BFASTDC: a Bitwise Algorithm for Mining Denial Constraints

Eduardo H. M. Pena[1] and Eduardo Cunha de Almeida[2]

[1] Federal University of Technology - Paraná
eduardopena@utfpr.edu.br
[2] Federal University of Paraná
eduardo@inf.ufpr.br

Abstract. Integrity constraints (ICs) are meant for many data management tasks. However, some types of ICs can express semantic rules that others ICs cannot, or vice versa. Denial constraints (DCs) are known to be a response to this expressiveness issue because they generalize important types of ICs, such as functional dependencies (FDs), conditional FDs, and check constraints. In this regard, automatic DC discovery is essential to avoid the expensive and error-prone task of manually designing DCs. FASTDC is an algorithm that serves this purpose, but it is highly sensitive to the number of records in the dataset. This paper presents BFASTDC, a bitwise version of FASTDC that uses logical operations to form the auxiliary data structures from which DCs are mined. Our experimental study shows that BFASTDC can be more than one order of magnitude faster than FASTDC.

Keywords: Denial Constraints, Data Profiling, Integrity Constraints, Data Quality

1 Introduction

Production databases often generate large and disordered datasets which become challenging to explore over time. Sometimes analysts will spend more time looking for relevant and clean data than they will do producing useful insights [1]. A research field that helps with this challenge is data profiling: the set of activities to gather statistical and structural properties, i.e, metadata, about datasets [2].

Data profiling research continually focus on developing efficient methods to discover integrity constraints (ICs) satisfied by datasets [2]. ICs validate the integrity and consistency of real-world entities that are represented in data and, although were initially devised for database schema design, are commonly used in other data management tasks, such as data integration [3] and data cleaning [4]. Well known exemplars of ICs include attribute dependencies (e.g, functional dependencies (FDs)), which express semantic relationships for data. Notice, however, that attribute dependencies may not be able to express important rules that hold in data, as shown by the examples below.

Consider an instance of relation, *employees*, as shown in Table 1. An FD could state that *(1) employees' names identify their manager*. A check constraint could

state that *(2) employees' salaries must be greater than their bonus*. Denial constraints (DCs) [5, 6] could state rules 1-2, and more expressive ones, for example, *(3) if two employees are managed by the same person, the one earning a higher salary has a higher bonus*. Thus, DCs are able to express many business rules, and subsume other types of ICs [6].

Table 1: An instance of relation *employees*.

	Name	Manager	Salary	Bonus
t_0	John	Jim	\$1000	\$300
t_1	Brad	Frank	\$1000	\$400
t_2	Jim	Mark	\$3000	\$1100
t_3	Paul	Jim	\$1200	\$400

DCs define sets of predicates that databases must satisfy to prevent attributes from taking combinations of values considered semantically inconsistent. For example, the FD (1) mentioned earlier can be defined as a sequence of (in)equality predicates: if two tuples from *employees* agree on $Name$ ($t_x.Name = t_y.Name$), then, they cannot disagree on $Managers$ ($t_x.Manager \neq t_y.Manager$). Notice that predicates of DCs are easily expressed by SQL queries and, therefore, DCs can be readily used with commercial databases.

DCs have been adopted as the IC language in various scenarios [5, 7]. Particularly, they have received considerable attention in data cleaning (violation of DCs usually indicates that data is dirty). Holoclean [7] and LLUNATIC [8] are examples of cleaning tools that use DCs. However, they assume DCs to be user-provided. Designing DCs is challenging because it requires expensive domain expertise that is not always available. Furthermore, DCs may become obsolete as business rules and data evolve. To overcome these limitations, DC-based cleaning tools (or any other DC-dependent solution) should also provide mechanisms to discover DCs holding on sample data.

Discovering DCs is nontrivial because the search space for DCs grows exponentially with the number of predicates. Predicates are defined over attributes, tuples and operators. For example, the *Salary* attribute in the relation *employees* define six predicates with the form $\{t_x.Salary\ w_o\ t_y.Salary\}$, $w_o \in \mathcal{W} : \{=, \neq, <, \leq, >, \geq\}$. Additionally, predicates are allowed to be defined over combinations of attributes and tuples. The predicate space $\mathbf{P}$ is the set of all predicates defined for a relation, and there are $2^{|\mathbf{P}|}$ DC candidates because a DC may be any subset of $\mathbf{P}$. Thus, checking DC candidates against every tuple combination of a relation instance becomes impractical [6].

Chu et al. [6] introduced important properties for DCs and presented a discovery algorithm called FASTDC. The algorithm uses the predicate space to compute sets of predicates that tuple pairs satisfy, namely, the evidence set. FASTDC then reduces the problem of discovering DCs to the problem of finding minimal covers for the evidence set. Unfortunately, a dominant computational

cost of FASTDC is computing the evidence set. The algorithm needs to test every pair of tuples of the relation instance on every predicate in **P**; therefore, its performance is highly dependent on the number of records.

In this paper, we present a new algorithm that improves DC discovery by changing how the evidence set is built. Our algorithm, BFASTDC, is a *bitwise* version of FASTDC that exploits *bit-level* operations to avoid unnecessary tuple comparisons. BFASTDC builds associations between attribute values and lists of tuple identifiers so that different combinations of these associations indicate which tuple pairs satisfy predicates. To frame evidence sets, BFASTDC operates over auxiliary bit structures that store predicate satisfaction data. This allows our algorithm to use simple logical operations (e.g., conjunctions and disjunctions) to imply the satisfaction of remaining predicates. In addition, BFASTDC can use two modifications described in [6] to discover approximate and constant DCs. These DCs variants let the discovery process to work with data containing errors (e.g., integrated data from multiple sources). In our experiments, BFASTDC produced considerable improvements on DCs discovery performance.
Organization. Section 2 discusses the related work. Section 3 reviews the definition of DCs and the DC discovery problem. Section 4 describes the BFASTDC algorithm. Section 5 presents our experimental study. Finally, Section 6 concludes this paper.

2 Related Work

Most works on IC discovery have focused on attribute dependencies. Liu et al. [9] presented a comprehensive review of the topic. Papenbrock et al. [10] have looked into implementation details, experimental evaluation, and comparison of various FD discovery algorithms.

Dependency discovery algorithms usually employ strategies to reduce the number of candidate dependencies they must check. For example, Tane [11] is an FD discovery algorithm that uses a level-wise approach to traverse the attribute-set lattice of a relation. Supersets of attributes from level $k+1$ of the lattice are pruned as Tane validates FDs from level k. FastFD [12] compares tuple pairs to build difference sets: the set of attributes in which two tuple differ. It uses depth-first search to find covers of difference sets and then derives valid FDs.

As data may be inconsistent, discovery algorithms need to, somehow, avoid returning unreliable ICs. Fan et al. [13] describe CTane and FastCFD to discovering conditional FDs, that is, FDs enforced by constants patterns. Conditional dependencies are particularly useful when working with integrated data because some dependencies may hold only on portions of the data [14]. Approximate discovery is another approach to avoid overfitting ICs [9, 15, 6]. For this matter, ICs are allowed to be approximately satisfied by a dataset. Liu et al. [9] also presented a discussion on satisfaction metrics for approximate discovery algorithms.

As opposed to dependency discovery, for which many algorithms were devised [9, 10], there are only two algorithms for discovering DCs: Hydra [16] and FASTDC [6]. Hydra can only detect exact variable DCs (DCs that is neither

approximate nor contains constant predicates). The principle of the algorithm is to avoid comparing redundant tuple pairs, i.e, tuple pairs satisfying the same predicate set. It generates preliminary DCs from a sample of tuple pairs and identifies the tuple pairs violating those DCs. Hydra then derives exact DCs from the evidence set built upon the combination of the sample and tuple pairs violating the preliminary DCs. Because Hydra eliminates the need for checking every pair of tuple, it is not able to count how many times a predicate set is satisfied by a dataset. This counting feature is precisely what enables FASTDC to discover approximate DCs. The inspiration for FASTDC comes from FastFD-FastCFD, and is twofold: pairwise comparison of tuples for extracting evidence from datasets; depth-first search for finding covers for the evidence and deriving valid ICs. As described in [6], simple modifications in FASTDC enable the algorithm to also discover DCs with constant predicates. BFASTDC is designed to avoid the exhaustive tuple pairs comparison of FASTDC, but keeping the ability to discover exact, approximate and constant DCs.

3 Background

Consider a relational database schema $\mathcal{R}$ and a set of operators $\mathcal{W} : \{=, \neq, <, \leq, >, \geq\}$. A DC [5, 6] has the form $\varphi : \forall t_x, t_y, ... \in r, \neg(P_1 \wedge ... \wedge P_m)$, where $t_x, t_y, ...$ are tuples of an instance of relation r of R, and $R \in \mathcal{R}$. A predicate P_i is a comparison atom with either the form $v_1 w_o v_2$ or $v_1 w_o c$: v_1, v_2 are variables $t_{id}.A_j$, $A_j \in R$, $id \in \{x, y, ...\}$, c is a constant from A_j's domain , and $w_o \in \mathcal{W}$.

Example 1 *We refer to the ICs (1), (2) and (3) from Section 1 to express the following DCs:* $\varphi_1 : \neg(t_x.Name = t_y.Name \wedge t_x.Manager \neq t_y.Manager), \varphi_2 : \neg(t_x.Salary < t_x.Bonus), \varphi_3 : \neg(t_x.Manager = t_y.Manager \wedge t_x.Salary > t_y.Salary \wedge t_x.Bonus < t_y.Bonus)$.

An instance of relation r satisfies a DC φ if at least one predicate of φ is false, for every pair of tuples of r. In other words, the predicates of φ cannot be all true at the same time. Following the conventions of [6], we consider there is only one relation in $\mathcal{R}$, and we limit the universal quantifier for DCs to at most two tuples, i.e, (t_x, t_y).

Table 2 shows the inverse, $\overline{w}_o$, and implication, $I(w_o)$, of the operators $w_o \in \mathcal{W}$. The inverse of a predicate $P : v_1 w_o v_2$ has the form $\overline{P} : v_1 \overline{w}_o v_2$, which is the logical complement of P. The set of predicates implied by P is $I(P) = \{P' \mid P' : v_1 w'_o v_2, \forall w'_o \in I(w_o)\}$. Every $P' \in I(P)$ is true if P is true.

Table 2: Inverse and implied operators.

w_o	$=$	$\neq$	$<$	$\leq$	$>$	$\geq$
$\overline{w}_o$	$\neq$	$=$	$\geq$	$>$	$\leq$	$<$
$I(w_o)$	$=, \leq, \geq$	$\neq$	$<, \leq, \neq$	$\leq$	$>, \geq, \neq$	$\geq$

We follow the problem definition of [6] to discover minimal DCs. A DC φ_1 on r is minimal if there does not exist a φ_2 such that both φ_1 and φ_2 are satisfied by r, and the predicates of φ_2 are a subset of φ_1. Chu et al. [6] also describe additional properties for DCs and an inference system that helps eliminating non-minimal DCs. An in-depth discussion on the theoretical aspects of DCs and other ICs can be found in [5, 17].

3.1 DC discovery

The first step to discover DCs is to set the predicate space $\mathbf{P}$ from which DCs are derived. Experts can define predicates for attributes based on the database structure. One could also use specialized tools, e.g, [18], for mining join relationships. Predicates on categorical attributes use operators $\{=, \neq\}$, and predicates on numerical attributes $\{=, \neq, <, >, \leq, \geq\}$. Figure 1 illustrates a predicate space for the relation *employees* from Section 1.

$P_1 : t_x.Name = t_y.Name$	$P_2 : t_x.Name \neq t_y.Name$	$P_3 : t_x.Name = t_x.Manager$
$P_4 : t_x.Name \neq t_x.Manager$	$P_5 : t_x.Manager = t_y.Manager$	$P_6 : t_x.Manager \neq t_y.Manager$
$P_7 : t_x.Salary = t_y.Salary$	$P_8 : t_x.Salary \neq t_y.Salary$	$P_9 : t_x.Salary < t_y.Salary$
$P_{10} : t_x.Salary \leq t_y.Salary$	$P_{11} : t_x.Salary > t_y.Salary$	$P_{12} : t_x.Salary \geq t_y.Salary$
$P_{13} : t_x.Bonus = t_y.Bonus$	$P_{14} : t_x.Bonus \neq t_y.Bonus$	$P_{15} : t_x.Bonus < t_y.Bonus$
$P_{16} : t_x.Bonus \leq t_y.Bonus$	$P_{17} : t_x.Bonus > t_y.Bonus$	$P_{18} : t_x.Bonus \geq t_y.Bonus$

Fig. 1: Example of predicate space for *employees*.

The satisfied predicate set $\mathbf{Q}_{t_\mu,t_\nu}$ of an arbitrary pair of tuples $(t_\mu, t_\nu) \in r$ is a subset $\mathbf{Q} \subset \mathbf{P}$ such that for every $P \in \mathbf{Q}$, $P(t_\mu, t_\nu)$ is true. The set of satisfied predicate sets is the evidence set $\mathbf{E}_r = \{\mathbf{Q}_{t_\mu,t_\nu} \mid \forall(t_\mu, t_\nu) \in r\}$. Different tuple pairs may return the same predicate set, hence, each $\mathbf{Q} \in \mathbf{E}_r$ is associated with an occurrence counter.

A cover for $\mathbf{E}_r$ is a set of predicates that intersects with every satisfied predicate set of $\mathbf{E}_r$, and it is minimal if none of its subsets equally intersects with $\mathbf{E}_r$. The authors of FASTDC demonstrate that minimal covers of $\mathbf{E}_r$ represent the predicates of minimal DCs [6]. Thus, the DC discovery problem becomes finding covers for evidence set $\mathbf{E}_r$.

FASTDC uses a depth-first search (DFS) strategy to find minimal covers for $\mathbf{E}_r$. Predicates of $\mathbf{P}$ are recursively arranged to form the branches of the search tree. To optimize the search, predicates that cover more elements of the evidence set are added to the path first. As minimal covers are discovered, unnecessary branches of the DFS are pruned with the inference system. Any path of the tree is a candidate cover that identifies a set of elements $\mathbf{E}_{path} \subset \mathbf{E}_r$ not yet covered. When a candidate cover includes a predicate P, elements that contain P are removed from its corresponding $\mathbf{E}_{path}$. The search stops for a branch when there are no more predicates in $\mathbf{E}_{path}$. If there are remaining elements in $\mathbf{E}_{path}$, then the related candidate cover is not minimal. The candidate cover is minimal if satisfies minimality property and $\mathbf{E}_{path}$ is empty.

The authors of FASTDC also present two modifications for their algorithm: A-FASTDC and C-FASTDC.

A-FASTDC is an algorithm for discovering approximate DCs, that is, DCs whose number of violations is bounded. The algorithm uses the same evidence set $\mathbf{E}_r$ as FASTDC, but modify the minimal cover search to work with approximation levels ϵ. In short, the search prioritizes predicates that appear in the most frequent predicate sets of $\mathbf{E}_r$. The search stops for branches of the search tree when their predicates cover frequent predicate sets. This means that the frequency of the predicate sets that were not used in the search are below a threshold $\epsilon\,|r|\,(|r|-1)$. This approximate approach is only possible because the evidence set $\mathbf{E}_r$ counts the number of times a predicate set appears in the dataset.

C-FASTDC discovers DCs with constant predicates. It builds a constant predicate space from attribute domains and then follows an Apriori approach to identify τ-frequent constant predicate sets. A constant predicate set $\mathbf{C}$ is τ-frequent if $\frac{|sup(\mathbf{C},r)|}{|r|} \geq \tau$, where $sup(\mathbf{C}, r)$ is the set of tuples of r that satisfy all predicates of $\mathbf{C}$ [6]. As τ-frequent predicate sets $\mathbf{C}$ are identified, FASTDC discovers the variable predicates holding on $sup(\mathbf{C}, r)$ and outputs DCs that are combinations of $\mathbf{C}$ and the variable predicates.

Challenge. FASTDC builds the evidence set by evaluating every predicates of the predicate space $\mathbf{P}$ on every pair of tuples of r. This computation requires $|\mathbf{P}| \times |r|^2$ predicate evaluations, of which at least half return false if we consider groups of predicates $\{P, \overline{P}, ...\}$. We next describe how BFASTDC minimizes this computational cost.

4 The BFASTDC Algorithm

BFASTDC operates *at the bit level* and takes advantage of the inversion and implication properties presented in Table 2. The computational cost of our approach grows as a function of the number of predicates that evaluate to true, and is potentially smaller than FASTDC. We next describe how to set simple data structures to represent predicate satisfaction.

4.1 Data structures

Attribute-values maps. Attribute values are organized as entries $\langle k, l \rangle$, where key k is an element of the set of values in attribute A_j, and l is a list of tuple identifiers such that $\forall id \in l$ then $t_{id}[A_j] = k$. Procedure $\textsc{Search}(A_j, k)$ finds the list l for k in A_j. $\textsc{Predecessors}(A_j, k)$ is defined for numerical attributes. It returns the set L_2 in A_j consisting of lists associated with the sequence of the next k_2 smaller than k. Notice that $\textsc{Search}(A_j, k)$ and $\textsc{Predecessors}(A_j, k)$ may return $\varnothing$ if they find no tuple identifiers associated with k. Figure 2.a depicts the assignment of tuples identifiers for *employees*. In the example, a key "**Jim**" from attribute $Name$ is inputted to $\textsc{Search}(Manager, \mathbf{Jim})$; and a key **1100** from attribute $Bonus$ is inputted to $\textsc{Predecessors}(Salary, \mathbf{1100})$.

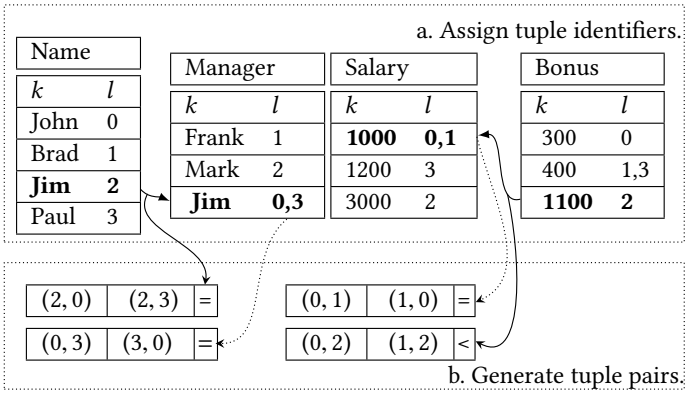


Fig. 2: Organizing attribute values: (a) assign tuple identifiers; (b) generate permutations (dashed line arrows)/Cartesian Products (solid line arrows).

Bit vectors. A bit vector B is associated with a predicate P to represent the relationship between P and the tuple pairs that satisfy P. Notice that a relation instance of size $|r|$ generates tuple pairs: $(t_0, t_0), (t_0, t_1), ..., (t_{|r|}, t_{|r|})$. Function 1 returns a unique identifier λ for a given pair of tuples (t_μ, t_ν) of r. Bit vector B holds 1 at position λ only if λ corresponds to a pair of tuples that satisfy P, otherwise B holds 0.

$$\lambda(t_\mu, t_\nu, r) = (|r|\,\mu) + \nu \tag{1}$$

Example 2 *Consider the predicate* $P_5 : t_x.Manager = t_y.Manager$ *and the relation employees from Section 1. In the sample, Predicate* P_5 *is satisfied by the following tuple pairs:* (t_0, t_3) *and* (t_3, t_0)*. From Function 1, considering the size of the instance* $|empolyees| = 4$*, with* $\lambda(t_0, t_3, employees)$ *and* $\lambda(t_3, t_0, employees)$ *we get tuple pairs identifiers* $\lambda = 3$ *and* $\lambda = 12$*. These* λ *are the indexes for which the bit vector associated with* P_5*,* B_5*, holds true.*

4.2 Building bit vectors

Before describing the strategies to efficiently obtain λ, we add some remarks regarding the possible forms of predicates.

Predicates involve one or two attributes, conventionally $\{A_1\}$ and $\{A_1, A_2\}$; and can be defined for two, (t_x, t_y), or one tuple, (t_x, t_x). We denote P_α and P_β to distinguish between two-tuple and single-tuple predicates, respectively. Let P^{w_o} be a predicate with the operator w_o, $w_o \in \mathcal{W} : \{=, \neq, <, \leq, >, \geq\}$. Hence, $P_\alpha^{w_1} : t_x.A_1 = t_y.A_1$ exemplify a two-tuple equality predicate on attribute $\{A_1\}$, $P_\beta^{w_2} : t_x.A_1 \neq t_x.A_2$ exemplify a single-tuple inequality predicate on attributes $\{A_1, A_2\}$, and so on. To ease notation for (in)equality predicates, when $o = 1$ and $o = 2$, we assume $\widehat{P}_\alpha \equiv P_\alpha^{w_1}, \widetilde{P}_\alpha \equiv P_\alpha^{w_2}$ and $\widehat{P}_\beta \equiv P_\beta^{w_1}, \widetilde{P}_\beta \equiv P_\beta^{w_2}$.

Logical operations are enough to set some of the bit vectors, but they require auxiliary bitmasks to prevent bit vectors B from holding incorrect values. Let

exponentiation denote bit repetition, e.g., $10^3 = 1000$. A bitmask $mask_{st} = (z_1, ..., z_{|r|})$, where $z_n = 10^{|r|}$, helps operations on single-tuple predicates as they are not related to pair of tuples (t_μ, t_ν) if $t_\mu \neq t_\nu$. Similarly, a bitmask $mask_{tt} = (z_1, ..., z_{|r|})$, where $z_n = 01^{|r|}$, helps operations on two-tuple predicates as they are not related to pair of tuples (t_μ, t_ν) if $t_\mu = t_\nu$.

Next, we describe four strategies that arrange the set of bit vectors $\mathbf{B}$ associated with the predicate space $\mathbf{P}$. Every $B \in \mathbf{B}$ is filled with zeros at the start.

1. Predicates involving one categorical attribute. Consider a predicate of the form $\widehat{P}_\alpha : \{t_x.A_1 = t_y.A_1\}$, and its associated bit vector $\widehat{B}_\alpha$. Given an entry $\langle k, l\rangle$ of A_1 where $|l| > 1$, permutations of two elements taken from l represent tuple pairs (t_μ, t_ν) that satisfy $\widehat{P}_\alpha$. From Function 1, these permutations generate tuple pair identifiers λ at which bit vector $\widehat{B}_\alpha$ is set to one, i.e, $\widehat{B}_{\alpha,\lambda} \leftarrow 1$. Figure 2.b illustrates some tuple pairs arranged for *employees*. For entry $\langle \mathbf{Jim}, \{0, 3\}\rangle$ from attribute *Manager*, tuple pairs $(0, 3)$ and $(3, 0)$ do satisfy a two-tuple equality predicate involving the attribute. This process repeats for every entry of A_1.

Now consider a predicate $\widetilde{P}_\alpha : \{t_x.A \neq t_y.A\}$, and its associated bit vector $\widetilde{B}_\alpha$. Observe that $\widetilde{B}_\alpha$ is the logical complement of $\widehat{B}_\alpha$. Thus, $\widetilde{B}_\alpha$ derives from a disjunction $(\vee)$ followed by an exclusive-or operation $(\oplus) : \widetilde{B}_\alpha \leftarrow (\widetilde{B}_\alpha \vee mask_{tt}) \oplus \widehat{B}_\alpha$.

2. Predicates involving two categorical attributes. Suppose that we want to find associations from attribute values of $Name$ to attribute values of $Manager$ in *employees*. Entries $\langle \mathbf{Jim}, \{2\}\rangle$ of $Name$ and $\langle \mathbf{Jim}, \{0, 3\}\rangle$ of $Manager$ generate an equality association, which is represented by the Cartesian product $\{(2, 0), (2, 3)\}$. Thus, consider an entry $\langle k_1, l_1\rangle$ taken from A_1 and a list of tuple identifiers l_2 such that $l_2 \leftarrow \textsc{Search}(A_2, k_1)$. Cartesian products $l_1 \times l_2$ represent tuple pair identifiers (t_μ, t_ν) that either satisfy a predicate $\widehat{P}_\alpha : \{t_x.A_1 = t_y.A_2\}$ or $\widehat{P}_\beta : \{t_x.A_1 = t_x.A_2\}$. Given λ corresponding to $(t_\mu, t_\nu) \in l_1 \times l_2$: if $t_\mu \neq t_\nu$ then $\widehat{B}_{\alpha,\lambda} \leftarrow 1$; otherwise, $\widehat{B}_{\beta,\lambda} \leftarrow 1$. Such process runs for every entry of A_1.

Computing $\widetilde{B}_\alpha \leftarrow (\widetilde{B}_\alpha \vee mask_{tt}) \oplus \widehat{B}_\alpha$ solves $\widetilde{P}_\alpha$. As for $\widetilde{P}_\beta$, it is sufficient to compute $\widetilde{B}_\beta \leftarrow (\widetilde{B}_\beta \vee mask_{st}) \oplus \widehat{B}_\beta$.

3. Predicates involving one numerical attribute. Numerical attributes additionally require predicates with the operators $\{<, \leq, >, \geq\}$. Given an entry $\langle k_1, l_1\rangle$ in A_1, the set L_2 such that $L_2 \leftarrow \textsc{Predecessors}(A_1, k_1)$ and lists of tuple identifiers $l_2 \in L_2$, the Cartesian product of every $l_1 \times l_2$ represent tuple pairs (t_μ, t_ν) that satisfy a predicate with the *less than* operator, $P_\alpha^{w_3}$. The tuple pair identifiers λ for which $B_\alpha^{w_3}$ holds one come from the products generated for every entry from A_1.

Bit vectors $\widehat{B}_\alpha$ and $\widetilde{B}_\alpha$ are set using permutations (strategy one). The predicates with the remaining operators are solved from $\widehat{B}_\alpha$ and $B_\alpha^{w_3}$. Predicate with *less than or equals* operator is given by: $B_\alpha^{w_4} \leftarrow (B_\alpha^{w_3} \wedge \widehat{B}_\alpha)$, with *greater than*: $B_\alpha^{w_5} \leftarrow \overline{B}_\alpha^{w_4}$, and *greater than or equals*: $B_\alpha^{w_6} \leftarrow (B_\alpha^{w_5} \wedge \widehat{B}_\alpha)$.

4. Predicates involving two numerical attributes. Bit vectors for single and two-tuple predicates $\{\widehat{B}_\alpha, \widetilde{B}_\alpha, \widehat{B}_\beta, \widetilde{B}_\beta\}$ are set using Cartesian products from attributes A_1 and A_2 (strategy two). In the same spirit, a slight modification on strategy three is sufficient to set order predicates involving two attributes. Cartesian products $l_1 \times l_2$ are generated such that $\langle k_1, l_1 \rangle$ is taken from A_1 and each $l_2 \in L_2$ is taken from PREDECESSORS(A_2, k_1). These products generate tuple pair identifiers λ that either satisfy $B_\alpha^{w_3}$ or $B_\beta^{w_3}$. The logical operations described earlier are applied on $\{\widehat{B}_\alpha, \widetilde{B}_\alpha, B_\alpha^{w_3}, \widehat{B}_\beta, \widetilde{B}_\beta, B_\beta^{w_3}\}$ to solve the remaining predicates.

4.3 Fitting bit vectors into memory

The length of a bit vector grows as a function of the relation instance size. A single bit vector would occupy $400Mb$ for a relation with $20k$ tuples. To avoid this problem and handle large relation instances, BFASTDC splits B into smaller chunks: $B = \sum_{s \in S} b_s$. The number of chunks is given by $|S| = \lceil |r|^2 / \omega \rceil$, where ω defines a maximum chunk size. The chunk size ω is related to the amount of available memory and bounds the range that chunk b_s operates.

Let b_s be a chunk being evaluated in turn s. Assume that a list of tuple pair identifiers $\Lambda = \{\lambda_1, ..., \lambda_c, ..., \lambda_{|\Lambda|}\}$, $\lambda_c < \lambda_{c+1}$, acknowledges B_{λ_c} to be true. The only portion of B in memory is b_s, so λ_c can be used to set b_{s,λ_c} only if it is in the range covered by b_s. If not, list Λ is skipped and the last λ_c used in Λ is marked. The list Λ can be iterated from λ_{c+1} in the next time it is acquired because tuple pair identifier λ_c will never be in the range of subsequent chunks b_{s+1}. Figure 3.a illustrates tuple pair identifiers on setting bit chunks. For better visualization, it considers only a subset of $\mathbf{P}$ from Figure 1.

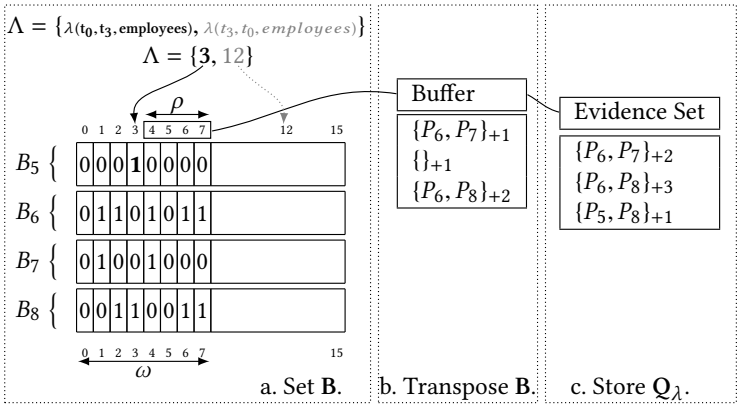


Fig. 3: Evidence set generation: (a) Fill chunks of size $\omega = 8$; (b) Transpose chunks to buffer of size $\rho = 4$; (c) Insert the buffer content into evidence set and update the predicate sets counters.

4.4 Assembling the evidence set

Each bit vector $B \in \mathbf{B}$ represents *the set of tuple pairs that satisfy predicate* P. Conversely, each element in the evidence set, $E \in \mathbf{E}_r$, is *the satisfied predicate set of a pair of tuples*. Our algorithm uses the same DFS strategy as FASTDC to search for minimal covers, hence, we need to transpose $\mathbf{B}$ into $\mathbf{E}_r$.

Let $i = 0, ..., |\mathbf{P}|$. Considering chunks of bit vectors $B_1 = \{b_{1,1}, ..., b_{1,S}\}, ..., B_{|\mathbf{P}|} = \{b_{|\mathbf{P}|,1}, ..., b_{|\mathbf{P}|,S}\}$, and $\mathbf{B} = \{B_1, ..., B_{|\mathbf{P}|}\}$, Chunks $b_{i,s}$ are transposed all at once (see Figure 3). The evidence set is built by inserting satisfied predicate sets $\mathbf{Q}_{t_\mu,t_\nu}$ into set $\mathbf{E}_r$ (see Figure 3.c). We can assume that $\mathbf{E}_r = \{\mathbf{Q}_\lambda \mid \forall \lambda \in r\}$ because λ is a unique identifier for pair of tuples $t_\mu, t_\nu \in r$. If $b_{i,s,\lambda} = 1$, then $P_j \in \mathbf{Q}_\lambda$. Notice that we only need to iterate over $b_{i,s}$ at indices λ that are set to true.

There are ω satisfied predicate sets $\mathbf{Q}$ to insert into $\mathbf{E_r}$ at each turn s. Given, $1 < \rho < \omega$, we have found that using a buffer holding ρ elements $\mathbf{Q}$ saves memory and decreases overall running time. If $b_{i,s,\lambda} = 1$, and λ is out of the buffer range, we skip iteration $b_{i,s}$ until the next round (similarly to chunks range scheme). At this stage, the predicate set counters of $\mathbf{E_r}$ are updated for further approximate discovery. Figure 3.b illustrates a buffer operation.

4.5 Implementation details

Hash-based dictionaries group entries of categorical attributes. Building them is linear since insertions on hash-based dictionaries are constant in time. Lookup operations are also performed in constant-time. We used sorted arrays to group entries of numerical attributes because they support operations $\{<, \leq, >, \geq\}$. Given a numerical entry $\langle k, l \rangle$, k and l are stored separately, into position h of two different arrays. A numerical entry is realigned by pairing both arrays with the same index h. For sorting, we have adapted the *Quicksort* algorithm to return the list of tuple identifiers for each distinct attribute value. Numerical entries are sorted according to k, which allows us to use binary search[3]. Finally, chunks and buffers are implemented as simple bitsets.

5 Experimental Study

In this section, we present our experimental study of BFASTDC. We compare BFASTDC with FASTDC to evaluate the scalability of our algorithm in the number of tuples and predicates. We also evaluate the performance of the algorithms on discovering approximate and constants DCs. Finally, we evaluate the effects that different sizes of chunks and buffers produce on the execution of BFASTDC.

[3] We have adapted binary search for procedure PREDECESSORS(A_j, k).

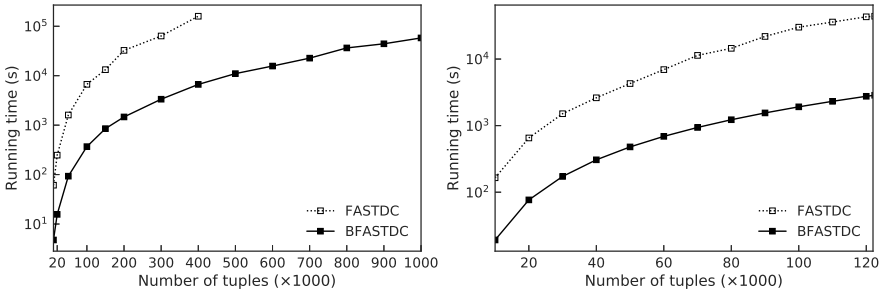
5.1 Experimental setup

Implementation and hardware. We implemented FASTDC and BFASTDC using Java programming language version 1.8. The algorithms use the same implementations of predicate space building and minimal cover search. To perform the experiments, we used a machine with a 3.4GHz Core i7, 8MB of L3 cache, 8GB of memory, running Linux. The algorithms run in main memory after dataset loading.
Datasets and predicate space. We used both synthetic and real-life datasets[4]: *Tax* and *Stock*. *Tax* is a synthetic compilation of personal information that includes fifteen attributes to represent addresses and tax-records. *Stock* gathers data from historical S&P 500 stocks in the form of a relation with seven attributes. We used *Tax* and *Stock* in our experiments because these datasets have already been used to evaluate DC discovery [6]. With regard to predicate spaces, we defined single and two-tuple predicates on: categorical attributes using operators $\{=, \neq\}$; numerical attributes using operators $\{=, \neq, <, >, \leq, \geq\}$. We defined predicates involving two different attributes provided that the values of the two attributes were in the same order of magnitude.

5.2 Results and discussion

In the first four experiments, we fixed chunk and buffer size of BFASTDC to $4000kb$ and $12kb$, respectively. These parameters are discussed in the fifth experiment. Furthermore, we report the average runtime of five runs for each experiment. Also, we consider a running time limit of 48 hours for all runs.
Exp-1: scalability in the number of tuples. We varied the number of tuples from 10,000 to 1,000,000 for *Tax*, and from 10,000 to 122,000 for *Stock*. Keeping the size of the predicate spaces constant for both datasets ($|\mathbf{P}| = 50$), we measured the running time in seconds of FASTDC and BFASTDC. Figure 4 shows their scaling behavior (Y axis are in log scale). The running time of both algorithms increases in a quadratic trend as we add more tuples in their input. However, the running time for BFASTDC were at least one order of magnitude smaller than the running time for FASTDC. To process 400,000 tuples of *Tax* (see Figure 4a), FASTDC took a little more than 2656 minutes. In contrast, BFASTDC processed the same input in approximately 110 minutes; an improvement ratio of approximately 24 times. FASTDC was not able to process more than 400,000 tuples of *Tax* within the running time limit. In turn, BFASTDC processed the entire *Tax* dataset (one million tuples) in approximately 16 hours. BFASTDC was also faster than FASTDC when running over *Stock* (see Figure 4b). It processed the full dataset in approximately 47 minutes, while FASTDC took more than 12 hours to reach completion.
Exp-2: scalability in the number of predicates. Setting the input of the algorithms to the first 20,000 tuples of *Tax* and *Stock*, we varied the number of predicates from 10 to 60. The attributes for which predicates were added to the

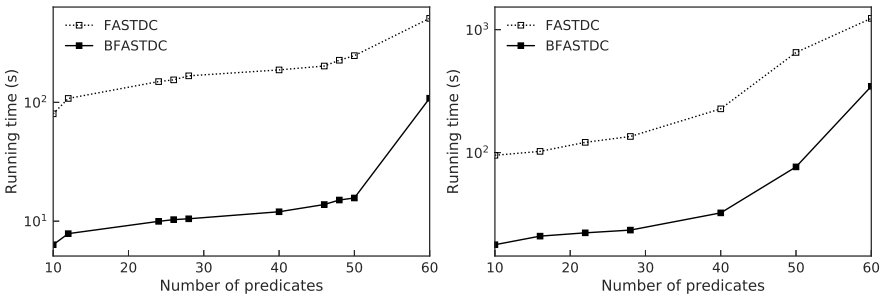
[4] Available at: `http://da.qcri.org/dc/`



(a) Tax (b) Stock

Fig. 4: Scalability of BFASTDC and FASTDC in the number of tuples.

predicate spaces were chosen at random. As shown in Figure 5 (Y axis are in log scale), the running time of the algorithms increases exponentially w.r.t. the number of predicates. In addition, the BFASTDC running time improvements over FASTDC degrades when the search for minimal covers includes larger predicate spaces.



(a) Tax (b) Stock

Fig. 5: Scalability BFASTDC and FASTDC in the number of predicates.

Exp-3: approximate DC discovery. For this experiment, we kept the number of tuples and the size of predicate space constant ($|r| = 20,000$ and $|\mathbf{P}| = 50$) for both datasets. We gradually increased the approximation levels ϵ from 10^{-6} to 2×10^{-5}. Figure 6 shows the running time for the approximate versions of BFASTDC and FASTDC (Y axis are in log scale). Despite their small improvements, the running time for both algorithms, for either *Tax* or *Stock*, remains in their original order of magnitude provided that only approximation levels differ. Indeed, varying the approximation levels did not impact on the algorithms' running time as much as varying the number of tuples or predicates did.

Exp-4: constant DC discovery. We used the same number of tuples and predicate space size as we did in experiment three. Then, we gradually increased the frequency threshold τ from 0.1 to 0.5. Figure 7 shows the running time that each algorithm took to discover constant DCs (Y axis are in log scale). The algorithms are sensitive to threshold τ. For *Tax*, smaller thresholds τ resulted in longer running times. As for *Stock*, FASTDC and BFASTDC returned within virtually the same running time. That is because there were no constant predicates to be considered by the variant portion of the algorithms.

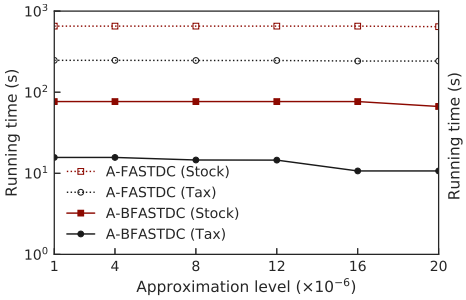


Fig. 6: Approximate DC discovery.

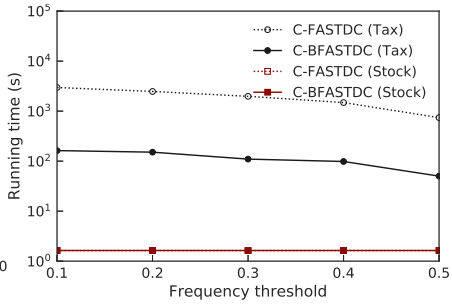


Fig. 7: Constant DC discovery.

Exp-5: BFASTDC parameters. We report this experiment using only *Tax* dataset because the same behavior and very similar parameters were seen for *Stock*. Fixing $|\mathbf{P}| = 50$, and $|r| = 100,000$, we varied chunk size ω from $250kb$ to $64,000kb$, and buffer size ρ from $5kb$ to $19kb$. Figure 8 shows that the running time does not improve as we rashly increase the size of chunks or buffers. For example, configurations where $\omega < 10000kb$ and $\rho < 14kb$ produced better results if compared to configurations with higher values. The best setting was $\omega = 4000kb$ and $\rho = 12kb$. To better understand this result, we monitored the cache activities in the evidence set building phase of BFASTDC. Table 3 shows some ratios between the monitoring of BFASTDC in its best setting and BFASTDC running in two extreme settings. The setting with bigger ω and ρ suffers from L1 cache invalidation (i.e., chunks are bigger than the cache line leading to cache misses). But, we observe an inflection point when accessing the LLC: bigger chunks need less concurrent access with less cache pollution. Therefore, we observe a sweet-spot where BFASTDC can be cache-efficient.

Table 3: Cache activities

Chunk (ω) and buffer (ρ) sizes	LLC misses	L1 misses	Running time
Baseline: $\omega = 4000kb$, $\rho = 12kb$	1	1	1
Low extreme: $\omega = 250kb$, $\rho = 5kb$	2.868	0.621	1.577
High extreme: $\omega = 64000kb$, $\rho = 19kb$	1.445	2.104	2.322

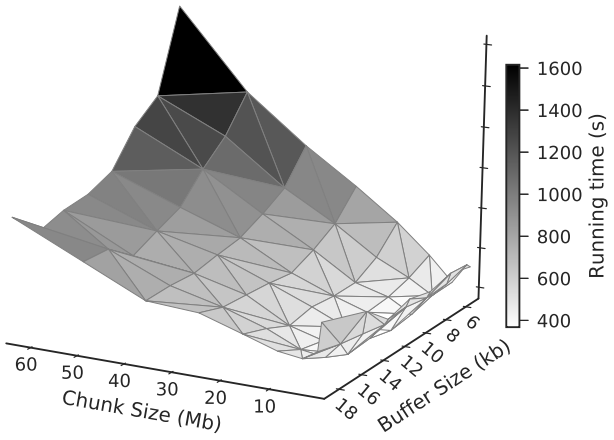


Fig. 8: Effect of different chunk/buffer sizes on running time.

Discussion. Our experiments confirm our earlier hypothesis: there is no need to check every predicate for every pair of tuples. With its attribute values organization, BFASTDC tracks bit vectors only for tuple pairs that do satisfy predicates. The bitwise representation of predicate satisfaction makes it possible to use logical operations, which are optimized in all modern CPU architectures. Such operations are cache-dependent because bit vectors are packed into processor words for processing. That is why there was an inflection point in the last experiment where the bigger the chunk and buffer sizes were, the worse the cache usage, and, therefore, the higher the running time. Experiment one demonstrates the effectiveness of BFASTDC in building the evidence set and the deep impact it had on the overall DC discovery performance. The improvements were seen in the subsequent experiments: BFASTDC was faster than FASTDC in approximate and constant DC discovery. Because of the exponential nature of the DFS used for minimal cover search, the two algorithms did not scale well with the number of predicates. Future studies could investigate not only algorithmic improvements for this phase, but how approximate discovery fits in there.

6 Conclusions

We presented BFASTDC, a bitwise, instance-driven algorithm for mining minimal DCs from relational data. BFASTDC improves the evidence set building phase of FASTDC based on two key principles: (i) it combines tuple identifiers from related values and avoids testing every pair of tuples on every predicate, and (ii) it exploits the implication relation between predicates to operate at *bit level*. BFASTDC was up to 24 times faster than FASTDC in our experimental

study. In addition, BFASTDC is able to work with noisy datasets when it is modified to discover approximate and constant DCs. For those reasons, we believe BFASTDC can be a valuable part of DC-dependent tools. Future research should improve minimal covers search and evaluate the quality of the discovered DCs on real use cases.

References

1. Kandel, S., Paepcke, A., Hellerstein, J.M., Heer, J.: Enterprise data analysis and visualization: An interview study. IEEE TVCG **18**(12) (Dec 2012)
2. Abedjan, Z., Golab, L., Naumann, F.: Profiling relational data: A survey. The VLDB Journal **24**(4) (August 2015) 557–581
3. Ayat, N., Afsarmanesh, H., Akbarinia, R., Valduriez, P.: Pay-as-you-go data integration using functional dependencies. In: Multidisciplinary Research and Practice for Information Systems. (2012) 375–389
4. Fan, W.: Data quality: From theory to practice. SIGMOD Rec. **44**(3) (December 2015) 7–18
5. Bertossi, L.: Database Repairing and Consistent Query Answering. Morgan & Claypool Publishers (2011)
6. Chu, X., Ilyas, I.F., Papotti, P.: Discovering denial constraints. Proc. VLDB Endow. **6**(13) (August 2013) 1498–1509
7. Rekatsinas, T., Chu, X., Ilyas, I.F., Ré, C.: Holoclean: Holistic data repairs with probabilistic inference. PVLDB Endow. **10**(11) (August 2017) 1190–1201
8. Geerts, F., Mecca, G., Papotti, P., Santoro, D.: That's all folks!: Llunatic goes open source. PVLDB (2014) 1565–1568
9. Liu, J., Li, J., Liu, C., Chen, Y.: Discover dependencies from data - a review. IEEE TKDE **24**(2) (February 2012) 251–264
10. Papenbrock, T., Ehrlich, J., Marten, J., Neubert, T., Rudolph, J.P., Schönberg, M., Zwiener, J., Naumann, F.: Functional dependency discovery: An experimental evaluation of seven algorithms. PVLDB. **8**(10) (June 2015) 1082–1093
11. Huhtala, Y., Kärkkäinen, J., Porkka, P., Toivonen, H.: TANE: an efficient algorithm for discovering functional and approximate dependencies. Comput. J. **42**(2) (1999) 100–111
12. Wyss, C., Giannella, C., Robertson, E.L.: Fastfds: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances - extended abstract. In: DaWaK, London, UK (2001) 101–110
13. Fan, W., Geerts, F., Li, J., Xiong, M.: Discovering conditional functional dependencies. IEEE TKDE **23**(5) (May 2011) 683–698
14. Fan, W., Geerts, F., Jia, X., Kementsietsidis, A.: Conditional functional dependencies for capturing data inconsistencies. ACM Trans. Database Syst. **33**(2) (June 2008) 6:1–6:48
15. Nakayama, H., Hoshino, A., Ito, C., Kanno, K.: Formalization and discovery of approximate conditional functional dependencies. In: Database and Expert Systems Applications - 24th International Conference, DEXA 2013, Prague, Czech Republic, August 26-29, 2013. Proceedings, Part I. (2013) 118–128
16. Bleifuß, T., Kruse, S., Naumann, F.: Efficient denial constraint discovery with hydra. Proc. VLDB Endow. **11**(3) (November 2017) 311–323
17. Fan, W., Geerts, F.: Foundations of Data Quality Management. Morgan & Claypool Publishers (2012)

18. Zhang, M., Hadjieleftheriou, M., Ooi, B.C., Procopiuc, C.M., Srivastava, D.: On multi-column foreign key discovery. PVLDB. **3**(1-2) (September 2010)