing process. The `Bucketizer` takes three input parameters: the *attributes* that should be bucketized, the *tables* that point to the actual data, and the *number of partitions* into which the dataset should be split. As Section 6.5 shows, BINDER's performance is not sensitive to the parameter *nPartitions*, so it is set to 10 by default. We call the partitions *seed partitions*, because BINDER might lazily refine them into smaller sub-partitions later on, in case they became too large to fit in main memory.

Overall, the bucketing process consists of three parts: (i) *value partitioning* (ll.), which reads the data and splits it into buckets, (ii) *dynamic memory handling* (ll. 15-21), which spills buckets to disk if main memory is exhausted, and (iii) *bucket management* (ll. 22-30), which writes the buckets to disk while piggybacking some additional statistics. We describe each of these parts in the following three subsections. In the fourth, we discuss the *lazy partition refinement* algorithm.

3.1 Value partitioning

The `Bucketizer` partitions the attribute values of an input dataset by iterating the dataset table-wise in order to keep possibly all buckets for one table at a time in main memory (l. 3). For each table, the `Bucketizer` reads the values in a tuple-wise manner (l. 6). It then fetches all those values from each tuple that belong to the attributes that it has to bucketize (ll. 7 & 8). For each non-null value, it calculates the partition number, *partitionNr*, via hash-partitioning (l. 11): The partition number for the current value is its hash-code modulo the number of partitions. Thus, same attribute values are placed into the same partitions, which preserves valid INDs across different partitions. By using a good, data-type specific hash function, the values are also distributed evenly on the partitions. Then, if the current value is new, i.e., it does not yet exist in its bucket, the `Bucketizer` simply stores it (ll. 12 & 13). At the same time, the algorithm piggybacks a value counter for each attribute, *nValsPerAttr*, that it increases with every new value (l. 14).

Algorithm 1: Bucketing

Data: *attributes*, *tables*, *nPartitions*
Result: *attrSizes*, *checkOrder*

```
1  array attrSizes size | attributes | as Long;
2  array emptyBuckets size nPartitions as Integer;
3  foreach table ∈ tables do
4      array buckets size | table.attr | × nPartitions as Bucket;
5      array nValsPerAttr size | table.attr | as Integer;
6      foreach tuple ∈ read(table) do
7          foreach attr ∈ table.attr do
8              if attr ∉ attributes then continue;
9              value ← line[attr];
10             if value = NULL then continue;
11             partitionNr ← hashCode(value) % nPartitions;
12             if value ∉ buckets[attr][partitionNr] then
13                 buckets[attr][partitionNr] ← ∪ value;
14                 nValsPerAttr[attr] ← + 1;
15         if memoryExhausted() then
16             lAttr ← max(nValsPerAttr);
17             foreach bucket ∈ buckets[lAttr] do
18                 attrSizes[lAttr] ← + sizeOf(bucket);
19                 writeToDisk(bucket);
20                 bucket ← ∅;
21             nValsPerAttr[lAttr] ← 0;
22     foreach attr ∈ table.attr do
23         foreach bucket ∈ buckets[attr] do
24             if bucket = ∅ then
25                 emptyBuckets[attr] ← emptyBuckets[attr] + 1;
26             else
27                 attrSizes[attr] ← + sizeOf(bucket);
28                 writeToDisk(bucket);
29 checkOrder ← orderBy(emptyBuckets);
30 return checkOrder, attrSizes;
```

nValsPerAttr is an array that BINDER uses later, to decide which buckets should be spilled to disk if main memory is exhausted.

3.2 Dynamic memory handling

Every time the `Bucketizer` partitions a tuple, it checks memory consumption. If the main memory is exhausted (e.g., if less than 10% of the memory is free), it spills the buckets of the largest attribute, i.e., with the largest number of attribute values, *nValsPerAttr*, to disk (ll. 15 & 16). Spilling only the largest buckets has two major benefits: First, spilling small buckets introduces the same file handling overhead as for large buckets, but the gain in terms of freed memory is negligible. Second, spilling to disk might cause duplicate values within a bucket, because the values on disk cannot be checked again in the remainder of the bucketing process for efficiency reasons; large buckets contain many different values anyway and are, hence, less likely to receive same values over and over again, which causes them to generate fewer duplicate values when being spilled to disk. The buckets of a table's primary key attribute, for instance, contain no duplicates. To find the largest attribute *lAttr* (and hence the largest buckets), the max() function queries an index that keeps track of the maximum value in the *nValsPerAttr* array (l. 16). Then, the `Bucketizer` iterates the largest buckets and writes them to disk (ll. 17-19). Afterwards, it clears the spilled buckets to free main memory and resets the spilled attribute in the *nValsPerAttr*-field (ll. 20 & 21).

3.3 Bucket management

Once the `Bucketizer` has partitioned a table, it writes all current buckets to disk from where the `Validator` can read them later on (ll. 23-28). In this way, it can reuse the entire main memory for bucketing the next table. While writing buckets to disk, the component also piggybacks the collection of two statistics: *attrSizes* and *emptyBuckets* (ll. 25 & 27). We describe each of these below:

(1) The attribute sizes array, *attrSizes*, stores the in-memory size of each bucketized attribute, and it is used by Binder to identify partitions that do not fit in main memory in the validation process[1]. The `Bucketizer` computes the byte-size of a bucket as follows:

$$size(bucket) = \sum_{string \in bucket} 8 + 8 \cdot \left\lceil \frac{64 + 2 \cdot |string|}{8} \right\rceil$$

This calculation assumes an implementation in Java and a 64 bit system, on which strings need 64 byte for pointers, headers, and length and 2 bytes for each character. After normalizing to a multiple of 8 byte, which is the smallest addressable unit, we add another 8 byte for index-structures needed in the validation phase. In our running example (see Figure 3), the size of bucket (A,1) is hence 160 byte, because it contains two values of length one.

(2) The empty buckets array, *emptyBuckets*, counts the number of empty buckets in each partition to later determine the most promising checking order, *checkOrder*, for the `Validator`. The intuition is to prioritize the partitions with the smallest number of empty buckets, because the `Validator` cannot use empty buckets to invalidate IND candidates. So, the `Validator` aims at identifying and discarding inactive attributes early on by checking those attributes with the lowest number of empty buckets first. For instance, the *emptyBuckets* array for our example in Figure 3 is [1,1,1], because each partition contains one empty bucket; unfortunately, this does not indicate an optimal checking order. Note that empty buckets mainly arise from attributes containing only a few distinct values, such as *gender*-attributes, which typically contain only two unique values.

3.4 Lazy partition refinement

Once the `Bucketizer` has finished splitting an input dataset into buckets, the `Validator` successively uploads seed partitions into main memory in order to validate IND candidates. However, sometimes seed partitions are larger than the main memory capacity. Thus, the `Bucketizer` needs to refine such seed partitions, which means that it splits a partition into smaller sub-partitions that fit into main memory. Refining a seed partition is, however, a costly operation as one has to read and write again most values. Indeed, one could collect some statistics about the input data, such as size or length of the data, to estimate the right number of seed partitions and avoid refining them. Unfortunately, collecting such statistics would introduce a significant overhead to the preprocessing step. Furthermore, the initial bucketing can become very fine-grained for large datasets, which causes the creation of numerous superfluous bucket-files on disk. Too many file operations (create, open, and close), in turn, dominate the execution time of the bucketing process and with it the execution time of the entire IND discovery.

For these reasons, the `Bucketizer` *lazily* refines the partitions whenever necessary. The main idea is to split large seed partitions into smaller sub-partitions while validating INDs. Assume, for instance, that Binder needs to check p_1 of Figure 3 and that only two values fit into main memory; Binder then lazily refines p_1 into [{a},{ },{a},{ }] and [{b},{b},{ },{ }]. This lazy refinement also allows

[1]Determining file-sizes instead is less accurate and more expensive.

Algorithm 2: Refinement

Data: *activeAttr*, *attrSizes*, *partitionNr*
Result: *attrSizes*, *checkOrder*

```
1  availMem ← getAvailableMemory();
2  partSize ← 0;
3  foreach attr ∈ activeAttr do
4      partSize ← + attrSizes[attr]/nPartitions;
5  nSubPartitions ← ⌈partSize/availMem⌉;
6  tables ← getTablesFromFiles(activeAttr, partitionNr);
7  if nPartitions > 1 then
8      checkOrder, attrSizes ← bucketize(activeAttr, tables,
9                                        nSubPartitions);
10     return checkOrder;
11 else
12     return {partitionNr};
```

Binder to dynamically reduce the number of sub-partitions when the number of active attributes decreases from one seed partition to the next one. For instance, if all IND candidates of an attribute are invalidated in some partition, Binder does not need to refine the attribute's buckets in all subsequent partitions, saving much I/O. In contrast to estimating the number of seed partitions, lazy refinement creates much fewer files and hence much less overhead for three reasons: (i) the number of required sub-partitions can more precisely be determined in the validation process, (ii) the number of files decreases with every invalidated attribute, and (iii) some small attributes can even stay in memory after refinement.

Algorithm 2 details the lazy refinement process. To refine a partition, the `Bucketizer` requires three inputs: *attrSizes* (size of the attributes' buckets in byte), *partitionNr* (identifier of the partition to be refined), and *activeAttr* (attributes to consider for refinement). Overall, the refinement process consists of two parts: (i) the *sub-partition number calculation* (ll. 1-5), which calculates the number of sub-partitions in which a seed partition has to be split, and (ii) the *value re-bucketing* (ll. 6-9), which splits a large partition into smaller sub-partitions. We explain these two parts below.

(1) *Sub-partition number calculation.* To decide if a split is necessary, the `Bucketizer` needs to know the in-memory size of each attribute in a partition. The component can get this information from the *attrSizes* array, which it collected during the bucketing process (see Section 3.3). Let *availMem* be the available main memory and *attr* one attribute. As the hash function created seed partitions of equal size, the `Bucketizer` can now calculate the size of an attributes's bucket independently of the actual bucket number as:

$$size(attr) = attrSizes[attr]/nPartitions$$

The `Bucketizer` then calculates the size of a partition *partitionNr* as the sum of the sizes of all its active attributes:

$$partSize = \sum_{attr \in activeAttr} size(attr)$$

Thus, given a seed partition, the `Bucketizer` simply returns the seed partition number without refinement if $availMem > partSize$. Otherwise, it needs to split the seed partition into *nPartitions* sub-partitions as follows:

$$nSubPartitions = \lceil partSize/availMem \rceil$$

(2) *Value re-bucketing.* If refinement is needed, Binder re-applies the bucketing process depicted in Algorithm 1 on the bucket files.

To run the bucketing process, the `Bucketizer` first calls the function `getTablesFromFiles()` that interprets each bucket as a table containing only one attribute (l. 6). Then, it successively reads the buckets of the current partition, re-hashes their values, and writes the new buckets back to disk (l. 8).

It is worth noting that distributing values from a bucket into different sub-partitions in an efficient manner is challenging for two reasons: *(i)* the values in each bucket are already similar with respect to their hash-values and thus redistributing them becomes harder, and *(ii)* refining seed partitions requires two additional I/O operations (for reading from and writing back to disk) for each value of an active attribute in a seed partition. BINDER addresses these two aspects as follows:

(i) To redistribute the values in a bucket, the `Bucketizer` re-partitions the values into *nSubPartitions* as follows:

$$x = \frac{hash(value)\ \%\ (nPartitions \cdot nSubPartitions) - partitionNr}{nPartitions}$$

Here, x is the sub-bucket number with $x \in [0, nSubPartitions - 1]$ denoting the sub-bucket for the given *value*. Taking the hashes of the values modulo the number of seed partitions *nPartitions* multiplied by the number of required sub-partitions *nSubPartitions* leaves us with *nSubPartitions* different numbers. For instance, if $nPartitions = 10$, $partitionNr = 8$, and $nSubPartitions = 2$, we obtain numbers in $\{8, 18\}$, because the hash-values modulo *nPartitions* always give us the same number, which is *partitionNr*. By subtracting the current partition number *partitionNr* from the modulo and, then, dividing by *nPartitions*, we get an integer x in $[0, nSubPartitions - 1]$ assigning the current value to its sub-bucket.

(ii) Concerning the additional I/O operations, we consider that each attribute can allocate at most $m = availMem / |activeAttr|$ memory for each of its buckets. However, in practice, most buckets are much smaller than m. Thus, our `Bucketizer` saves many I/O operations by not writing and reading again the sub-buckets of such small buckets back to disk, i.e., whenever $m < size_{attr}$.

4. FAST IND DISCOVERY

Given a set of IND candidates, the `Validator` component successively checks them against the bucketized dataset. Algorithm 3 shows the validation process in detail. While the `Validator` reads the bucketized dataset directly from disk, it requires three additional inputs: the IND *candidates* that should be checked, the *checkOrder* defining the checking order of the partitions, and the *attrSizes* indicating the in-memory size of each bucket. See Section 3.3 for details about the *checkOrder* and *attrSizes* structures. The *candidates* input is a map that points each possible dependent attribute (i.e., included attribute) to all those attributes that it might reference (i.e., that it might be included in).

During the validation process, the `Validator` removes all invalid INDs from the *candidates* map so that only the valid INDs survive until the end of the process. Overall, the validation process consists of two parts: (i) the *partition traversal* (ll. 1-6), which iterates the partitions and maintains the attributes, and (ii) the *candidate validation* (ll. 7-22), which checks the candidates against a current partition. We explain both parts in the following two sections. Afterwards, we take a closer look at the candidate pruning capabilities of BINDER and discuss our design decisions.

4.1 Partition traversal

As its first step, the `Validator` collects all active attributes *activeAttr*, which are all attributes that participate in at least one IND candidate (l. 1). The `Validator` uses *activeAttr* to prune inactive attributes during the validation (l. 6): if an attribute is removed from all IND candidates, it is also removed from this set and ignored for the rest of the validation process. The `Validator` checks IND candidates against the bucketized dataset in the checking order, *checkOrder*, previously defined by the `Bucketizer` (l. 2). Before validation, the `Validator` calls the `Bucketizer` to refine the current seed partition into smaller sub-partitions if necessary (l. 3), i.e., if the partition does not fit in main memory (see Section 3.4). Notice that the current seed partition is the only sub-partition if no refinement was needed. After the refinement process, the `Validator` iterates the sub-partitions to check the candidates against them (l. 4). As the `Validator` might invalidate candidates on the current sub-partition in each iteration, it first updates the *activeAttr* set before starting the validation of the current iteration (l. 5). If no active attributes are left, which means that all IND candidates became invalid, the `Validator` stops the partition traversal (l. 6); otherwise, it proceeds to validate the IND candidates.

Algorithm 3: Validation

Data: *candidates*, *checkOrder*, *attrSizes*
Result: *candidates*

```
1  activeAttr ← getKeysAndValues(candidates);
2  foreach partitionNr ∈ checkOrder do
3      subPrtNrs ← refine(activeAttr, attrSizes, partitionNr);
4      foreach subPartitionNr ∈ subPrtNrs do
5          activeAttr ← getKeysAndValues(candidates);
6          if activeAttr = ∅ then break all;
7          map attr2value as Integer to {};
8          map value2attr as String to {};
9          foreach attr ∈ activeAttr do
10             bucket ← readFromDisk(attr, subPartitionNr);
11             attr2value.get(attr) ← bucket;
12             foreach value ∈ bucket do
13                 value2attr.get(value) ← ∪ attr;
14         foreach attr ∈ activeAttr do
15             foreach value ∈ attr2value.get(attr) do
16                 if candidates.get(attr) = ∅ then break;
17                 if value ∉ value2attr.keys then continue;
18                 attrGrp ← value2attr.get(value);
19                 foreach dep ∈ attrGrp do
20                     candidates.get(dep) ← \ attrGrp;
21                 value2attr.remove(value);
22 return candidates;
```

4.2 Candidate validation

The `Validator` first loads the current sub-partition into main memory and then checks the IND candidates on this sub-partition. To support fast checking, the `Validator` now builds two indexes upon the partition's values: the dense index *attr2values* and the inverted index *values2attr*. For the following illustration, assume that our running example dataset shown in Figure 2 has been bucketized into only one partition. Figure 5 then shows the two index structures that the `Validator` would create for this single partition.

(1) *Dense Index.* The index *attr2values* maps each attribute to the set of values contained in this attribute. The `Validator` constructs the index when loading a partition into main memory in a bucket-wise manner. As each bucket represents the values of one

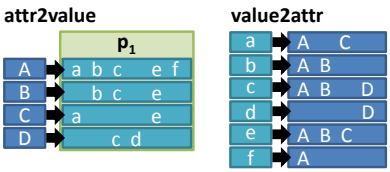

Figure 5: Dense index *attr2value* and inverted index *value2attr* for our example attributes {A,B,C,D} and values {a,b,c,d,e,f}.

attribute, the `Validator` can easily build this index with a negligible overhead. When validating a current partition, the `Validator` uses *attr2values* to discard attributes that become independent of all other attributes, i.e., that are not included in any other attribute.

(2) *Inverted Index.* The inverted index *values2attr* maps each value to all those attributes that contain this value. Similar to DeMarchi's algorithm [12] and Bauckmann's SPIDER algorithm [2], the `Validator` uses the sets of attributes containing a common value to efficiently validate IND candidates via set intersection.

Index initialization. The `Validator` initializes the two indexes in an attribute-wise manner (ll. 11-13). For each active attribute, it reads the corresponding bucket from disk. Then, it points the active attribute to the read values in the *attr2values* index (left side of Figure 5). To initialize the *values2attr* index, the `Validator` inverts the key-value pairs and points each value to all those attributes where it occurs in (right side of Figure 5).

IND validation. Having initialized the two indexes with the current sub-partition, the `Validator` uses them to efficiently remove non-inclusions from the set of IND *candidates*. For this purpose, it again iterates all active attributes and, for each active attribute, all the attribute's values (ll. 14 & 15). If the current attribute does not depend on any other attribute anymore, i.e., its *candidate* entry is empty, the `Validator` does not check any other values of this attribute and it can proceed with the next active attribute (l. 16). Sometimes, the current value has already been handled with a different attribute and the inverted index does not contain the value anymore. Then, the `Validator` proceeds with the next value of the same attribute (l. 17). Otherwise, it retrieves the group of attributes *attrGrp* containing the current value and intersects the set of referenced attributes of each of the group's members with this group (ll. 18-20). The intuition behind this intersection is that none of the *attrGrp* attributes can be included in an attribute that is not part of this group, because it would not contain the value of this particular attribute group. When the intersections are done, the `Validator` removes the current value from the inverted index *values2attr* in order to avoid checking the same attribute group again for different members of the group (l. 21).

Example. Let us walk through the validation process once again with the example indexes shown in Figure 5. We use Table 1 to list the intermediate results of the validation process. Each column in the table represents a dependent attribute and the cells list the attributes that are referenced by the respective dependent attribute. When reading the table from top to bottom, the initial IND candidates assume that each attribute is included in all three other attributes. Then, the `Validator` starts to check these candidates by looking up attribute A in the *attr2values* index and its first value a in the *values2attr* index. There, the `Validator` finds the attribute group $\{A, C\}$. Then, it intersects the referenced attribute sets of A and C with this set. Thereby, we retrieve $\{C\}$ for attribute A and $\{A\}$ for attribute C. Next, the `Validator` continues with the second value of attribute A, which is b, and finds the attribute group $\{A, B\}$. After intersecting attribute A's set of referenced attributes with this

Table 1: Checking process over the example indexes of Figure 5

	A	B	C	D
look up	**B,C,D**	**A,C,D**	**A,B,D**	**A,B,C**
A → a → A,C	**C**	A,C,D	**A**	A,B,C
A → b → A,B	**-**	**A**	A	A,B,C
B → c → A,B,D	**-**	**A**	A	**A,B**
B → e → A,B,C	**-**	**A**	**A**	A,B
D → d → D	-	A	A	**-**

attribute group, attribute A's set is empty. Thus, the `Validator` stops the checking of attribute A and continues with attribute B.

For attribute B, the `Validator` cannot find its first value b in the inverted index anymore, because it has already been handled. Therefore, it continues with value c and handles the corresponding attribute group $\{A, B, D\}$. The same follows for e. As the `Validator` has checked all values of B now, it moves to attribute C. Because all of attribute C's values have also been handled, it continues checking attribute D. Here, it finds the value d unchecked, which disassociates attribute D from A and B. Finally, the `Validator` terminates yielding the two inclusion dependencies $B \subseteq A$ and $C \subseteq A$.

4.3 Candidate pruning

A common practice to prune IND candidates is to collect some statistics about the buckets, such as min and max values, in order to prune some attributes before even reading their first buckets. However, in early evaluations, we found that collecting such statistics is more expensive than their pruning effect: We observed that the bucketing process dominates the entire IND discovery process by taking up 95% of the execution time; hence, additional costs for statistic collection in this step dominate any possible pruning-gain. Therefore, the `Validator` relies on two non-statistics-based pruning techniques to significantly speed up the validation process: *intra-partition* and *inter-partition* pruning.

Intra-partition pruning. During the validation of a partition, some attributes become independent of all other attributes, such as attribute A in our example. This means that they no longer appear as a dependent attribute in any IND candidate. In these cases, the `Validator` directly prunes the rest of the attribute's values from the validation process by skipping to the next active attribute in the *attr2values* index. The value f in our example, for instance, is never checked. In real-world datasets, many of such f-values exist, because attributes containing a large number of different values have an especially high chance of not being included in any other attribute.

Inter-partition pruning. After validating a partition, attributes become inactive if they neither depend on other attributes nor get referenced. Thus, by frequently updating the *activeAttr* set, the `Validator` can prune entire attributes from further validation processes. In consequence, the partitions become smaller and smaller, which continuously reduces the time needed for partition refinement, bucket loading, and index creation. Fewer attributes also reduce the average size of attribute groups in the *values2attr* index, which in turn makes the intersections faster.

In summary, the `Validator` does not need to read and check all attributes entirely due to the two indexes and its lazy bucketing and refinement techniques (if not all attributes participate in any inclusion dependency). All candidates that "survive" the validation on all partitions are valid inclusion dependencies.

5. IND CANDIDATE GENERATION

The `CandidateGenerator` is the driver component of the entire IND discovery process. It generates the IND candidates, instructs

Algorithm 4: Candidate Generation

```
   Data: attributes, tables, dataTypes, nPartitions
   Result: dep2refs
1  map dep2refs as Integer to {};
2  map candidates as Integer to {};
   // Unary IND detection
3  checkOrder, attrSizes ← bucketize(attributes, tables,
4                                    nPartitions);
5  emptyAttr ← {a ∈ attributes | attrSizes[a] = 0};
6  foreach cand ∈ attributes × attributes do
7      if cand[0] = cand[1] then
8          continue;
9      if dataTypes[cand[0]] ≠ dataTypes[cand[1]] then
10         continue;
11     if cand[0] ∈ emptyAttr then
12         dep2refs.get(cand[0]) ← ∪ cand[1];
13     else
14         candidates.get(cand[0]) ← ∪ cand[1];
15 dep2refs ← ∪ validate(candidates, checkOrder, attrSizes);
   // N-ary IND detection
16 lastDep2ref ← dep2refs;
17 while lastDep2ref ≠ ∅ do
18     candidates ← generateNext(tables, lastDep2ref);
19     if candidates = ∅ then break;
20     attrCombinations ← getKeysAndValues(candidates);
21     checkOrder, attrSizes ← bucketize(attrCombinations,
22                                       tables, nPartitions);
23     lastDep2ref ← validate(candidates, checkOrder,
24                            attrSizes);
25     dep2refs ← ∪ lastDep2ref
26 return dep2refs;
```

the Bucketizer to partition the data accordingly and then calls the Validator to check the candidates on the bucketized dataset. It is worth noting that Binder uses the same algorithms (the bucketing, validation, and candidate generation processes) to discover both all unary and all n-ary INDs. To the best of our knowledge Binder is the first approach to achieve this.

Algorithm 4 shows the candidate generation process in detail. It takes four parameters: the three arrays *attributes*, *tables*, and *dataTypes*, which store metadata about the dataset, and the *nPartitions* variable, which defines the number of seed partitions. The CandidateGenerator detects both unary and n-ary INDs. At first, the component generates all unary IND candidates and runs them through the bucketing and validation processes (ll. 1-15). Then, if n-ary INDs should be detected as well, the CandidateGenerator starts a level-wise generation and validation process for n-ary IND candidates (ll. 16-26). Each level represents the INDs of size *i*. The iterative process uses the already discovered INDs of size *i* to generate IND candidates of size $i + 1$. While this traversal strategy is already known from previous works [1, 12], our candidate validation techniques contribute a significant improvement and simplification to the checking of n-ary IND candidates.

5.1 Unary IND detection

The CandidateGenerator starts by defining the *dep2refs* map, in which we store all valid INDs, and the *candidates* map, in which we store the IND candidates that still need to be checked (ll. 1 & 2). Both data structures map dependent attributes to lists of referenced attributes. The algorithm then calls the Bucketizer to partition the dataset for the unary IND detection (l. 3). The Bucketizer splits all *attributes* of all *tables* into *nPartitions* buckets as explained in Section 3. Next, the CandidateGenerator collects all empty attributes in the *emptyAttr* set using the previously measured attribute sizes (l. 5). An empty attribute is an attribute that contains no values. Using the attributes, their data types, and the set of empty attributes, the CandidateGenerator iterates the set of all possible unary IND candidates, which is the cross product of all attributes (l. 6). Each candidate *cand* is a pair of one dependent attribute *cand*[0] and one referenced attribute *cand*[1]. If both are the same, the IND is trivial and is discarded (ll. 7 & 8).

Like the state-of-the-art [2, 12], the CandidateGenerator also discards candidates containing differently typed attributes (ll. 9 & 10). However, if INDs between differently typed attributes are of interest, e.g., if numeric columns can be included in string columns, this type-filter can be omitted. Furthermore, the CandidateGenerator excludes empty attributes, which are contained in all other attributes by definition, from the validation process. After discarding some first candidates, the remaining candidates are either valid per definition, i.e., the dependent attribute is empty, or they need to be checked against the data (l. 14). The CandidateGenerator calls the Validator to check the IND candidates (see Secion 4) and places the valid INDs into the *dep2refs* map (l. 15). If only unary INDs are required, the algorithm stops here; otherwise, it continues with the discovery of n-ary INDs.

5.2 N-ary IND detection

For the discovery of n-ary INDs, the CandidateGenerator incrementally generates and checks ever larger candidates. The generation is based on the apriori-gen-algorithm [1], which traverses the lattice of attribute combinations level-wise. In detail, the CandidateGenerator first copies the already discovered unary INDs into the *lastDep2ref* map (l. 16), which stores all discovered INDs of the last finished validation, i.e., of the last iteration. While the *lastDep2ref* map is not empty, i.e., the last validation found at least one new inclusion dependency, the CandidateGenerator keeps generating and checking ever larger INDs (ll. 17-25).

Candidate generation. To generate the n-ary IND candidates *nAryCandidates* of size $i + 1$ from the valid INDs *lastDep2ref* of size *i*, the CandidateGenerator uses the generateNext() function (l. 18). Basically, for each IND $R_j[X] \subseteq R_k[Y]$ with $|X| = |Y| = i$, this function finds all IND candidates $R_j[XA] \subseteq R_k[YB]$ so that:

(1) $R_j[X] \subseteq R_k[Y]$ and $R_j[A] \subseteq R_k[B]$

(2) $\forall X_i \in X : X_i < A$

(3) $A \notin X$, $B \notin Y$, and $R_j[A] \neq \emptyset$

where X and Y are attribute lists and A and B are individual attributes. In other words, both sequences and single attributes are pair-wise INDs (1). As INDs are permutable, e.g., if $R_j[AB] \subseteq R_k[CD]$ is an IND then $R_j[BA] \subseteq R_k[DC]$ is an IND as well, the CandidateGenerator checks only one such permutation. Therefore, it orders the dependent attributes lexicographically and generates only those candidates whose attributes follow this order (2). The CandidateGenerator also does not generate trivial left or right hand sides that contain an attribute twice or that contain an empty attribute (3).

Re-bucketing. Having generated the next *nAryCandidates*, the CandidateGenerator needs to bucketize again the dataset according to these new candidates. It cannot reuse the bucketized

dataset from the previous run, because the information about co-occurring values of different attributes gets lost when the values are bucketized. For instance, if the `CandidateGenerator` has to check the candidate $R_j[AB] \subseteq R_k[CD]$, then it checks if $\forall r \in R_j[AB] : r \in R_k[CD]$. As record r cannot be reconstructed from previous bucketings, the `CandidateGenerator` has to re-execute the bucketing algorithm with one small difference: Instead of single attribute values, Algorithm 1 bucketizes records from attribute combinations *attrCombinations* that either occur as a dependent or referenced attribute combination in any IND candidate. Technically, the `Bucketizer` can simply combine the values of such records with a *dedicated* separator character to then bucketize the combined values. For instance, consider that we need to check the IND candidate $R_j[AB] \subseteq R_k[CD]$. Now, assume that the `Bucketizer` reads the record (f, b, e, c) from our example schema $R_1[A, B, C, D]$. Then, it partitions the value $'f\#b'$ for $R_1[AB]$ and $'e\#c'$ for $R_1[CD]$.

It is worth emphasizing that the resulting buckets can become much larger than the buckets created for the unary IND checking: First, the combined values for n-ary IND candidates of size i are i-times larger on average than the single values, without counting the separator character. Second, the number of non-duplicate values increases exponentially with the size of the IND candidates so that more values are to be stored. Binder can still handle this space complexity through its dynamic memory handling (Section 3.2) and the lazy partition refinement (Section 3.4) techniques.

Validation. After the bucketing, the `CandidateGenerator` calls the `Validator` with the current set of n-ary IND candidates. Here, the validation of n-ary candidates is the same as the validation of unary candidates. The `Validator` has just to consider that the buckets refer to attribute combinations, e.g., to $R[AB]$. After validating the n-ary IND candidates of size $i + 1$, the `CandidateGenerator` then supplements the final result *dep2ref* with the set of newly discovered INDs. In case that no new INDs are found, the `CandidateGenerator` stops the level-wise search, because all unary and n-ary INDs have already been discovered. As a result, Binder reports the *dep2ref* map that now contains all valid INDs.

6. EVALUATION

We evaluate and compare the performance of Binder with two state-of-the-art systems for IND discovery. In particular, we carried out this evaluation with five questions in mind: How good does Binder perform when both (i) *varying the number of rows* (Section 6.2) and (ii) *varying the number of columns* (Section 6.3)? How well does Binder behave when processing (iii) *different datasets* (Section 6.4)? What is the performance impact regarding the (iv) *internal techniques* of Binder (Section 6.5) and how does Binder perform for (v) *discovering n-ary INDs* (Section 6.6)?

6.1 Experimental setup

Hardware. All experiments were run on a Dell PowerEdge R620 with two Intel Xeon E5-2650 2.00 GHz CPUs and 128 GB DDR3 RAM. The server runs CentOS 6.4 as operating system. For some experiments, we reduce the server's memory to 8 GB in order to evaluate the algorithm's performance on limited memory resources.

Systems. We compare against two other systems, namely Spider [2], which to date is the fastest algorithm for unary INDs discovery, and Mind [12], which we believe is the most popular system for n-ary IND discovery. For Binder we set the input parameter *nPartitions* to 10 for all experiments and show in Section 6.5 why this is a good default value. We implemented all three algorithms within our Metanome data profiling tool using Java 7. The Metanome framework standardizes the configuration of different profiling algorithms and their input and output handling[2]. Hence, it makes them easier to compare. Binder and the other systems' binaries are publicly available for experimental reproducibility[3].

Table 2: Datasets and their characteristics

Name	File Size	Attr.	Unaries	N-aries	n_{max}
COMA	20 KB	4	0	0	1
SCOP	16 MB	22	43	40	4
CENSUS	112 MB	48	73	147	6
WIKIPEDIA	540 MB	14	2	0	1
BIOSQL	560 MB	148	12463	22	2
WIKIRANK	697 MB	35	321	339	7
LOD	830 MB	41	298	1361005	8
ENSEMBL	836 MB	448	142510	100	4
CATH	908 MB	115	62	81	3
TESMA	1,1 GB	128	1780	0	1
PDB	44 GB	2790	800651	unknown	
PLISTA	61 GB	140	4877	unknown	
TPC-H	100 GB	61	90	6	2

Data storage. Our experiments consider both a database and raw files as data storage, because this choice influences the system's runtimes. Spider, for instance, uses SQL order-by statements on database inputs for its sorting phase and external memory sorting on files. Mind uses SQL for all inclusion checks and, hence, only runs on databases. Binder reads the dataset only once and then maintains the data itself, which is why the algorithm executes equally on both storage types. For our experiments, we use CSV-formatted files and the IBM DB2 9.7 database in its default setup.

Datasets. Our experiments build upon both synthetic and real-world datasets. The real-world datasets we use are: COMA, WIKIPEDIA, and WIKIRANK, which are small datasets containing image descriptions, page statistics, and link information, respectively, that we crawled from the Wikipedia knowledge base; SCOP, BIOSQL, ENSEMBLE, CATH, and PDB, which are all excerpts from biological databases on proteins, dna, and genomes; CENSUS, which contains data about peoples' life circumstances, education, and income; LOD, which is an excerpt of linked open data on famous persons and stores many RDF-triples in relational format; and PLISTA [7], which contains anonymized web-log data provided by the advertisement company Plista; The last two datasets TESMA and TPC-H are synthetic datasets, which we generated with the db-tesma tool for person data and the dbgen tool for business data, respectively. Table 2 lists these datasets with their *file size* on disk, number of *attributes*, number of all *unary* INDs, number of all *n-ary* INDs with $n > 1$, and the INDs' maximum arity n_{max}. Usually, most n-ary INDs are of size $n = 2$, but in the LOD dataset most n-ary INDs are of size $n = 4$. A link collection to these datasets and the data generation tools is available online[3].

6.2 Varying the number of rows

We start evaluating Binder with regard to the length of the dataset, i.e., the number of rows. For this experiment we generated five TPC-H datasets with different scale factors from 1 to 70. The two left-side charts in Figure 6 show the result of the experiment for 128 GB (top) and 8 GB (bottom) of main memory.

We observe that with 128 GB of main memory, Binder is up to 6.2 times faster on file inputs and up to 1.6 times faster on a database than Spider. Binder outperforms Spider for three main reasons: First, building indexes for the IND validation is faster than

[2]http://www.metanome.de

[3]http://hpi.de/en/naumann/projects/repeatability/data-profiling/ind

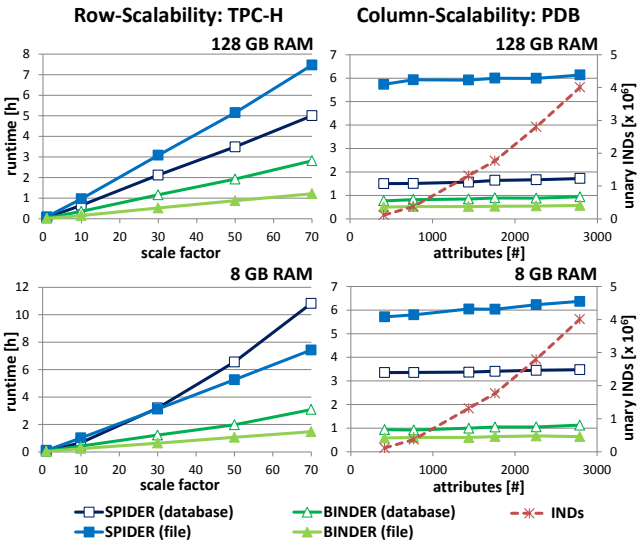

Figure 6: Runtimes measured scaling rows or columns

Spider's sorting. Second, the indexes are not constructed for all attribute values, because some inactive attributes can be discarded early on. Third, Binder reads the data once, whereas Spider queries the data multiple times in order to sort the different attributes. The results also show that Binder performs worse on database inputs than on file inputs although the algorithm is exactly the same. In fact, its execution time almost doubles. The reason is the database overhead of query parsing, record formatting, and, in particular, inter-process data transfer. Spider, on the other hand, profits from a database: it uses the built-in sorting algorithms, which are highly optimized for the given data. Additionally, by directly eliminating duplicate values, the database also reduces the amount of data that it sends to the validation process. Despite these advantages, Binder still clearly outperforms Spider even on database inputs.

We also observe that with 8 GB of main memory, both algorithms become slower, as expected. The runtimes of Binder increased by 8-9% and the runtimes of Spider by 10%. This is because both need their external memory techniques, which is spilling and refinement in Binder and external memory sorting in Spider. However, we observe that the runtime of Spider on the database significantly increased, because the database sorting algorithm is apparently less performant on insufficient main memory. As a result, Binder is now up to 3.5 times faster than Spider. When using raw files as input, Binder is up to 5.1 times faster than Spider. This is a bit less than when having 128 GB of RAM, because the refinement strategy of Binder requires slightly more spills to disk.

In summary, Binder significantly outperforms Spider on both database and file inputs. It scales linearly with an increasing number of rows, because the bucketing process scales linearly and dominates the overall runtime of Binder with 92-95%; in fact, the scalability is slightly super-linear, because the relative runtime costs for the validation process decreased with the length of the data from 8 to 5% due to our pruning techniques.

6.3 Varying the number of columns

We now evaluate Binder with regard to the width of the dataset, i.e., the number of attributes. In these experiments we used the PDB dataset, which comprises 2,790 attributes in total. We start with a subset of 411 attributes and continuously add tables from the PDB dataset to its subset. Note that we had to increase the open files limit in the operating system from 1,024 to 4,096 for Spider, to avoid the "Too many open files" exception.

The two right-side charts in Figure 6 show the result of the experiment for 128 GB (top) and 8 GB (bottom). We additionally plot the number of discovered INDs (red line in the charts) in these charts as they increase with the number of attributes. Similar to the row scalability experiment, Binder significantly outperforms Spider: (i) it is up to 1.8 times faster on a database and up to 10.7 times faster on raw files with 128 GB of main memory; and (ii) it is up to 3.1 times faster on a database and up to 9.9 times faster on raw files with 8 GB of main memory. We see that these improvement factors stay constant when adding more attributes, because the two IND validation strategies have the same complexity with respect to the number of attributes. However, it is worth noting that although the number of INDs increases rapidly, the runtimes of both algorithms only increase slightly. This is because the bucketing (for Binder) and sorting (for Spider) processes dominate the runtimes with 95% and 99% respectively.

6.4 Varying the datasets

The previous sections have shown that Binder's performance does not depend on the number of rows or columns in the dataset. As the following experiments on several real-world and two synthetic datasets will show, it instead depends on four other characteristics of the input datasets: (i) the *dataset size*, (ii) the *number of duplicate values per attribute*, (iii) the *average number of attributes per table*, and (iv) the *number of prunable attributes*, i.e., attributes not being part of any IND. Thus, to better evaluate Binder under the influence of these characteristics, we evaluated it on different datasets and compare its performance to that of Spider.

Figure 7 shows the results of these experiments. We again executed all experiments with 128 GB (top) and 8 GB (bottom) main memory. Notice that only the runtimes for the large datasets differ across the two memory settings. Overall, we observe that Binder outperforms Spider on all datasets, expect COMA and SCOP. We examine these results with respect to the four characteristics mentioned above:

(1) Dataset size. As we observe in the scalability experiments in Sections 6.2 and 6.3, the improvement of Binder over Spider does not depend on the number of rows or columns if the datasets are sufficiently large. However, we observe in these results that Binder is slower than Spider if the datasets are very small, such the COMA dataset with only 20 KB. This is because the file creation costs on disk dominates the runtimes of Binder: Binder creates ten files per attribute[4] whereas Spider only creates one file per attribute. For large datasets, however, the file creation costs are negligible in Binder. Therefore, Binder could simply keep its buckets in main memory to handle with small datasets.

(2) Number of duplicate values per attribute. Spider has an advantage over Binder if the input dataset contains many duplicate values and the IND detection is executed on a database. This is because Spider uses database functionality to remove such duplicates. The results for the CENSUS, BIOSSQL, and ENSEMBL datasets, which contain many duplicate values, show that Spider on a database can compete against Binder on a database. This also shows the efficiency of Binder to eliminate duplicate values in the bucketing process. In contrast to these results, when running on top of raw datasets, Binder again significantly outperforms Spider. We also observe that Binder is much better than Spider for the generated TESMA dataset, which contains only few duplicate values.

(3) Average number of attributes per table. The experiments in Section 6.3 have shown that the overall number of attributes does

[4]In contrast to Spider, Binder only opens one of these files at a time

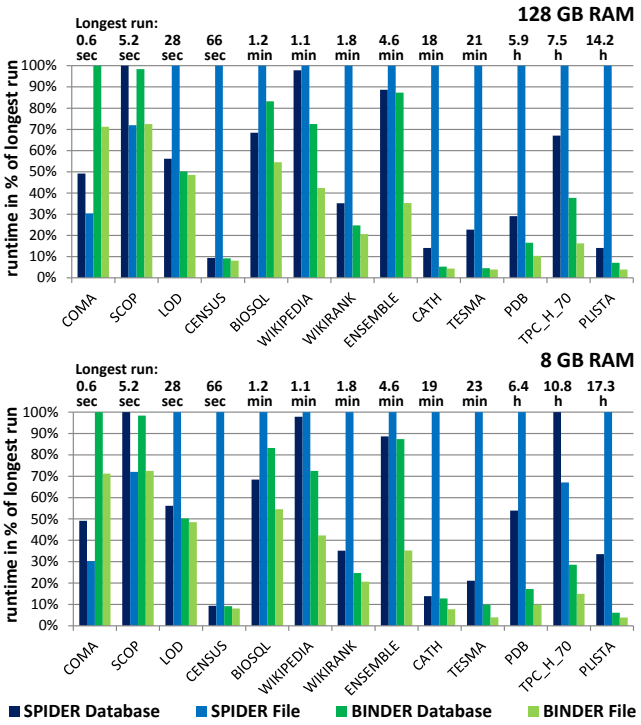

Figure 7: Runtime comparison on different datasets

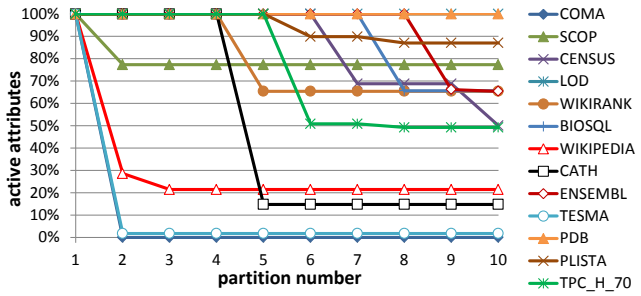

Figure 8: Active attributes per partition for the different datasets

not influence the performance difference between Binder and Spider. However, we particularly observed that if the average number of attributes per table is high, Binder's performance is not affected as much as the performance of Spider: Spider needs to access a large table once for reading each of its attributes, which introduces a large overhead especially if the dataset needs to be read from files. Binder, on the other hand, needs to spill buckets more often to disk, but this introduces only a small overhead as the buckets are written to disk anyway. The results for the TESMA and PLISTA dataset show this aspect best: Having 32 and 35 attributes per table on average, respectively, Binder significantly outperforms Spider.

(4) Number of prunable attributes. In contrast to Spider, which cannot prune inactive attributes before or during sorting values, Binder fully profits from attributes that are not part of any IND thanks to the inter-partition pruning (Section 4.3). For the SCOP dataset, for instance, Binder performs as good as Spider even if this dataset is still very small (16 MB). This is because Binder already prunes all attributes that are not part of any IND while processing the first partition. In this way, Binder saves significant indexing time whereas Spider cannot save sorting time for prunable attributes. Figure 8 shows the pruning power in more detail: it lists the percentage of active attributes for the different partitions showing how many attributes Binder pruned in the different partitions. Notice that all attributes that survive as active attributes until Partition 10 are part of at least one valid IND. For instance, 50% of the attributes finally appear in at least one IND for the TPC-H dataset. In these results, we observe that Binder achieves the best pruning on the COMA, TESMA, WIKIPEDIA, and SCOP datasets. It can prune almost all unimportant attributes in the first partitions. This means that the values of these attributes are not read, indexed, or compared in the following partitions. Nonetheless, we see that the pruning capabilities of Binder have no effect for the LOD and PDB datasets, because all of their attributes either depend on another attribute or reference another attribute.

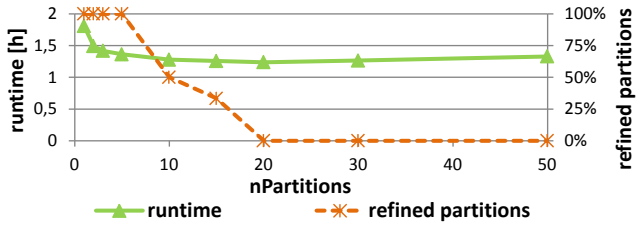

Figure 9: The nPartitions parameter on TPC-H 70 using 8 GB RAM

6.5 BINDER in-depth

So far, we have shown the high superiority of Binder over Spider. We now evaluate Binder in detail studying the impact of: (i) the *nPartitions* parameter; (ii) the data structures we use in the validation process; and (iii) the index-based candidate checking.

(1) Number of seed buckets. Recall that the *nPartitions* parameter specifies the number of seed buckets that Binder should use. Although one could use any value for this parameter – thanks to the lazy refinement used by Binder –, the number of seed buckets influences the performance of Binder as shown in Figure 9. It shows both the runtime of Binder for different *nPartitions* values and the percentage of refined partitions for each number of seed buckets on the TPC-H 70 dataset. As expected, on the one hand, we observe that taking a small number of seed partitions (two to three) decreases the performance of Binder, because it needs many costly bucket refinements. On the other hand, we observe that choosing a large number of seed partitions, e.g., 50, also reduces the performance of Binder, because the file overhead increases as well. In between very small and very large values, we see that Binder is not sensitive to the *nPartitions* parameter, showing its robustness to this parameter. It is worth noting that Spider takes 7.4 hours to compute the same dataset, which is still more than 4 times slower than the worst choice of *nPartitions* in Binder. Notice that we observed identical results for the PLISTA and PDB dataests and very similar results for the other datasets.

(2) Lists vs. BitSets. In the validation process, Binder checks the generated IND candidates against the bucketized dataset by intersecting the candidates referenced attribute sets with those sets of attributes that all contain a same value. Technically, we can maintain the attribute sets as Lists or BitSets. BitSets have two advantages over Lists if most bits are set: they offer a smaller memory footprint and their intersection is faster. Thus, we implemented and tested both data structures. In our experiments, we did not observe a clear performance difference between the two data structures. This is because the attribute sets are typically very small and the intersection costs become almost constant costs [12]. However, we observed a higher memory consumption for BitSets on datasets with many attributes, e.g., 41% higher on the PDB dataset, because they are very sparsely populated. Furthermore, the number of candidates increases quadratically with the number of attributes but while the

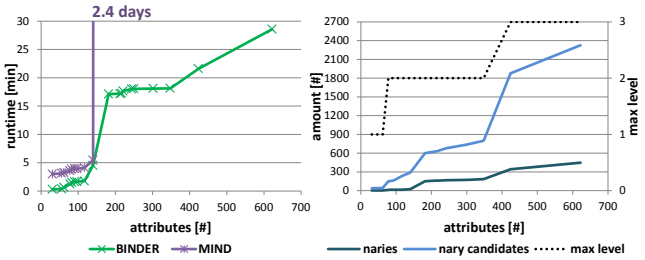

Figure 10: Runtimes of Binder and Mind scaling columns for PDB

candidate number shrinks over time using Lists, they stay large using BitSets. For these reasons, Binder uses Lists.

(3) Indexing vs. sorting. Recall that the `Validator` component checks the IND candidates using two indexes, but it could also use Spider's sort-merge join approach instead. One can imagine that the sort-based approach might be faster than using indexes, because it does not need an external memory sorting algorithm due to Binder's bucketing phase. Thus, this approach would be clearly faster than the original Spider algorithm. However, we observed in our experiments that this sort-based approach is still 2.6 times slower than the indexing approach on average.

6.6 N-ary IND discovery

We now evaluate the performance of Binder when discovering all n-ary INDs of a dataset. Due to Binder's dynamic memory management and lazy partition refinement techniques, it can find n-ary INDs using the same, efficient discovery methods as for unary INDs. As other related works do not achieve this, we compare Binder with the Mind algorithm, which is designed specifically for n-ary IND discovery. Notice that we had to conduct these experiments on database inputs only, because Mind uses SQL queries.

Our first experiment measures Binder's and Mind's runtime while scaling the number of attributes on the PDB dataset. We report the results in the left chart of Figure 10. We also show in the right chart of Figure 10 the increase of n-ary candidates and n-ary INDs in this experiment. We observe that Mind is very sensitive to the number of IND candidates and becomes inapplicable (runtime longer than two days) already when the number of candidate checks is in the order of hundreds. Binder, however, scales well with the number of candidates and in particular with the number of discovered INDs, because it efficiently reuses bucketized attributes for multiple validations. We observe that while MIND runs in 2.4 days for 150 attributes, Binder runs under 20 minutes for up to 350 attributes and under 30 minutes for 600 attributes. In contrast to Mind, we also observe that Binder is not considerably influenced by the IND's arity (max level line in the right chart of Figure 10), because it reuses data buckets whenever possible while Mind combines data values again with every SQL validation.

To see how Binder's discovery techniques perform in general, we evaluated the two algorithms on different datasets. Figure 11 depicts the measured runtimes. The experiment shows that, overall, the runtimes of Binder and Mind correlate with the size of the dataset and the number of n-ary INDs. In detail, we observe that if no (or only few) INDs are to be discovered, as in the COMA, WIKIPEDIA, and TESMA datasets, Binder's bucketing process does not pay off and single SQL queries may perform better. However, if a dataset contains n-ary INDs, which is the default in practice, Binder is orders of magnitude faster than Mind. We measured improvements by up to more than three orders of magnitude in comparison to Mind (e.g., for CENSUS). This is because Mind, on the one hand, checks each IND candidate separately, which makes

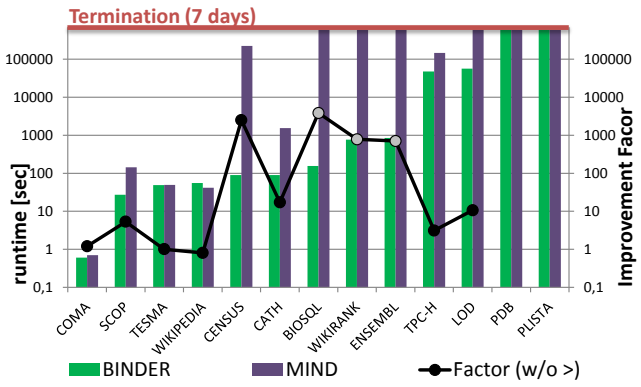

Figure 11: Comparing Binder and Mind in n-ary IND detection

it access attributes and values multiple times; Binder, on the other hand, re-uses the powerful unary candidate validation for the n-ary candidates, which lets it validate many candidates simultaneously. Considering that data access is the most expensive operation, Binder complexity in terms of I/O is $O(n)$ whereas Mind complexity is $O(2^n)$, where n is the number of attributes. Thus, even though Mind also uses pruning techniques for the IND checks, i.e., it stops matching the values of two attributes if one value is already missing, the single checks are already much too costly.

As Figure 11 shows, we had to terminate Mind after seven days for six datasets. Binder could process four of these datasets in less than a day, but also exceeds seven days for PDB and PLISTA. This is because both datasets contain an extremely high number of n-ary INDs. In PDB, we found a set of almost duplicate tables and PLISTA contains two tables, `statistics` and `requests`, with same schemata. At some point, this causes the generation of an exponential number of IND candidates. As Binder scales with the size of the result, its execution time also increases exponentially.

With the TPC-H dataset, the experiment proves that even though the number of INDs (and IND candidates) is low, Binder can still outperform Mind. This is because the SQL checks become unproportionally expensive on high numbers of attribute values. Note that we tried different validation queries based on `LEFT OUTER JOIN`, `NOT IN`, and `EXCEPT` statements (limited on the first not joinable tuple) for Mind all showing the same limitation. Especially when the tables or the IND candidates' left and right hand sides become larger, the SQL-queries become clearly slower.

In summary, our experiments showed that only if tables are small and their number of n-ary IND candidates is low, Mind can compete with Binder. But as most real world datasets are large and contain many n-ary INDs, Binder is by far the most efficient algorithm for exact and complete n-ary IND discovery.

7. RELATED WORK

Unary INDs. Bell and Brockhausen introduced an algorithm that first derives all unary IND candidates from previously collected data statistics, such as data types and min-max values. It then validates IND candidates using SQL `join`-statements [3]. Validated IND candidate are used to prune yet untested candidates. However, SQL-based validation is very costly, because it accesses the data for each IND candidate.

De Marchi et al. proposed an algorithm that transforms the data into an inverted index pointing each value to the set of all attributes containing the value [12]. One can then retrieve valid INDs from the attribute sets by intersecting them. Despite this efficient technique, the algorithm yields poor performance, because it applies the

attribute set intersections for all values, without being able to discard attributes that have already been removed from all their IND candidates. Furthermore, building such an inverted index for large datasets (i.e., not fitting in main memory) is very costly as it involves many I/O operations. BINDER solves both of these issues.

Bauckmann et al. proposed SPIDER [2], which is an adapted sort-merge-join approach. First, it individually sorts the values of each attribute, removes duplicate values, and writes these sorted lists to disk. Then it applies an adapted (for early termination) sort-merge join approach to validate IND candidates. SPIDER also prunes those attributes from the validation process that have been removed from all their IND candidates. This technique makes SPIDER the most efficient algorithm for unary IND detection in related work. However, SPIDER still comes with a large sorting overhead that cannot be reduced by its attribute pruning. And if a column does not fit into main memory, external sorting is required. Furthermore, SPIDER's scalability in the number of attributes is technically limited by most operating systems, because they limit the number of simultaneously open file handlers and SPIDER requires one per attribute.

N-ary INDs. De Marchi et al. introduced the bottom-up level-wise algorithm MIND, which uses an apriori-gen-based approach [1] for IND candidate generation [12]. Since MIND uses costly SQL-statements and validates all n-ary IND candidates individually, BINDER outperforms this algorithm applying a much more efficient validation strategy based on data partitioning.

Koeller and Rundensteiner proposed an efficient algorithm for the discovery of high-dimensional n-ary INDs [8]. The algorithm's basic assumption is that most n-ary INDs are very large and, hence, occur on high lattice levels. In practice we observed, however, that most INDs in real-world datasets are small. With their ZIGZAG algorithm [13], De Marchi and Petit also developed a technique to identify large n-ary INDs. This approach combines a pessimistic bottom-up with an optimistic depth-first search. Nevertheless, these approaches also test the IND candidates using costly SQL-statements, which does not scale well with the number of INDs. The CLIM approach [11] discusses an idea to avoid these single SQL checks by applying closed item set mining techniques on a new kind of index structure. However, the feasibility of such an algorithm is not proven.

In contrast to state-of-the-art, BINDER is the only system that utilizes the same efficient validation techniques for both unary and n-ary IND detection. BINDER's partitioning process allows it to handle the increasing value sizes with the same index-based checking method. This makes BINDER not only more efficient but also easier to implement and maintain than other approaches.

Foreign Key Discovery. The primary use case for INDs is the discovery of foreign keys. In general, this is an orthogonal task that uses the IND discovery as a preprocessing step. For instance, Rostin et al. proposed rule-based discovery techniques based on machine learning to derive foreign keys from INDs [16]. Zhang et al., in contrast, integrated the IND detection in the foreign key discovery by using approximation techniques [17]. Their specialization on foreign key discovery makes their approach inapplicable to other IND use cases, such as query optimization [6], integrity checking [5], or schema matching [9], which demand complete and not approximate results.

8. CONCLUSION & FUTURE WORK

We presented BINDER, the currently most effective system for exact and complete unary and n-ary IND discovery. It uses a divide & conquer approach, which allows it to handle very large datasets. BINDER divides a dataset into smaller partitions and applies pruning techniques to perform fast IND-checking at the partition-level. This makes our approach highly efficient. In particular, BINDER does not rely on existing database functionality nor does it assume that the dataset fits into main memory, in contrast to most related work. Our experimental results show its superiority: it is more than one order of magnitude (26 times) faster than SPIDER and up to more than three orders of magnitude (more than 2500 times) faster than MIND. The results also show that BINDER scales to much larger datasets than this state-of-the-art.

As BINDER's performance is I/O bound, we started working in a distributed version of BINDER in order to parallelize I/O operations. As future work, we aim at devising efficient techniques for the processing of discovered INDs, because the result sizes can grow exponentially with the number of attributes. These techniques must, however, be use case specific, because different use cases require different subsets of the complete result that BINDER provides.

9. REFERENCES

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, 1994.

[2] J. Bauckmann, U. Leser, and F. Naumann. Efficiently computing inclusion dependencies for schema discovery. In *ICDE Workshops*, 2006.

[3] S. Bell and P. Brockhausen. Discovery of data dependencies in relational databases. Technical report, Universität Dortmund, 1995.

[4] M. A. Casanova, R. Fagin, and C. H. Papadimitriou. Inclusion dependencies and their interaction with functional dependencies. In *PODS*, 1982.

[5] M. A. Casanova, L. Tucherman, and A. L. Furtado. Enforcing inclusion dependencies and referencial integrity. In *VLDB*, 1988.

[6] J. Gryz. Query folding with inclusion dependencies. In *ICDE*, 1998.

[7] B. Kille, F. Hopfgartner, T. Brodt, and T. Heintz. The Plista dataset. In *NRS Workshops*, 2013.

[8] A. Koeller and E. A. Rundensteiner. Discovery of high-dimensional inclusion dependencies. In *ICDE*, 2002.

[9] M. Levene and M. W. Vincent. Justification for inclusion dependency normal form. *TKDE*, 12:2000, 1999.

[10] J. Liu, J. Li, C. Liu, and Y. Chen. Discover dependencies from data - a review. *TKDE*, 24:251–264, 2012.

[11] F. D. Marchi. CLIM: Closed inclusion dependency mining in databases. In *ICDM Workshops*, 2011.

[12] F. D. Marchi, S. Lopes, and J.-M. Petit. Unary and n-ary inclusion dependency discovery in relational databases. *JIIS*, 32:53–73, 2009.

[13] F. D. Marchi and J.-M. Petit. Zigzag: A new algorithm for mining large inclusion dependencies in databases. In *ICDM*, 2003.

[14] R. J. Miller, M. A. Hernandez, L. M. Haas, L.-L. Yan, H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30:78–83, 2001.

[15] F. Naumann. Data profiling revisited. *SIGMOD Record*, 42:40–49, 2014.

[16] A. Rostin, O. Albrecht, J. Bauckmann, F. Naumann, and U. Leser. A machine learning approach to foreign key discovery. In *WebDB*, 2009.

[17] M. Zhang, M. Hadjieleftheriou, B. C. Ooi, C. M. Procopiuc, and D. Srivastava. On multi-column foreign key discovery. *VLDB Endowment*, 3:805–814, 2010.