Discovery of Denial Constraints with Hardware Acceleration

SERGIO LUIZ MARQUES FILHO, Federal University of Paraná, Brazil
MARCO ANTONIO ZANATA ALVES, Federal University of Paraná, Brazil
EDUARDO CUNHA DE ALMEIDA, Federal University of Paraná, Brazil

Denial constraints (DCs) formalize integrity rules that keep data consistent across profiling and cleaning tasks. However, discovering DCs remains computationally expensive due to the exponential size of the predicate search space and the cost of maintaining large intermediate data structures. Existing software-based DC discovery algorithms rely on evidence set materialization, candidate enumeration, and minimality checks, resulting in superlinear runtime growth and unpredictable performance, which limits their scalability in practice. We present DCArray, an FPGA-based accelerator that performs DC discovery directly in hardware pipelines. DCArray encodes DC predicates as boolean patterns and evaluates them using highly parallel logical units, each operating within a single clock cycle. Boolean patterns are encoded in a hardware-friendly prefix-tree design, eliminating the need for intermediate evidence sets and reducing memory overhead and data movement. DCArray implements minimality checks at the circuit level, achieving predictable runtime independent of dataset distribution. Across real and synthetic datasets, DCArray achieves speedups of up to 1560× over DCFinder and up to 10× over ECP on 1M tuples, while efficiently utilizing HBM/PCIe. At the 10M-tuple scale, end-to-end performance is primarily limited by sustained host-device I/O bandwidth, making DCArray a practical alternative for production data-quality pipelines.

CCS Concepts: • **Information systems** → **Data profiling and discovery**; **Integrity checking**; *Data cleaning*; • **Hardware** → *Hardware accelerators.*

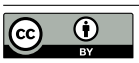
Additional Key Words and Phrases: Denial Constraints, Data Profiling, Data Quality, Constraint Discovery, FPGA Acceleration, Hardware-Accelerated Data Processing, Data Cleaning Pipelines

ACM Reference Format:
Sergio Luiz Marques Filho, Marco Antonio Zanata Alves, and Eduardo Cunha de Almeida. 2026. Discovery of Denial Constraints with Hardware Acceleration. *Proc. ACM Manag. Data* 4, 3 (SIGMOD), Article 164 (June 2026), 22 pages. https://doi.org/10.1145/3802041

1 Introduction

Data integrity rules are essential for maintaining consistent and reliable data for data-driven applications. However, maintaining data integrity becomes increasingly difficult due to inconsistencies introduced by data integration, human error, and system faults. Table 1 presents a sample from the Flights dataset, where we intentionally included integrity violations. Consider a business rule specifying that flights operated by the same aircraft (i.e., sharing the same tail number) must not be associated with different airline IDs. In Table 1, tuples t_1 and t_7 both operate aircraft N436AA but are assigned to different airlines (19805 versus 20366), violating this constraint. Another rule specifies that flights originating from the same airport must have consistent origin city names. Tuples t_2 and t_8 both depart from MCO but list different cities (OrlandoFL versus MiamiFL), violating this constraint.

Authors' Contact Information: Sergio Luiz Marques Filho, slmfilho@inf.ufpr.br, Federal University of Paraná, Curitiba, Brazil; Marco Antonio Zanata Alves, mazalves@inf.ufpr.br, Federal University of Paraná, Curitiba, Brazil; Eduardo Cunha de Almeida, eduardo@inf.ufpr.br, Federal University of Paraná, Curitiba, Brazil.



ACM 2836-6573/2026/6-ART164
https://doi.org/10.1145/3802041

	Flight Num	Origin AirportID	AirlineID	OriginCity MarketID	TailNum	Origin	Origin CityName
t_1	305	12953	19805	31703	N436AA	LGA	NewYorkNY
t_2	1527	13204	20355	31454	N163US	MCO	OrlandoFL
t_3	4352	11618	20366	31703	N15555	EWR	NewarkNJ
t_4	5358	12892	20304	32575	N295SW	LAX	LosAngelesCA
t_5	4602	12892	19393	32575	N476WN	LAX	LosAngelesCA
t_6	176	11298	19805	30194	N436AA	DFW	Dallas/FortWorthTX
t_7	305	12953	20366	31703	N436AA	LGA	NewYorkNY
t_8	1527	13204	20355	31454	N163US	MCO	MiamiFL

Table 1. A subset of the Flights dataset containing integrity violations.

These business rules can be formally expressed as denial constraints (DCs), which provide a flexible language that subsumes a variety of integrity constraint languages, including key constraints, functional dependencies, order dependencies, and conditional functional dependencies [3, 12, 23]. A DC expresses a set of predicates that cannot be true together for any tuple pair combination t_x and t_y. For example, the first business rule can be expressed as the DC: $\varphi_1 : \neg(t_x.AirlineID \neq t_y.AirlineID \wedge t_x.TailNum = t_y.TailNum)$. Similarly, the second rule becomes: $\varphi_2 : \neg(t_x.OriginCityName \neq t_y.OriginCityName \wedge t_x.Origin = t_y.Origin)$. When business rules are known in advance, they can be expressed as DCs and used to detect violations [24]. However, in many real-world scenarios, the underlying integrity constraints are unknown and must be discovered from the data itself, enabling both the identification of implicit rules and the detection of violations such as those in Table 1.

The discovery of DCs is a fundamental step in data profiling and cleaning pipelines. Manually discovering DCs is challenging because it requires domain expertise and database knowledge, and is a time-consuming, error-prone task [29]. Automatic DC discovery is more desirable. However, DC discovery algorithms face exponential search complexity due to the combinatorial explosion of predicate combinations. The DC discovery algorithms based on software rely on intermediate data structures such as evidence sets, candidate patterns, and violation lists to validate and prune constraints [1, 3, 11, 22, 23, 25, 32]. These intermediate data structures can exceed the size of the original dataset several times, resulting in high memory consumption and runtime overhead [26]. As a result, the discovery of DCs is rarely integrated into production data management pipelines [6, 27, 31], such as data exploration, integration, and cleaning systems.

Recent efforts have explored parallel software-based techniques to speed up the discovery [23, 25], but their scalability is often constrained by CPU-bound processing and memory contention between threads [18]. To overcome these limitations, we exploit hardware-accelerated architectures, such as Field Programmable Gate Arrays (FPGAs), which provide massive parallelism and high memory bandwidth, making them well-suited for scaling predicate evaluation in DC discovery.

In this paper, we approach the DC discovery problem from a hardware perspective and present DCArray, an FPGA-based accelerator for DC discovery. DCArray exploits circuit-level parallelism to evaluate constraint predicates directly in hardware. Each predicate is represented as a Boolean logic unit that operates within a single clock cycle. By encoding DCs as boolean patterns organized in a tree-like structure, DCArray avoids explicit evidence-set and candidate materialization, significantly reducing intermediate data structures and associated data movement between host memory and high-bandwidth memory (HBM). Minimality checking, a well-known computational bottleneck in DC discovery [25], is implemented at the circuit level, resulting in predictable and low-latency execution. By combining HBM with optimized data movement and parallel processing, DCArray achieves high throughput and scalable execution across the evaluated dataset sizes and attribute counts. Experimental results show that DCArray significantly reduces DC discovery time compared

to software-based algorithms, establishing FPGA acceleration as a promising alternative for large-scale data profiling pipelines. Our major contributions are as follows:

- Hardware-accelerated DC discovery: We propose DCArray, the first FPGA-based hardware architecture for the automatic discovery of exact DCs.
- Boolean predicate encoding: We design a hardware-friendly representation that encodes DC predicates as Boolean logic units organized in a prefix-tree structure. This design avoids explicit evidence sets, candidate lists, and violation tables, reducing memory usage and supporting parallel predicate evaluation.
- Predictable hardware-level minimality checks. We design and implement minimality verification directly at the circuit level, using constant-time logic operations that ensure predictable runtime behavior.
- FPGA prototype and evaluation: We implement DCArray on FPGA hardware communicating over PCIe and evaluate it using both real-world and synthetic datasets. In selected configurations, our prototype achieves speedups of more than 3 orders of magnitude compared to state-of-the-art software-based DC discovery algorithms, demonstrating the potential of hardware acceleration for data profiling and cleaning pipelines.

The remainder of this paper is organized as follows. Section 2 reviews background information about DCs and FPGA-based acceleration. Section 3 presents the DCArray architecture and predicate encoding model. Section 4 describes our experimental setup and performance evaluation of DCArray on real and synthetic datasets. Finally, Section 5 concludes the paper and suggests future work.

2 Background and Related Work

2.1 Denial Constraints

A denial constraint (DC) is a first-order logic formula used in data profiling and data cleaning to enforce integrity constraints [3, 23]. Intuitively, a DC is a conjunction of predicates that prohibits specific combinations of tuples from appearing together in a relation. Formally, a DC φ over a relation instance r is defined as $\varphi : \forall t_x, t_y \in r, \neg(p_1 \wedge ... \wedge p_n)$ where each predicate p_i is of the form $t_x.A\ \phi\ t_y.B$, with $\phi \in \{=, \neq, <, \leq, >, \geq\}$ and A, B attributes of the relation schema R. A DC φ holds in r if no distinct pair of tuples t_x, t_y simultaneously satisfies all predicates.

A valid DC satisfies four properties [3, 12]: non-triviality, symmetry, minimality, and soundness. Non-triviality ensures that a DC is not satisfied by all possible datasets, avoiding tautological forms such as $\neg(p \wedge \neg p)$. Symmetry guarantees that satisfaction is invariant under tuple swapping, i.e., if φ holds for t_x, t_y, it also holds for t_y, t_x. Minimality requires that no predicate in φ can be removed without invalidating the constraint, preventing redundant generalizations. Finally, a DC is sound if it captures genuine attribute relationships in the data, rather than holding due to coincidental or independent predicates [12].

Typically, the discovery of DCs consists of three main steps [1, 4, 11, 22, 23, 25] as follows:

1. Predicate space construction. The first step builds the search space of possible predicates from pairwise tuple comparisons. Each pair of tuples t_x, t_y serves as the processing unit in DC discovery, requiring iteration over all tuple pairs in the dataset [4]. Predicates over categorical attributes use equality and inequality operators $\{=, \neq\}$, while predicates over numerical attributes additionally employ the comparison operators $\{<, \leq, >, \geq\}$. Table 2 illustrates the resulting predicate space for the sample dataset shown in Table 1. The predicate space also includes cross-column predicates, i.e., predicates defined between different attributes $A \neq B$, of the form $p : t_x.A\ \phi\ t_y.B$. For example, predicates $p_{25} : t_x.FlightNum = t_y.OriginAirportID$ and $p_{30} : t_x.FlightNum \geq t_y.OriginAirportID$ illustrate cross-column relationships derived during this step.

$p_1 : t_x.FlightNum = t_y.FlightNum$	$p_2 : t_x.FlightNum \neq t_y.FlightNum$
$p_3 : t_x.FlightNum < t_y.FlightNum$	$p_4 : t_x.FlightNum \leq t_y.FlightNum$
$p_5 : t_x.FlightNum > t_y.FlightNum$	$p_6 : t_x.FlightNum \geq t_y.FlightNum$
$p_7 : t_x.OriginAirportID = t_y.OriginAirportID$	$p_8 : t_x.OriginAirportID \neq t_y.OriginAirportID$
$p_9 : t_x.OriginAirportID < t_y.OriginAirportID$	$p_{10} : t_x.OriginAirportID \leq t_y.OriginAirportID$
$p_{11} : t_x.OriginAirportID > t_y.OriginAirportID$	$p_{12} : t_x.OriginAirportID \geq t_y.OriginAirportID$
$p_{13} : t_x.AirlineID = t_y.AirlineID$	$p_{14} : t_x.AirlineID \neq t_y.AirlineID$
$p_{15} : t_x.AirlineID < t_y.AirlineID$	$p_{16} : t_x.AirlineID \leq t_y.AirlineID$
$p_{17} : t_x.AirlineID > t_y.AirlineID$	$p_{18} : t_x.AirlineID \geq t_y.AirlineID$
$p_{19} : t_x.OriginCityMarketID = t_y.OriginCityMarketID$	$p_{20} : t_x.OriginCityMarketID \neq t_y.OriginCityMarketID$
$p_{21} : t_x.OriginCityMarketID < t_y.OriginCityMarketID$	$p_{22} : t_x.OriginCityMarketID \leq t_y.OriginCityMarketID$
$p_{23} : t_x.OriginCityMarketID > t_y.OriginCityMarketID$	$p_{24} : t_x.OriginCityMarketID \geq t_y.OriginCityMarketID$
$p_{25} : t_x.FlightNum = t_y.OriginAirportID$	$p_{26} : t_x.FlightNum \neq t_y.OriginAirportID$
$p_{27} : t_x.FlightNum < t_y.OriginAirportID$	$p_{28} : t_x.FlightNum \leq t_y.OriginAirportID$
$p_{29} : t_x.FlightNum > t_y.OriginAirportID$	$p_{30} : t_x.FlightNum \geq t_y.OriginAirportID$
$p_{31} : t_x.TailNum = t_y.TailNum$	$p_{32} : t_x.TailNum \neq t_y.TailNum$
$p_{33} : t_x.Origin = t_y.Origin$	$p_{34} : t_x.Origin \neq t_y.Origin$
$p_{35} : t_x.OriginCityName = t_y.OriginCityName$	$p_{36} : t_x.OriginCityName \neq t_y.OriginCityName$

Table 2. A sample predicate space of the Flights dataset, showing predicates formed by comparing tuple attributes using six comparison operators ($=, \neq, <, \leq, >, \geq$). Predicates include both same-attribute comparisons (p_1-p_{24}) and cross-attribute comparisons (p_{25}-p_{30}).

2. Evidence set construction. An evidence e_{t_x,t_y} is a compact representation of the predicates satisfied by a tuple pair t_x, t_y, formally defined as $e_{t_x,t_y} = \{p \mid p \in P,\ t_x, t_y \models p\}$. The second step builds the evidence set E_r, an intermediate data structure storing all evidences for a relation instance r, defined as $E_r = \{e_{t_x,t_y} \mid \forall t_x, t_y \in r\}$. Although E_r is typically smaller than the complete set of tuple pairs [25], its construction is computationally intensive. In software-based discovery algorithms, experimental analyses show that the evidence set construction dominates response time, due to the exhaustive pairwise comparisons required [1, 23, 25].

3. DC Enumeration. DC enumeration constitutes a combinatorial problem with strong connections to Boolean algebra and set covering [8]. A cover is a subset of predicates that collectively intersect with all evidence sets, ensuring that no evidence simultaneously satisfies all predicates in a candidate DC [1, 4]. Enumerating DCs thus requires exploring all minimal predicate covers that hold over the evidence set. The computational complexity of this process grows exponentially with the number of attributes, and, by extension, with the number of possible predicates, making DC enumeration a scalability bottleneck in DC discovery.

2.1.1 Software-based DC Discovery. In software-based DC discovery, algorithms based on sets of evidence have become the standard approach [1, 3, 11, 22, 23, 25, 32]. FastDC [3] uses depth-first search to evaluate all possible DC candidates in a lattice of column combinations. FastDC showed that the problem of discovering DCs can be transformed into a problem of discovering minimal coverage sets of an evidence set. It seeks minimal coverage via evidence-set intersections. Additionally, it showed inference rules for DCs that can be used to branch-prune the search tree, reducing the search space. FastDC, however, requires significant memory to build the lattice, which may take days to process, making its approach infeasible [21].

BFastDC [22] and DCFinder [23] process chunks of evidence at a time to benefit from hardware caches. They also use indices to derive the predicate satisfactions. DCFinder uses attribute-value indexing to construct position list indexes (PLIs) and avoid exhaustive comparisons between tuple pairs. It also employs predicate selectivity to avoid the large number of logical operations required for discovery. However, these algorithms require revisiting the chunks of evidence to build the evidence set, which can be quadratic in the number of evidence.

Hydra [1] discovers exact DCs using a sampling-based approach to construct intermediate DC sets and iteratively refine them. It begins by sampling tuple pairs to form an intermediary evidence set, from which an initial set of DCs is derived. Corrections are then applied by identifying pairs of tuples that violate these DCs, enabling the determination of the complete evidence set. This method reduces the computational cost of evidence set construction. Finally, Hydra extracts the final DCs from the complete evidence set.

The Evidence Context Pipeline (ECP) [25] introduces new intermediate data representations, column indexes, and algorithms to build a parallel pipeline for generating evidence sets. Its key contribution is the concept of an evidence context, a compact representation that minimizes duplicate evidence instances. Additionally, ECP incorporates DC enumeration techniques leveraging inverted indices and pruning strategies to reduce the search space and enable parallelism.

In summary, the software-based DC discovery algorithms extract DCs from the evidence set's intermediate structures, and all of these algorithms materialize such structures in memory. Poorly optimized evidence set construction can cause significant problems, including a large memory footprint, incorrect cache utilization, poor parallelism, and other issues that hinder DC discovery.

2.2 Field Programmable Gate Array (FPGA)

The use of specialized hardware for data processing has a long history [18]. Traditional solutions rely on Application-Specific Integrated Circuits (ASICs), which deliver high performance and energy efficiency by implementing logic for specific workloads. However, ASIC design involves long development cycles and high fabrication costs, limiting its adoption outside large-scale applications.

FPGAs offer a more flexible alternative. They enable designers to implement and evaluate custom hardware architectures without the high expenses associated with ASICs. FPGAs combine fine-grained parallelism and distributed on-chip memory, enabling high-throughput, low-latency execution across a variety of data-processing workloads [7, 19].

An FPGA consists of a two-dimensional array of configurable logic cells connected through a programmable interconnect network. After manufacturing, the device is configured by loading a hardware description, typically written in a Hardware Description Language (HDL), which defines both the logic implemented in each cell and the routing of signals across the fabric.

The three basic modules found in an FPGA are: Configurable Logic Block (CLB), Configurable I/O Blocks (IOBs), and Programmable Interconnects. The CLB constitutes the basic logic component of an FPGA. CLBs are made up of two basic components: Flip-Flops used for sequential logic to store binary states and data between clock cycles, and Look-Up Tables (LUTs) used to implement combinational logic by storing a truth table. Second, the IOBs are the input/output subsystem, providing an interface between the external world and the internal architecture of the FPGA. Third, programmable interconnects are wire segments of varying lengths interconnected via electrically programmable switches. These segments transmit signals between CLBs and from CLBs to IOBs. Aside from these fundamental building blocks, FPGAs may also include clock circuits and several Intellectual Property (IP) subsystems, such as Digital Signal Processing (DSP), Arithmetic Logic Units (ALUs), decoders, and memory modules, including low-latency, high-throughput on-chip block RAM (BRAM). BRAM is typically implemented using SRAM (similar to a CPU cache) and can serve multiple purposes, such as scratch pads, FIFOs, data caches, and IO [14].

2.2.1 Predicate Evaluation in FPGAs. The predicate evaluation in the DC discovery problem differs from predicate evaluation in previous FPGA acceleration work. Methods designed to perform predicate evaluation on queries, such as [9, 30], are used to filter tuples based on the query predicates. In [30], for example, the main computational unit evaluates a single predicate performing a specific operation on specific bytes of the database row. By employing multiple PEs, all predicates in a row

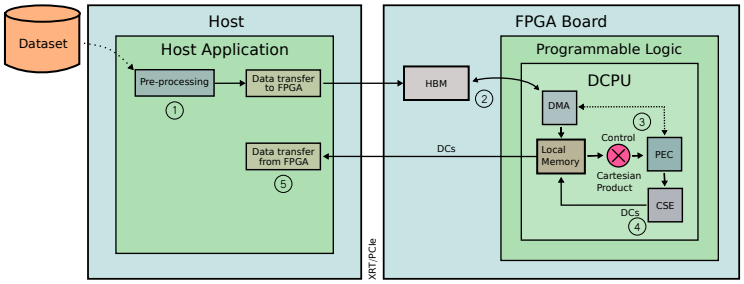


Fig. 1. The main components of the DCArray. 1. The host application preprocesses the dataset and transfers data to the FPGA board via PCIe. 2. On the FPGA, data is stored in HBM and streamed through DMA to the DCArray's local memory (BRAMs). 3. The Predicate Evaluation Component (PEC) evaluates predicates in parallel using a Cartesian product pipeline, while 4. the Cover Search Element (CSE) performs DC enumeration and minimality checking. 5. Discovered DCs are transferred back to the host.

are evaluated concurrently. In the search for DCs, however, we are interested in exploring the space of all predicates, seeking a conjunction of predicates that holds over a dataset.

Glacier [19] is a query compiler that transforms algebraic query representations into hardware circuits capable of evaluating arithmetic and boolean operators. However, Glacier recompiles the circuit for each incoming query, introducing prohibitive overhead for DC discovery, in which thousands of candidate DCs must be evaluated on a single dataset [3]. Furthermore, predicate evaluations in Glacier are performed sequentially, causing circuit latencies to accumulate and limiting throughput.

SwiftSpatial [7] accelerates spatial joins by decomposing datasets into tiles and leveraging FPGA parallelism across operators and pipeline stages to identify intersecting tile pairs. SwiftSpatial evaluates only intersections of predicates between two input objects. In contrast, DC discovery requires the simultaneous evaluation of arbitrary Boolean combinations of predicates, with each attribute in the dataset potentially serving as an input. Consequently, DC discovery demands a larger set of inputs and higher parallelism than spatial join accelerators, motivating a specialized FPGA architecture capable of fully exploiting circuit-level parallelism for predicate evaluation.

In summary, prior FPGA accelerators in data management primarily target tasks such as hash joins, deduplication, filtering, and string matching. While these FPGAs demonstrate the effectiveness of hardware acceleration for data-intensive workloads, they do not implement DC discovery, which requires exhaustive predicate evaluation, DC enumeration, and minimality checking. In particular, none of these accelerators support DC minimality enforcement or cover computation, both core components of DC discovery that incur significant computational overhead in software-based DC discovery. Consequently, we compare DCArray experimentally only against state-of-the-art software-based DC discovery algorithms that implement full enumeration and minimality, ensuring a fair and task-aligned baseline. FPGA-based systems are discussed solely to contextualize DCArray within the broader landscape of hardware-accelerated data management.

3 The DCArray Processing Unit

3.1 Overview

In this section, we present the architecture, implementation, and predicate-encoding model of DCArray. DCArray is a hardware accelerator designed to evaluate attribute values in a dataset and identify the exact DCs that hold over the dataset.

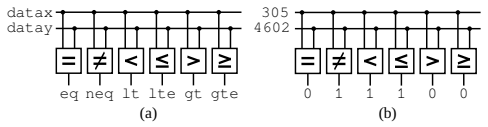


Fig. 2. (a) High-level diagram of the Logical Unit. (b) LU evaluation of predicates $p_{1...6}$ shown in Table 2, for tuples t_1 and t_5 of the Flights dataset.

DCArray achieves high performance through a combination of circuit-level parallel predicate evaluation, hardware-level minimality checking, and streaming data movement. First, it accelerates the comparison of attribute values using dedicated circuits for massively parallel processing. Second, it implements minimality checks directly in hardware to eliminate redundant DC candidates with predictable latency. Third, it employs a streaming data movement architecture that orchestrates DMA transfers and on-chip buffering.

DCArray is organized into three components:

(1) **Predicate Evaluation**, which performs Boolean predicate evaluation across tuple pairs;
(2) **Minimality**, which checks for redundant or non-minimal DC candidates;
(3) **Data Movement**, which orchestrates Direct Memory Access (DMA) transfers and on-chip buffering.

Each component is fully pipelined, with synchronization handled implicitly through hardware streaming interfaces. Figure 1 illustrates the interaction between the main components of DCArray and the FPGA board, detailed in this section.

3.2 Predicate Space Construction

In the first step of DC discovery, we compare attribute values to derive and evaluate DC predicates. A pair of tuples is the fundamental processing unit in the DC discovery, and building the predicate space requires iterating over all tuple pairs [4]. In relational terms, this component corresponds to a vectorized selection operator applied simultaneously across multiple attribute pairs, exploiting FPGA fine-grained parallelism to achieve one-cycle predicate evaluation per tuple pair.

3.2.1 Logical Unit (LU). The LU is the core component of the DCArray and is implemented within the Predicate Evaluation Component (PEC). Each LU executes a set of comparison operations $=, \neq, <, \leq, >, \geq$ over incoming data streams, as illustrated in Figure 2(a). The component receives its operands through the input ports `datax` and `datay`, which are fed by a pipelined data path, and produces results through six output ports: `eq`, `neq`, `lt`, `lte`, `gt`, and `gte`.

The LU integrates dedicated comparison circuits for each relational operator, enabling fully parallel evaluation. All output signals are generated concurrently when new input values are available. The input and output ports are implemented as buses of n parallel wires on the FPGA fabric, supporting one comparison per wire per clock cycle [19]. Each output bit encodes the truth value of its corresponding predicate, producing 1 when the comparison holds and 0 otherwise.

Figure 2(b) shows an example using the attribute FlightNum from the Flights dataset, comparing tuples t_1 and t_5 with values 305 and 4602, respectively. In this case, the LU outputs $res\_neq = 1$ for the non-equality predicate p_2, and $res\_eq = 0$ for the equality predicate p_1. This design allows the parallel evaluation of all predicates $p_{1...6}$ listed in Table 2, achieving one-cycle latency per input pair.

3.2.2 Predicate Evaluation Component (PEC). The Predicate Evaluation Component (PEC) integrates multiple Logic Units (LUs) to perform parallel comparisons over batches of tuples. Each PEC

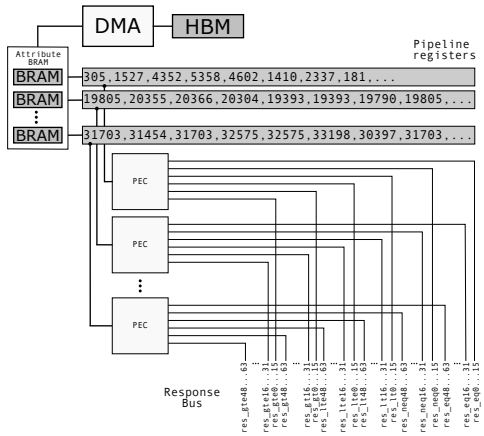


Fig. 3. DCArray BRAM, pipeline, and PEC division per attribute, enabling parallel predicate evaluation with dedicated components per attribute.

is parameterized by a configurable batch size, which defines how many values can be compared in parallel and directly impacts performance and resource utilization. Given a batch size, the number of comparisons executed per clock cycle equals the number of operators multiplied by 2^{batch_size}. The batch size is bounded by the logic and memory resources available on the target FPGA.

Attribute values are structured in pipelines and stored in local memory. These values are retrieved from FPGA dual-port Block RAMs (BRAMs) with a parameterized size of 32 KB. To ensure efficient parallel random memory access, we adopt the BRAM configuration from [20]. This setup includes a dedicated read/write port that interfaces with the DMA via the Advanced eXtensible Interface (AXI) protocol, and an additional read port for retrieving attribute values that feed the pipeline during comparison execution. We also design the DCArray to work with more than one attribute at a time to increase parallelism, as shown in Figure 3. Each attribute is assigned to an independent PEC and an exclusive BRAM instance, allowing distinct pipelines [20]. In our implementation, each attribute is allocated a 32 KB BRAM at a distinct address space.

The DMA operates in scatter/gather mode, transferring attribute data from FPGA DRAM into the corresponding BRAMs. The DCArray configures the DMA with the source address in DRAM, the destination address in BRAM, and the number of bytes to transfer. This enables automatic computation of address ranges and data movement that feed the BRAM with the attributes data. Further details on the DMA and memory hierarchy are provided in Sections 3.4.1 and 3.4.2. The outputs of all attribute PECs are concatenated into a response bus, which feeds the Cover Search Element (CSE) components (Section 3.3.1).

To illustrate the concept of batch processing adopted during the predicate evaluation, consider the attribute FlightNum from the Flights dataset, where predicates $p_{1...6}$ are evaluated. To run the Cartesian product, every value of $t_x.FlightNum$ is compared with every value of $t_y.FlightNum$. We obtain the same result by comparing batches of values, as shown in Figure 4. Figure 5 illustrates an example PEC configuration with batch size = 4, where the predicates $p_{1...6}$ are evaluated over tuples $t_1 \dots t_4$ and $t_5 \dots t_8$.

In the DCArray, the predicate evaluation cost for a dataset of n tuples is $(\frac{n}{batch\_size})^2$ FPGA clock cycles. Since attributes are processed in parallel, the evaluation time remains independent of the number of attributes.

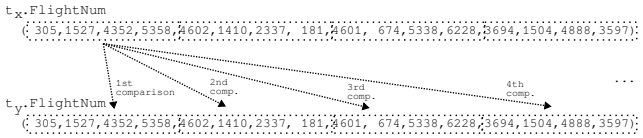


Fig. 4. Predicate evaluation dynamics using batches with batch size = 4, showing batched Cartesian product computation.

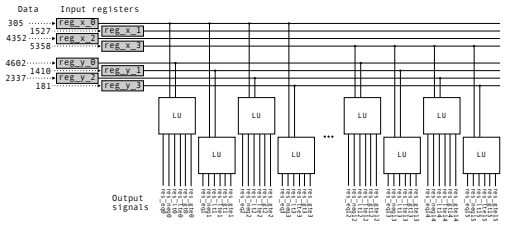


Fig. 5. Example of a PEC parameterized with a batch size of 4, showing parallel predicate evaluation across tuple pairs using LUs that generate output signals for six comparison operators.

3.3 DC Enumeration

DC enumeration is the process of generating DCs that are held in the dataset. This process handles the search for minimal DCs, which are DCs that cannot be derived from any other. Ensuring minimality eliminates redundant constraints and reduces search space [3].

We organize the predicate space as a Set Enumeration Tree (SE-Tree), illustrated in Figure 6. Each SE-Tree node encodes a combination of predicates, and the entire SE-Tree enumerates all possible predicate subsets. The SE-Tree offers two main advantages. First, the SE-tree provides a non-redundant search technique [28]. Second, it is well-suited for FPGA implementation, once it establishes a hierarchical structure of predicate combinations, in which smaller combinations are evaluated first, and their results are progressively propagated to the remaining combinations until all have been processed. For example, we use the Boolean algebra encoding $A_{=}$ to denote an equality predicate on attribute A, with other comparison operators following the same notation. Thus, a combination such as $A_{=} + B_{\neq}$ represents the disjunction of the equality predicate on attribute A and the non-equality predicate on attribute B. To optimize traversal, we apply the forward path sharing technique for predicate prefix merging, originally introduced in FPGA-accelerated airplane routing trees [13]. In this technique, combinations with fewer predicates are placed at higher levels of the SE-Tree, while those with more predicates appear at lower levels. The rest of this section describes the FPGA implementation of the SE-Tree nodes, followed by the hardware-accelerated minimality search.

3.3.1 Cover Search. Evidence is data satisfying one or more predicates. A cover is a set of predicates that overlap with evidence. While evidence set construction typically dominates the overall runtime, the cover search phase can become a bottleneck as the predicate space grows. Existing software-based DC discovery relies on materializing intermediate data structures, such as evidence sets or auxiliary indexes, to support cover search. Constructing and maintaining these structures not only incurs substantial computational overhead but also results in high memory consumption, often several times larger than the original dataset [25, 26]. For datasets with many attributes, the

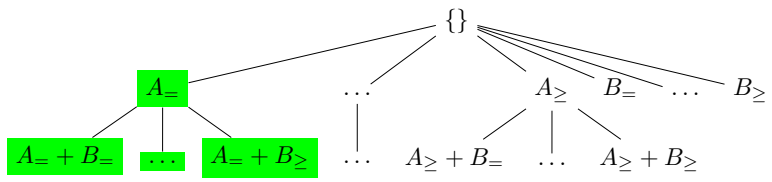


Fig. 6. An overview of the SE-Tree with the possible combinations of two attributes.

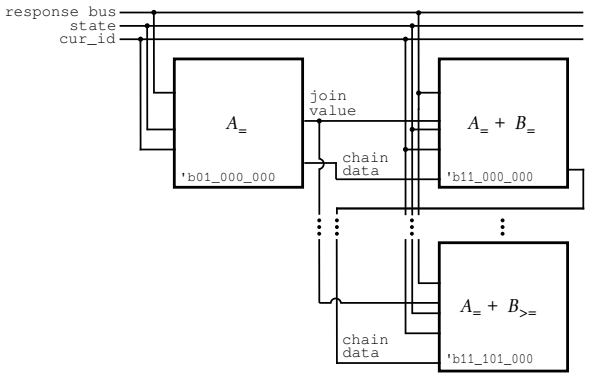


Fig. 7. High-level view of CSEs organized for the predicate branch $A_=$ of the SE-Tree (Figure 6).

time required to enumerate DCs from the evidence set increases exponentially with the number of columns [23, 25, 32].

Instead of using such intermediate data structures, the DCArray uses components named Cover Search Elements (CSEs). CSEs are responsible for performing cover verification in alignment with predicate evaluation. They work by receiving signals from the PECs' outputs and using internal registers to determine whether predicates hold.

The nodes of the SE-Tree are implemented with CSEs. Each CSE node maintains a register named `join`, responsible for keeping the comparison between the current output of the PEC component and the previous evaluations stored in the CSE's `join` register. If the evaluation holds through the entire predicate verification, the current node maintains a predicate combination that holds, indicating a valid DC. This design allows two benefits. First, the use of multiple registers eliminates the need for an intermediate evidence set. Second, the SE-Tree design allows the evaluation of all possible combinations of predicates, ensuring full coverage of the search space. For example, to evaluate the branch of predicates highlighted in Figure 6, a set of CSEs can be connected as shown in Figure 7. In this setup, parent CSEs are linked to their child CSEs, allowing reuse of predicate evaluation results. This means that the CSEs representing the conjunction of predicates, such as $A_= + B_=$ to $A_= + B_\geq$, can reuse the results generated by the node representing the predicate $A_=$.

3.3.2 Minimal cover. To test the minimal cover, we verify if a subset of the tested predicates also covers the evidence. The DCArray design is composed of two parts to ensure minimality. First, all CSEs are tied together in a chain, similar to systolic architectures, allowing data to traverse the tree from one CSE to the next [5]. Second, CSEs use their internal registers and the data that traverses through the chain to evaluate whether the predicates are minimal. The chain connecting

the CSEs is illustrated in Figure 7. It is structured as a set of n parallel wires, each parameterizable and representing a different predicate combination.

Conjunctions of predicates are represented as bitsets. The most significant bits encode which attributes are activated in the predicate, while the least significant bits encode the specific comparison operation. The subset 000 encodes the equality operator =, and the other representations follow sequentially, until 101, which encodes the less-than-or-equal operator ≤. For instance, the CSE with the predicate $A_{=}$, encoded by the bitset 01_000_000, has a predicate composed of the attribute A with operation 000. In another example, the CSE with the predicate $A_{=} + B_{\neq}$, encoded by the bitset 11_001_000, has predicates composed of the attribute A with operation 000 and the attribute B with operation 001.

To avoid collisions in the chain, the `Control` component manages a counter named `cur_id` that signals which CSE is allowed to broadcast in the chain, ensuring that only one CSE broadcasts per clock cycle whenever they have DCs that hold. All CSEs maintain distinct index numbers (`IDs`) in their internal registers. The CSE's `ID` follow the SE-tree order, which guarantees that DCs with fewer predicates are evaluated first, and, consequently, CSEs in the lower levels of the SE-tree can invalidate non-minimal DCs. When the `ID` of a particular CSE k equals `cur_id`, the CSE is allowed to broadcast along the chain when it has a DC that holds. Otherwise, the CSE verifies whether the encoded $bitset_{cur\_id} \subset bitset_k$.

For example, to verify whether $A_{=}$ is a subset of $A_{=} + B_{\neq}$, their bitset encodes are verified with 01_000_000 ⊂ 11_001_000. If this condition holds, the CSE representing the $A_{=} + B_{\neq}$ predicates is marked as non-minimal, indicating that although the CSE maintains the predicates that hold, it is not minimal and should not be broadcast. CSEs marked as not minimal continue to allow information to pass through, ensuring communication along the chain.

After the minimality is started, the Control component traverses all CSEs using one clock cycle for each CSE, allowing them to invalidate non-minimal predicate combinations. The number of CSEs is given by the equation $\sum_{i=1}^{\mathcal{A}} \left(\binom{\mathcal{A}}{i} \cdot 6^i \right)$, where $\mathcal{A}$ is the number of attributes. For instance, with two attributes ($\mathcal{A} = 2$), the chain contains 48 CSEs, yielding a minimality check latency of 48 FPGA cycles. Since DCArray processes attributes fully in parallel, the overhead of minimality verification is negligible compared to DC enumeration [25].

As we have seen, all CSEs are connected in a chain, allowing them to verify information passing through the chain and invalidate encoded predicates whenever they detect non-minimal ones. At the end of this process, only minimal predicates remain in the chain. These final results are written back to memory, completing the enumeration pipeline.

3.4 Data Movement

The Data Movement component is responsible for feeding tuples to the FPGA at line-rate throughput. Tuples are loaded from host memory using a scatter/gather DMA engine that partitions the input by attribute and streams each partition to a dedicated on-chip memory segment (BRAM). This design enables all PECs to operate independently on disjoint attribute columns, minimizing contention and ensuring a continuous data supply.

From a database-system perspective, this component serves as a tuple producer and distributor, similar to the scan and projection operators in relational pipelines. While the input may be row-oriented (e.g., CSV), our host-side engine materializes contiguous fixed-width column buffers before copying them to HBM for DMA-based streaming, as explained next.

3.4.1 Transfer descriptor chain (TDC). To provide fresh data to the DCArray, we employ a DMA engine. We developed a C++ host program using the native XRT library API to preprocess the

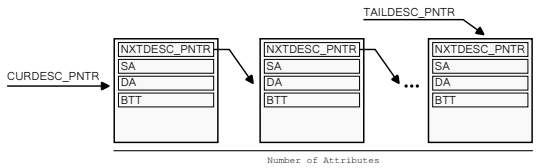


Fig. 8. Transfer descriptor chain (TDC) inside the DCArray.

TailNum

k	*l*
N436AA	0
N163US	1
N15555	2
N295SW	3
N476WN	4

Origin

k	*l*
LGA	0
MCO	1
EWR	2
LAX	3
DFW	4

CityName

k	*l*
NewYorkNY	0
OrlandoFL	1
NewarkNJ	2
LosAngelesCA	3
Dallas/FortWorthTX	4
MiamiFL	5

Fig. 9. Dictionary encoding transformation for categorical attributes, mapping variable-size strings to fixed-size integer indexes.

input dataset, store it in the FPGA's HBM, and orchestrate transfers to the on-chip BRAMs via a memory-mapped AXI interface.

The DMA enables high-bandwidth transfers between memory-mapped sources and destinations using the AXI protocol [33]. It exposes control registers (`initialization`, `status`, and `control`) through an AXI4-Lite interface. Within DCArray, the DMA operates in scatter/gather (SG) mode, allowing the `control` component to schedule transfers automatically between HBM and attribute BRAMs. In SG mode, transfers are organized into a transfer descriptor chain (TDC), which internally queues descriptors to enable prefetching and parallel transfer operations [33].

The TDC maps the dataset's attribute data stored in the HBM to the attribute BRAMs inside the DCArray. The attribute mapping is organized so that each descriptor represents the data transfer for one attribute of the dataset. Descriptors are linked via a next-pointer, forming a chain that begins at `CURDESC_PNTR` and ends at `TAILDESC_PNTR` (Figure 8).

Each descriptor in the chain contains the source address (SA) of an attribute in the HBM, the destination address (DA) of the correspondent attribute in the BRAM, and the number of bytes to transfer (BTT). The TDC itself resides in a dedicated BRAM and is fully managed by the `control` unit, which updates the list whenever new transfers are required.

Overall, for a dataset with n tuples and BRAM size $BRAM\_size$, the number of DMA transfers required is $\lceil (\frac{n}{BRAM\_size})^2 \rceil$.

3.4.2 Memory Hierarchy. The DCArray is prepared to work with categorical string attributes and numerical date, integer, and float attributes. The main issue with categorical attributes is that they commonly vary in size, making it difficult to calculate the BTT value used by the DMA. We take advantage of a property of categorical attributes, and, together with dictionary encoding [1], that categorical attributes use solely the operators $\{=, \neq\}$, allowing the replacement of variable-size strings by their fixed-size index values encoded in their corresponding dictionary. This transformation guarantees that all string values k are replaced by unique indexes l, covering all categorical attributes in the dataset, as shown in Figure 9.

The DCArray maintains one dedicated BRAM for each input attribute, enabling the PEC to process attributes in parallel. The BRAM is configured as a dual-port BRAM to enable simultaneous access to the memory space with multiple data widths. The first port is connected to the DMA component to transfer data from the board HBM to the attributes' BRAMs via the AXI protocol. The

	t_1	t_2	t_3	•••	t_n	t_{n+1}
BRAMs	B_1	B_2	B_3	•••	B_n	
PECs		B_1	B_2	B_3	•••	B_n

Fig. 10. The DCArray pipeline of batches B_1 to B_n in their respective time frames.

second port is connected directly to the `control` component, which manages data transfers using simple load operations to feed the memory pipeline discussed in Section 3.2.2 and hide memory latency.

As discussed in section 3.2.2, the PEC components operate on a batch of data of parameterizable size. Figure 10 shows the memory pipeline in which, after the first batch of data is brought from BRAM, the PECs can process batches in the same time frame t_i, thus hiding memory latency.

3.4.3 DCArray Management. The `control` component implements a state machine to manage the execution of the DC discovery, illustrated by Figure 11. The DCArray starts and remains in the `IDLE` state until it receives the *start* signal. This signal means that a dataset is available in the FPGA's HBM, and the DCArray moves to the `PROCESSING` state.

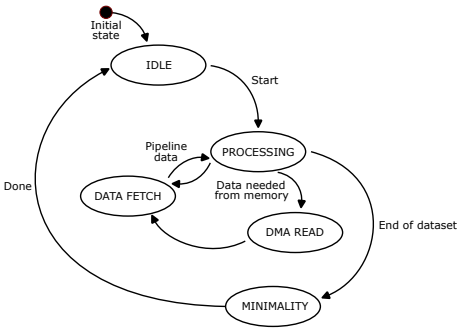


Fig. 11. The DCArray state machine, managing execution flow from IDLE through data processing (PROCESSING, DMA READ, DATA FETCH) to minimality checking (MINIMALITY).

In the `PROCESSING` state, the PECs and CSEs components consume the dataset and process the predicates. As discussed, the unit of processing in DC discovery is a tuple pair. The DCArray loads a tuple pair at the state `DMA READ`, this state triggers the setup of the TDC: the *initial* and *tail* pointers, the *source* and *destination* addresses, and the *BTT* registers. The DMA then initiates the data transfer and the DCArray and waits for an interrupt request (IRQ) signal informing that the DMA component has finished the memory transfer. After the IRQ is received, the DCArray goes to the `DATA FETCH` state. In the `DATA FETCH` state, the `Control` component feeds the pipeline with fresh data.

After all the data in the dataset has been processed by the PECs and CSEs components, the DCArray moves to the MINIMALITY state. In this state, the CSEs check the minimality of the combination of predicates as discussed in Section 3.3.2. When the CSEs have finished performing the minimality, the `control` component issues a done signal, informing that the processing of the current dataset has finished, and the DCArray returns to the `IDLE` state in which it waits for a new dataset to be processed. The DCs in the dataset remain available in the `control` component and can be read via the AXI protocol.

4 Experimental Evaluation

We conducted an experimental evaluation to answer the following research questions:

- RQ1: How does the performance of DCArray compare to state-of-the-art software-based DC discovery algorithms?
- RQ2: How does DCArray scale with respect to dataset size and number of attributes?
- RQ3: How do variations in dataset size and number of attributes impact the internal performance and resource utilization of DCArray?
- RQ4: How does DCArray compare to state-of-the-art software-based DC discovery in terms of performance-per-dollar metric?

To directly assess performance and scalability, we report both runtime and sustained throughput (MB/s) as the primary evaluation metrics in Sections 4.2 and 4.3. These metrics capture not only computation but also data transfer and minimality costs, providing a comprehensive view of system behavior. In Section 4.4, we analyze cost efficiency to quantify the economic benefits of hardware-accelerated DC discovery.

4.1 Setup

Our baseline comparison contrasts a single FPGA accelerator against multi-threaded CPU-based implementations. As discussed in Section 2.2.1, we restrict experimental baselines to software-based DC discovery systems that implement full enumeration and minimality, as existing FPGA accelerators target different data management tasks. Our experiments compared DCArray vs. three software-based DC discovery algorithms: Hydra [1], DCFinder [23], and ECP [25]. The experiments were conducted in accordance with their reproducibility guidelines, using their standard parametrization. All software-based systems guarantee that the discovered DCs are valid and minimal. However, the exact number of DCs may differ due to variations in implementation-specific pruning strategies. The software-based experiments were run on an LMDE 5 (elsie) machine equipped with a dual-socket 2× AMD EPYC 7401 24-core 2.0 GHz CPU and 64 GB of DRAM. Hydra is the only sequential algorithm among the baselines, while DCFinder and ECP are multithreaded and automatically utilize the available 48 physical cores via their built-in parallel execution, without manual tuning of thread counts.

To perform DCArray experiments, we synthesize the hardware on the publicly accessible AMD Heterogeneous Accelerated Compute Clusters (HACC) program [15–17] cluster at ETH Zürich as the evaluation platform, using an Alveo U50 accelerator card, with 8 GB High Bandwidth Memory (HBM) and PCIe Gen4 interconnect. Although the FPGA has 8 GB of HBM, DCArray does not require the entire dataset to reside on the accelerator at once. As discussed in Section 3.4, data can be streamed in batches over PCIe using the DMA engine, enabling scalability beyond on-board memory capacity. We developed a host program running on an AMD x86_64 processor (communicating over PCIe), employing the Xilinx Runtime (XRT) to interface with the programmable logic and the DCArray, which is responsible for loading data onto the FPGAs and starting the calculation kernel. We used standard FPGA and Xilinx IP configurations without custom optimization, ensuring comparable baseline conditions across all evaluated systems.

We evaluate DCArray and the software-based discovery algorithms on a mix of real-world and synthetic datasets spanning multiple scales (from 4K tuples up to full dataset sizes, plus a 10M-tuple synthetic variant). The real datasets include Flights (499,309 tuples of departure and arrival information), Hospital (114,920 tuples from a U.S. government healthcare dataset), and SPStock (122,497 tuples of historical S&P 500 stock data). We also use the synthetic Tax dataset from [2], which contains 1 million tuples, and a 10-million-tuple version obtained by replicating the 1M dataset ten times.

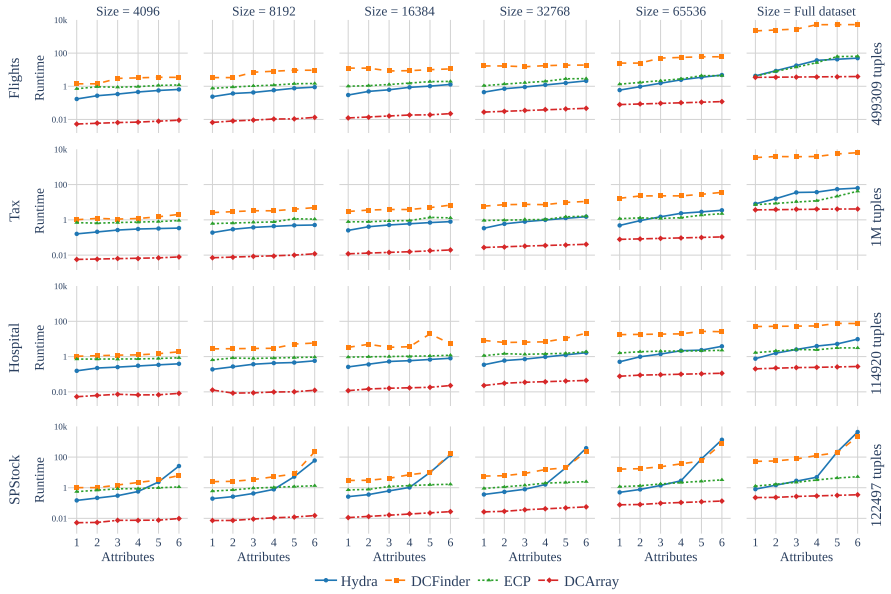


Fig. 12. End-to-end execution time (in seconds) varying dataset size and number of attributes, illustrating predictable scaling behavior.

The goal of the incremental scaling across dataset sizes is to analyze architectural behavior at small scales while demonstrating practical scalability at full dataset sizes. For each dataset, we evaluate configurations ranging from 1 to 6 attributes. We limit our evaluation to DCs involving at most 6 attributes. As shown by [12], sound DCs consistently exhibit small predicate counts across datasets and discovery algorithms, with virtually no sound constraints exceeding 6 predicates. Longer DCs predominantly correspond to unsound DCs caused by predicate agglomeration over uncorrelated attributes.

4.2 Performance Comparison

We evaluate the performance and scalability of DCArray against software-based DC discovery. The comparison spans increasing dataset sizes and attribute dimensionalities across all evaluated datasets. End-to-end execution time is reported in Figure 12, sustained throughput in Table 3, and relative speedups in Table 4. All measurements include host-to-device data transfers, FPGA-side processing, and minimality checking.

Overall Performance. DCArray consistently outperforms software-based systems across all datasets. Compared to ECP, DCFinder, and Hydra, it achieves speedups of 10×, 1560×, and 15×, respectively, on the Tax dataset with 1 million tuples with 6 attributes. These gains primarily stem from eliminating the evidence set materialization step, which can dominate the runtime in software-based discovery (see [23, 25]). While ECP and DCFinder construct full evidence sets and perform candidate pruning in memory, and Hydra partially mitigates this cost through sampling, DCArray evaluates predicates and enforces constraints directly in hardware pipelines, avoiding all intermediate materialization.

Throughput and Scalability Trends. Table 3 reports sustained throughput as the dataset size increases. DCArray throughput remains relatively stable up to the full real datasets and the 1M synthetic. However, throughput decreases at 10M due to I/O and host-device transfer effects (Table

Attrib.	Method	Flights Dataset size						Tax Dataset size						Hospital Dataset size						SP Stocks Dataset size					
		4k	8k	16k	32k	64k	~500k	4k	8k	16k	32k	64k	1M	4k	8k	16k	32k	64k	114k	4k	8k	16k	32k	64k	122k
1	Hydra	0.096	0.139	0.221	0.296	0.445	0.471	0.102	0.173	0.259	0.391	0.543	0.494	0.105	0.174	0.253	0.381	0.518	0.687	0.110	0.169	0.250	0.362	0.526	0.651
	DCFinder	0.012	0.010	0.005	0.008	0.010	0.001	0.016	0.013	0.022	0.022	0.016	0.001	0.016	0.012	0.020	0.016	0.015	0.010	0.016	0.013	0.022	0.023	0.016	0.010
	ECP	0.023	0.043	0.065	0.120	0.195	0.490	0.024	0.053	0.084	0.143	0.226	0.553	0.023	0.051	0.070	0.116	0.164	0.314	0.030	0.055	0.091	0.146	0.224	0.412
	DCArray	3.096	4.936	5.347	4.716	3.323	0.578	2.861	4.608	5.523	4.829	3.351	1.088	3.088	2.575	5.617	5.731	3.419	2.596	3.131	4.540	5.863	4.942	3.427	2.308
2	Hydra	0.122	0.178	0.267	0.369	0.550	0.475	0.156	0.224	0.321	0.442	0.580	0.504	0.146	0.241	0.366	0.442	0.536	0.670	0.151	0.250	0.367	0.492	0.679	0.710
	DCFinder	0.017	0.015	0.027	0.026	0.016	0.002	0.027	0.022	0.036	0.036	0.023	0.002	0.029	0.023	0.026	0.042	0.029	0.020	0.032	0.025	0.043	0.043	0.030	0.019
	ECP	0.035	0.073	0.120	0.190	0.311	0.501	0.050	0.101	0.169	0.275	0.404	0.933	0.046	0.079	0.134	0.179	0.277	0.495	0.048	0.092	0.168	0.236	0.392	0.632
	DCArray	5.419	8.076	9.657	8.552	6.077	1.141	5.592	8.582	9.802	9.140	6.325	2.083	5.237	7.738	8.961	8.498	5.893	4.655	6.045	8.995	9.944	9.133	6.541	4.490
3	Hydra	0.145	0.232	0.320	0.436	0.507	0.337	0.186	0.263	0.384	0.498	0.527	0.336	0.195	0.267	0.372	0.548	0.568	0.625	0.116	0.168	0.252	0.325	0.369	0.432
	DCFinder	0.016	0.014	0.024	0.026	0.016	0.002	0.044	0.030	0.051	0.052	0.034	0.003	0.041	0.034	0.059	0.060	0.042	0.030	0.030	0.025	0.037	0.034	0.029	0.017
	ECP	0.056	0.093	0.153	0.238	0.370	0.419	0.070	0.134	0.225	0.371	0.655	1.104	0.070	0.126	0.197	0.292	0.388	0.624	0.075	0.126	0.186	0.263	0.478	0.653
	DCArray	7.537	10.809	12.119	11.244	8.366	1.679	7.711	11.545	13.716	12.101	8.871	3.020	6.746	11.146	12.282	11.272	8.378	6.629	8.952	11.987	13.176	12.415	9.649	7.245
4	Hydra	0.145	0.227	0.303	0.430	0.427	0.218	0.217	0.299	0.430	0.527	0.447	0.430	0.216	0.306	0.447	0.546	0.486	0.532	0.116	0.168	0.252	0.325	0.369	0.432
	DCFinder	0.020	0.017	0.030	0.030	0.019	0.002	0.055	0.040	0.068	0.072	0.046	0.004	0.052	0.044	0.072	0.077	0.054	0.038	0.030	0.025	0.037	0.034	0.029	0.017
	ECP	0.069	0.111	0.173	0.264	0.376	0.302	0.090	0.171	0.289	0.497	0.791	1.325	0.089	0.160	0.261	0.370	0.516	0.871	0.075	0.126	0.186	0.263	0.478	0.653
	DCArray	9.522	12.394	14.202	13.259	10.235	2.198	10.296	14.979	16.770	14.955	11.170	3.941	9.850	13.404	15.488	14.127	10.449	8.560	8.952	11.987	13.176	12.415	9.649	7.245
5	Hydra	0.147	0.216	0.321	0.415	0.375	0.236	0.254	0.336	0.468	0.517	0.463	0.365	0.241	0.357	0.484	0.533	0.563	0.503	0.035	0.031	0.033	0.033	0.017	0.012
	DCFinder	0.023	0.018	0.033	0.035	0.022	0.002	0.055	0.040	0.067	0.067	0.047	0.004	0.056	0.032	0.017	0.060	0.049	0.034	0.026	0.021	0.032	0.033	0.023	0.014
	ECP	0.073	0.115	0.171	0.232	0.300	0.161	0.103	0.143	0.237	0.442	0.710	0.919	0.105	0.189	0.300	0.428	0.617	0.843	0.083	0.139	0.210	0.300	0.500	0.607
	DCArray	10.556	15.186	17.239	15.200	11.912	2.693	11.633	16.403	18.339	17.507	13.115	4.887	12.120	16.187	18.062	15.890	12.452	10.147	10.805	13.511	14.571	13.593	10.817	8.292
6	Hydra	0.152	0.221	0.305	0.371	0.330	0.241	0.289	0.386	0.496	0.522	0.451	0.380	0.253	0.342	0.490	0.475	0.412	0.324	0.004	0.003	0.003	0.002	0.001	0.001
	DCFinder	0.029	0.021	0.035	0.041	0.026	0.002	0.048	0.040	0.057	0.071	0.044	0.004	0.053	0.034	0.073	0.037	0.061	0.042	0.016	0.001	0.002	0.003	0.002	0.001
	ECP	0.082	0.134	0.201	0.271	0.360	0.187	0.108	0.179	0.304	0.486	0.702	0.573	0.115	0.211	0.330	0.419	0.690	1.018	0.088	0.146	0.235	0.317	0.485	0.603
	DCArray	10.933	14.767	17.524	16.755	13.230	3.095	12.506	16.501	20.092	19.199	14.624	5.810	11.949	15.945	17.219	17.760	13.888	11.478	10.140	12.930	14.340	14.095	11.692	9.083

Table 3. Sustained throughput (in MB/s) of DC discovery algorithms across increasing dataset sizes, from 4K tuples to the full dataset.

Attrib.	Method	Flights Dataset size						Tax Dataset size						Hospital Dataset size						SP Stocks Dataset size					
		4k	8k	16k	32k	64k	~500k	4k	8k	16k	32k	64k	1M	4k	8k	16k	32k	64k	114k	4k	8k	16k	32k	64k	122k
1	Hydra	32.4	35.6	24.2	15.9	7.5	1.2	28.0	26.6	21.3	12.4	6.2	2.2	29.3	14.8	22.2	15.0	6.6	3.8	28.4	26.9	23.4	13.7	6.5	3.5
	DCFinder	266.1	503.2	1013.1	605.0	317.3	633.2	177.0	363.2	250.8	220.6	213.8	923.8	197.1	214.9	286.1	360.1	229.0	250.4	193.6	345.5	266.0	216.6	213.4	231.0
	ECP	134.1	113.5	82.0	39.2	17.0	1.2	121.1	86.2	65.5	33.9	14.9	2.0	135.5	50.5	79.8	49.3	20.8	8.3	106.0	83.2	64.2	33.8	15.3	5.6
2	Hydra	44.6	45.3	36.2	23.2	11.0	2.4	35.9	38.2	30.5	20.7	11.0	4.1	35.9	32.2	24.5	19.2	11.0	7.0	39.9	35.9	27.1	18.6	9.6	6.3
	DCFinder	311.1	539.3	358.8	329.0	373.7	682.5	206.5	386.4	268.8	252.7	274.1	1010.8	182.2	339.1	339.9	204.0	206.5	235.8	186.7	357.8	231.5	211.7	217.9	239.9
	ECP	154.1	110.7	80.3	44.9	19.5	2.3	110.9	84.9	58.1	33.2	15.8	2.2	113.9	98.4	66.6	47.5	21.2	9.4	126.9	98.2	59.1	38.8	16.7	7.1
3	Hydra	51.9	46.7	37.9	25.8	16.5	5.0	41.4	43.8	35.7	24.3	16.8	9.0	34.7	41.7	33.0	20.6	14.7	10.6	41.1	47.1	37.6	21.9	14.6	10.5
	DCFinder	458.0	773.8	515.7	431.3	533.7	789.3	173.3	385.8	269.5	232.3	259.7	956.0	165.2	323.5	209.2	188.2	201.5	223.5	204.4	378.8	248.6	236.2	240.9	280.0
	ECP	134.4	115.8	79.0	47.3	22.6	4.0	109.9	86.0	60.8	32.6	13.5	2.7	96.4	88.8	62.5	38.5	21.6	10.6	115.2	104.6	72.5	39.9	18.2	8.4
4	Hydra	65.9	54.7	46.8	30.8	24.0	10.1	47.5	50.1	39.0	28.4	25.0	9.2	45.7	43.8	34.6	25.9	21.5	16.1	77.2	71.2	52.2	38.2	26.2	16.8
	DCFinder	468.8	748.3	474.8	444.3	550.8	1423.5	187.2	372.3	246.8	207.2	243.2	960.1	187.8	301.6	214.0	184.2	193.1	222.9	298.6	485.4	354.8	364.6	338.2	438.5
	ECP	138.8	111.3	82.2	50.2	27.2	7.3	114.2	87.7	58.0	30.1	14.1	3.0	110.6	83.7	59.3	38.1	20.2	9.8	118.7	94.8	70.7	47.2	20.2	11.1
5	Hydra	72.0	70.4	53.7	36.7	31.7	11.4	45.8	48.8	39.2	33.9	28.3	13.4	50.3	45.4	37.3	29.8	22.1	20.2	312.2	435.9	439.3	413.7	628.6	679.2
	DCFinder	450.3	863.0	526.9	436.2	546.1	1446.4	211.7	409.9	275.2	260.4	279.8	1330.7	215.0	505.5	1059.1	264.6	255.4	294.4	423.2	651.6	455.2	414.1	462.5	612.8
	ECP	145.4	132.2	100.6	65.5	39.7	16.7	113.4	114.7	77.4	39.6	18.5	5.3	115.2	85.8	60.1	37.1	20.2	12.0	130.0	97.3	69.5	45.3	21.6	13.7
6	Hydra	71.8	66.7	57.4	45.1	40.1	12.8	43.3	42.8	40.5	36.8	32.4	15.3	47.3	46.7	35.2	37.4	33.7	35.5	2684.6	3931.8	4921.3	6838.9	10024.0	12532.4
	DCFinder	382.2	696.1	498.8	404.2	506.1	1354.8	260.1	414.8	349.8	270.0	336.2	1560.6	227.6	475.4	235.0	479.0	227.6	272.9	642.2	14615.5	6066.0	4137.5	5667.4	6317.7
	ECP	133.6	110.3	87.4	61.7	36.8	16.6	116.1	92.3	66.0	39.5	20.8	10.1	103.9	75.6	52.1	42.4	20.1	11.3	115.9	88.6	61.1	44.5	24.1	15.1

Table 4. DCArray speedup over DC discovery software-based methods for data sizes from 4k tuples to the full dataset.

5). For example, with 6 attributes, DCArray sustains up to 5.81 MB/s on the 1M Tax dataset, while the other algorithms reach at most 0.6 MB/s. This behavior implies that DCArray's execution time grows approximately linearly with the number of tuples.

In software-based algorithms, throughput remains below 1 MB/s in nearly all configurations due to quadratic tuple-pair comparisons and memory contention. Hydra improves over these baselines through sampling but still suffers from declining throughput as input size scales to 1M and 10M. At the 10M scale, DCFinder did not complete within our 12-hour time bound, a limit also adopted in the DCFinder paper. Table 5 reports the completed results for Hydra, ECP, and DCArray (and marks DCFinder as a timeout).

Method	Runtime (s)	Throughput (MB/s)
Hydra	2678.01	0.090
DCFinder	timeout	-
ECP	2253.33	0.107
DCArray	363.97	0.659

Table 5. Runtime and throughput for the Tax dataset with 6 attributes and 10 million tuples.

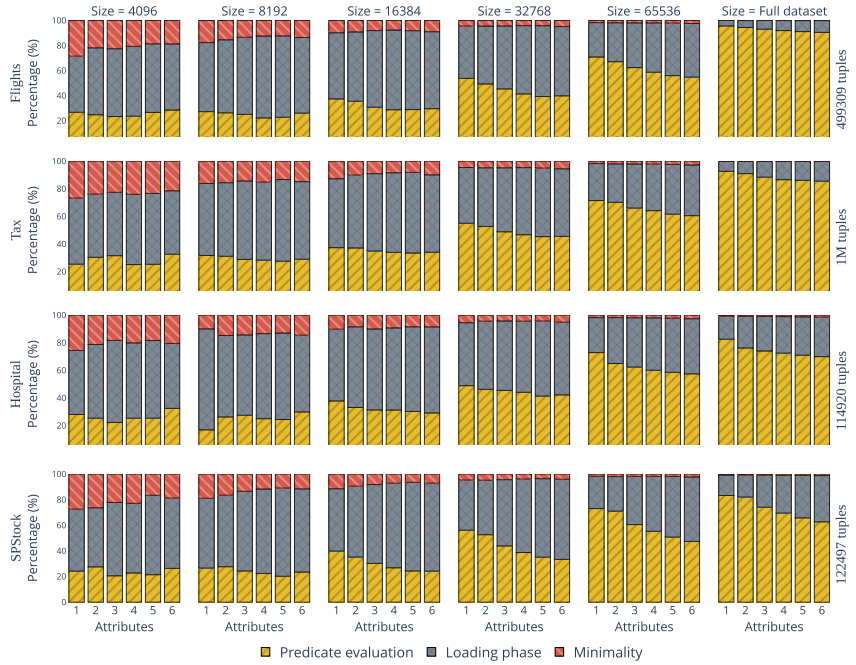


Fig. 13. End-to-end runtime breakdown of DCArray into data loading, predicate evaluation, and minimality checking phases.

Impact of Attribute Dimensionality. As the number of attributes increases from 1 to 6, we observed that the throughput delivered by DCArray increases proportionally for small and medium datasets and remains stable for full datasets. For example, throughput increases from 1.08 MB/s with 1 attribute to 5.81 MB/s with 6 attributes on the 1M Tax dataset. This trend reflects the parallel architecture of DCArray, in which each attribute is processed by a dedicated predicate evaluation component. In contrast, software-based algorithms show limited or no throughput gains with additional attributes, as higher dimensionality yields evidence set growth and increases candidate enumeration costs.

Minimality and Enumeration Steps. Software-based algorithms rely on expensive enumeration strategies, such as minimal cover search (MCS) in DCFinder, specialized algorithms in Hydra, and hybrid enumeration in ECP, whose runtime depends heavily on data distribution and thread scheduling [25], making it difficult to predict their runtime. In contrast, DCArray implements minimality directly in hardware using circuit-level checks, whose runtime depends solely on the number of CSEs. This means that the DCArray design avoids superlinear memory growth and makes execution time proportional to the number of tuples and predicates, rather than to the size of intermediate evidence. The minimality check in DCArray has a predictable runtime, as it uses a fixed number of clock cycles based on the number of tuples (i.e., CSEs). For example, when processing the full Flights dataset with 6 attributes, the minimality-checking phase accounts for 0.2% of the total end-to-end runtime (see Figure 13). This fraction decreases to 0.07% with 3 attributes and falls below 0.05% with a single attribute, indicating that minimality overhead remains bounded and diminishes as predicate dimensionality decreases. We discuss these results in more detail in the next section.

4.3 Varying Dataset Size and Number of Attributes

We decompose the end-to-end runtime of DCArray into three phases: (i) data loading from the host to HBM, (ii) FPGA-side predicate evaluation, and (iii) minimality checking. Figure 13 reports the relative contribution of these phases across increasing dataset sizes and numbers of attributes for all datasets.

Loading phase. The results show that a substantial fraction of the end-to-end runtime is spent on data loading. With 6 attributes in the full dataset, this phase accounts for 9.4% of the runtime for Flights, 14.4% for Tax, 28.6% for Hospital, and 36.1% for SPStock. This fraction remains stable as dataset sizes and configurations increase, indicating that data transfer overhead scales linearly with the number of tuples. Under the evaluated configurations, this behavior confirms that PCIe/HBM data transfer is the dominant bottleneck, rather than FPGA compute. Importantly, this data transfer cost is orthogonal to the DC discovery logic implemented in DCArray and would directly benefit from higher-bandwidth interconnects (e.g., CXL [10]), without requiring changes to the accelerator design.

Predicate evaluation. Considering the runtime spent in the processing phase, the coefficient of variation is low as the number of attributes grows. For example, to complete the DCs' evaluation of the entire Flights dataset, the DCArray runtime showed a low coefficient of variation of 4.92%, and this behavior was similar across all other datasets and numbers of tuples. The reason for such low variation stems from full parallelism: each attribute is handled by a dedicated PEC operating on an independent BRAM partition. Consequently, scaling the number of attributes increases transfer and minimality times, not compute latency, demonstrating the efficiency of DCArray's multi-attribute parallel design. However, evaluating more attributes requires transferring more data to the FPGA, which increases the time spent loading the dataset into HBM as the number of attributes increases.

The processing time showed no significant differences as the number of attributes varied, indicating that BRAM partitioning per attribute and creating one PEC component per attribute enabled parallel processing.

Minimality. Minimality is enforced by dedicated CSE units with fixed latency that depends only on the number of predicates, not on the number of tuples. As a result, the minimality phase accounts for only a small and bounded fraction of the total runtime. For example, on the full Flights dataset with six attributes, minimality represents approximately 0.2% of the end-to-end execution time. Across all datasets and configurations, this fraction remains low and decreases further as the tuple count increases, since the absolute cost of minimality remains constant while overall execution time grows linearly. Although minimality checking corresponds to an NP-complete cover problem in theory, implementing it directly in hardware with a fixed number of clock cycles per CSE ensures that its overhead scales with predicate dimensionality rather than dataset size.

Summary. Overall, DCArray exhibits predictable and stable scaling behavior as dataset sizes increase. For larger tuple sets, execution time is dominated by streaming predicate evaluation and data movement. As shown in Table 5, throughput degrades at the 10M scale due to I/O and host-device transfer overhead, while execution time remains predictable. Under this execution model, scaling to larger datasets (e.g., billion-tuple scale) is primarily a matter of sustained I/O bandwidth rather than of accelerator design limitations. Based on the validated 10M-tuple performance (363.97 seconds for DCArray, 2253.33 seconds for ECP), linear projection estimates billion-scale processing at approximately 10 hours for DCArray versus 62 hours for ECP, maintaining the observed 6.2× speedup.

4.4 Cost Efficiency Comparison

We use Amazon Web Services (AWS) on-demand pricing as a cost proxy model to assess cost efficiency under realistic deployment conditions and to ensure a fair comparison across CPU- and FPGA-based platforms. For each discovery algorithm, we compute performance-per-dollar (perf/$) as the ratio of speedup to execution cost. Execution cost is computed by multiplying the measured end-to-end runtime by the hourly price of a comparable AWS instance corresponding to the underlying hardware architecture, as specified in Section 4.1. Prices correspond to on-demand rates in AWS US East (N. Virginia) region as of January 2026, excluding storage and data transfer fees. This analysis contrasts the estimated costs of F2 FPGA-based instances (e.g., f2.6xlarge) and M-class CPU-based instances (e.g., m7g.4xlarge).

Dataset	Speedup	Relative Cost	perf/$
Flights	16.56×	6.0%	275×
Tax	10.14×	9.8%	102×
Hospital	11.26×	8.8%	127×
SP Stocks	15.05×	6.6%	227×
FPGA instance on demand price (per minute)			$ 0.033 USD
CPU instance on demand price (per minute)			$ 0.011 USD

Table 6. Performance-per-dollar (perf/$) improvement of DCArray over the best software-based method (ECP).

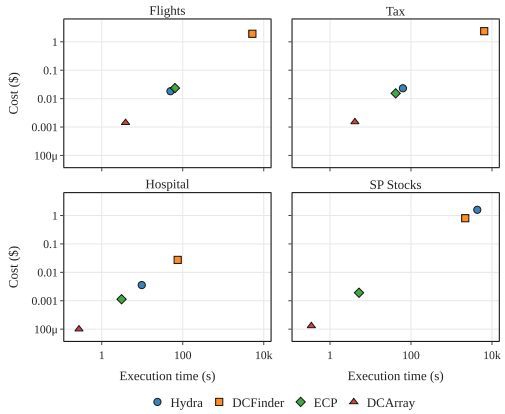


Fig. 14. Cost-effectiveness comparison of DCArray and software-based discovery, showing execution time versus incurred on-demand cost using AWS pricing.

Figure 14 reports the incurred execution costs, while Table 6 summarizes the resulting performance-per-dollar improvement of DCArray over the best-performing software baseline (ECP), providing a conservative cost comparison. Across all six-attribute datasets, DCArray consistently delivers higher absolute performance at a lower monetary cost. In particular, DCArray achieves perf/$ improvements ranging from 102× while costing only 9.8% of ECP's execution cost (Tax full dataset) to 275× while costing 6% of ECP's execution cost (Flights full dataset), showing that runtime reductions offset the higher hourly price of FPGA instances. These results indicate that DCArray is not only faster, but also more cost-effective for DC discovery workloads.

5 Conclusion

We presented DCArray, a hardware architecture for discovering DCs on FPGAs. DCArray combines predicate evaluation, minimality checking, and data movement in a pipelined, parallel execution design that mitigates the dominant bottlenecks of software-based DC discovery.

The architecture employs a scatter/gather DMA engine for on-demand data transfer, BRAM partitioning per attribute for parallel predicate evaluation, and type normalization to fixed-width formats, enabling precise memory management and predictable data movement. A latency-hiding pipeline bridges the BRAM and compute components, fully exploiting the FPGA's spatial parallelism and distributed memory resources.

Our experimental evaluation demonstrates that DCArray achieves up to three orders of magnitude speedup over software DC discovery algorithms (Hydra, DCFinder, and ECP), while maintaining stable performance across datasets with varying sizes and numbers of attributes. Detailed analysis shows that predicate evaluation dominates execution time at scale, but its cost decreases proportionally with batch size, confirming the effectiveness of the data-parallel pipeline. Additionally, scatter/gather DMA and BRAM partitioning sustain high data throughput and efficient memory utilization.

Overall, DCArray demonstrates that FPGA-based acceleration is a scalable approach to data dependency discovery, combining predictable hardware performance with substantial throughput gains. Future work includes extending DCArray to support more expressive dependency types and exploring other hardware-software co-designs for large-scale datasets that exceed on-chip memory limits. We also plan to study other hardware-friendly data structures to reduce memory consumption and processing throughput even further.

Acknowledgments

This work was supported in part by AMD under the Heterogeneous Accelerated Compute Clusters (HACC) program. We thank the HACC team at ETH Zürich for access to FPGA resources. This work was also supported by PROEX CAPES and CNPq under grants 302909/2022-2, 310753/2023-6, and 444192/2024-7. We are grateful to Alberto Lerner for valuable feedback during the early stages of this work.

References

[1] Tobias Bleifuß, Sebastian Kruse, and Felix Naumann. 2017. Efficient denial constraint discovery with hydra. *Proceedings of the VLDB Endowment* 11 (11 2017), 311–323. doi:10.14778/3157794.3157800

[2] Philip Bohannon, Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2007. Conditional Functional Dependencies for Data Cleaning. In *2007 IEEE 23rd International Conference on Data Engineering*. 746–755. doi:10.1109/ICDE.2007.367920

[3] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *Proc. VLDB Endow.* 6, 13 (2013), 1498–1509. doi:10.14778/2536258.2536262

[4] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Holistic data cleaning: Putting violations into context. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 458–469. doi:10.1109/ICDE.2013.6544847

[5] Oğuzhan Coşkun, Erkan Egeli, Erdal Tarcan, İbrahim Kurtuluş, and Güneş Yılmaz. 2023. Analysis and Implementation of a Daisy-Chain Serial Peripheral Interface Bus for a Communication Network with Multiple PLC Modules. doi:10.1109/ELECO60389.2023.10416045

[6] Stella Giannakopoulou, Manos Karpathiotakis, and Anastasia Ailamaki. 2020. Cleaning Denial Constraint Violations through Relaxation. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 805–815. doi:10.1145/3318464.3389775

[7] Wenqi Jiang, Martin Parvanov, and Gustavo Alonso. 2023. SwiftSpatial: Spatial Joins on Modern Hardware. arXiv:2309.16520 [cs.DB] https://arxiv.org/abs/2309.16520

[8] Batya Kenig and Dan Shlomo Mizrahi. 2025. Enumeration of Minimal Hitting Sets Parameterized by Treewidth. In *28th International Conference on Database Theory, ICDT 2025 (LIPIcs, Vol. 328)*, Sudeepa Roy and Ahmet Kara (Eds.).

8:1–8:20. doi:10.4230/LIPICS.ICDT.2025.8

[9] Hao Kong, Wenyan Lu, Yan Chen, Jingya Wu, Yu Zhang, Guihai Yan, and Xiaowei Li. 2023. DOE: database offloading engine for accelerating SQL processing. *Distributed Parallel Databases* 41, 3 (2023), 273–297. doi:10.1007/S10619-023-07427-Z

[10] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proc. VLDB Endow.* 17, 10 (2024), 2568–2575. doi:10.14778/3675034.3675047

[11] Ester Livshits, Alireza Heidari, Ihab F. Ilyas, and Benny Kimelfeld. 2021. Approximate Denial Constraints. *Proc. VLDB Endow.* 13, 10 (mar 2021), 1682–1695. doi:10.14778/3401960.3401966

[12] Albert Martin, Eduardo C. de Almeida, Oscar Romero, and Anna Queralt. 2025. How and Why False Denial Constraints are Discovered. *Proc. VLDB Endow.* 18, 10 (Sept. 2025), 3477–3489. doi:10.14778/3748191.3748209

[13] Fabio Maschi, Muhsen Owaida, Gustavo Alonso, Matteo Casalino, and Anthony Hock-Koon. 2020. Making Search Engines Faster by Lowering the Cost of Querying Business Rules Through FPGAs. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 2255–2270. doi:10.1145/3318464.3386133

[14] Mehdi Moghaddamfar, Christian Färber, Wolfgang Lehner, Norman May, and Akash Kumar. 2021. Resource-Efficient Database Query Processing on FPGAs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware, DaMoN 2021, 21 June 2021, Virtual Event, China*, Danica Porobic and Spyros Blanas (Eds.). ACM, 4:1–4:8. doi:10.1145/3465998.3466006

[15] Javier Moya, Matthias Gabathuler, Mario Ruiz, and Gustavo Alonso. 2023. fpgasystems/hacc: ETHZ-HACC. Zenodo. doi:10.5281/zenodo.8340448 https://doi.org/10.5281/zenodo.8340448.

[16] Javier Moya, Mario Ruiz, and Gustavo Alonso. 2023. fpgasystems/sgrt: ETHZ-SGRT. Zenodo. doi:10.5281/zenodo.8346565 https://doi.org/10.5281/zenodo.8346565.

[17] Javier Moya, Mario Ruiz, and Gustavo Alonso. 2024. fpgasystems/hdev: HACC Development. Zenodo. doi:10.5281/zenodo.14202998 https://doi.org/10.5281/zenodo.14202998.

[18] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Data processing on FPGAs. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 910–921. doi:10.14778/1687627.1687730

[19] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 229–240. doi:10.14778/1687627.1687654

[20] Muhsen Owaida, Gustavo Alonso, Laura Fogliarini, Anthony Hock-Koon, and Pierre-Etienne Melet. 2019. Lowering the Latency of Data Processing Pipelines Through FPGA based Hardware Acceleration. *Proc. VLDB Endow.* 13, 1 (2019), 71–85. doi:10.14778/3357377.3357383

[21] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. 2015. Functional Dependency Discovery: An Experimental Evaluation of Seven Algorithms. *Proc. VLDB Endow.* 8, 10 (jun 2015), 1082–1093. doi:10.14778/2794367.2794377

[22] Eduardo H. M. Pena and Eduardo Cunha de Almeida. 2018. BFASTDC: A Bitwise Algorithm for Mining Denial Constraints. In *Database and Expert Systems Applications - 29th International Conference, DEXA 2018, Regensburg, Germany, September 3-6, 2018, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 11029).* 53–68. doi:10.1007/978-3-319-98809-2_4

[23] Eduardo H. M. Pena, Eduardo C. de Almeida, and Felix Naumann. 2019. Discovery of Approximate (and Exact) Denial Constraints. *Proc. VLDB Endow.* 13, 3 (nov 2019), 266–278. doi:10.14778/3368289.3368293

[24] Eduardo H. M. Pena, Eduardo Cunha de Almeida, and Felix Naumann. 2021. Fast Detection of Denial Constraint Violations. *Proc. VLDB Endow.* 15, 4 (2021), 859–871. doi:10.14778/3503585.3503595

[25] Eduardo H. M. Pena, Fabio Porto, and Felix Naumann. 2022. Fast Algorithms for Denial Constraint Discovery. *Proc. VLDB Endow.* 16, 4 (dec 2022), 684–696. doi:10.14778/3574245.3574254

[26] Chaoqin Qian, Menglu Li, Zijing Tan, Ai Ran, and Shuai Ma. 2023. Incremental discovery of denial constraints. *VLDB J.* 32, 6 (2023), 1289–1313. doi:10.1007/S00778-023-00788-Y

[27] Theodoros Rekatsinas, Xu Chu, Ihab F. Ilyas, and Christopher Ré. 2017. HoloClean: Holistic Data Repairs with Probabilistic Inference. *Proc. VLDB Endow.* 10, 11 (2017), 1190–1201. doi:10.14778/3137628.3137631

[28] Ron Rymon. 1992. Search through Systematic Set Enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning* (Cambridge, MA) *(KR'92)*. 539–550.

[29] Shaoxu Song, Fei Gao, Ruihong Huang, and Chaokun Wang. 2020. Data Dependencies over Big Data: A Family Tree. *IEEE Transactions on Knowledge and Data Engineering* 34, 10 (2020), 1–1. doi:10.1109/TKDE.2020.3046443

[30] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database analytics acceleration using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques* (Minneapolis, Minnesota, USA) *(PACT '12)*. 411–420. doi:10.1145/2370816.2370874

[31] Nicolas Tamalu, Leandro Augusto Ensina, Eduardo Cunha de Almeida, Eduardo Henrique Monteiro Pena, and Luiz Eduardo Soares de Oliveira. 2023. Fault Detection in Transmission Lines: a Denial Constraint Approach. In *Proceedings of the 38th Brazilian Symposium on Databases, SBBD 2023, Belo Horizonte, Brazil*. SBC, 231–243. doi:10.5753/SBBD.2023.231718

[32] Renjie Xiao, Zijing Tan, Haojin Wang, and Shuai Ma. 2022. Fast Approximate Denial Constraint Discovery. *Proc. VLDB Endow.* 16, 2 (nov 2022), 269–281. doi:10.14778/3565816.3565828

[33] Xilinx 2018. *AXI Central Direct Memory Access v4.1*. Xilinx. Available at https://docs.amd.com/r/en-US/pg034-axi-cdma.

Received October 2025; revised January 2026; accepted February 2026