Storage-Centric Relation Design via High-Quality Approximate Functional Dependencies

Rui Ding
Northeastern University
Shenyang, China
ruiding.neu@outlook.com

Xiaochun Yang
Northeastern University
Shenyang, China
yangxc@mail.neu.edu.cn

Bin Wang
Northeastern University
Shenyang, China
binwang@mail.neu.edu.cn

Quanqing Xu
OceanBase
Hangzhou, China
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase
Hangzhou, China
rizhao.ych@oceanbase.com

ABSTRACT

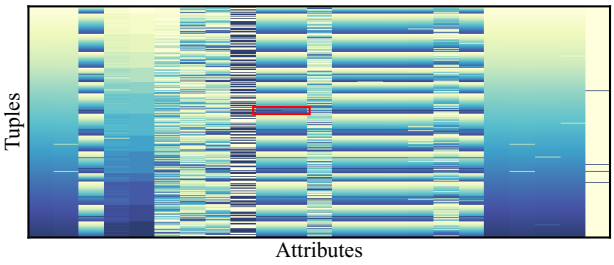
As storage costs continue to rise, reducing redundancy has become increasingly important. In relational databases, classical normalization addresses redundancy through exact functional dependencies (FDs), but this rule-based design paradigm is not inherently cost-aware and does not necessarily minimize storage in practice. Moreover, much real-world redundancy follows FD+Δ patterns, where FDs hold for most tuples but are violated by a small fraction. To address this, we propose RelaxRD, a storage-centric relaxed schema design that leverages approximate functional dependencies (AFDs) to reduce redundancy in FD+Δ. Rather than treating all AFDs as equally useful signals, we quantify the storage value of AFD subsets via duplicate gain and select a high-quality subset for decomposition. It decomposes tuples satisfying the selected AFDs while retaining violating tuples. The key issue is that selecting a high-quality subset is difficult due to conflicts and the exponential search space. To tackle this, we develop a family of efficient filtering techniques to eliminate low-value and unpromising candidates without exhaustive enumeration. Extensive experiments on real-world datasets demonstrate that RelaxRD consistently achieves substantial storage savings.

PVLDB Reference Format:
Rui Ding, Xiaochun Yang, Bin Wang, Quanqing Xu, and Chuanhui Yang. Storage-Centric Relation Design via High-Quality Approximate Functional Dependencies. PVLDB, 19(9): 2385 - 2397, 2026.
doi:10.14778/3819518.3819558

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/dinry/RelaxRD.

1 INTRODUCTION

Recent market reports have indicated sustained increases in NAND flash and SSD prices [19, 42]. Coupled with the explosive growth of data, storage has become a primary bottleneck. In relational databases, normalization (e.g., 3NF and BCNF) reduces redundancy by decomposing relations using exact functional dependencies (FDs), improving logical consistency and avoiding anomalies [44, 49]. However, in schema design, logical objectives are often misaligned with physical storage reduction. Logical schema design follows a rule-based paradigm: given an FD set, it checks whether a schema satisfies normal-form constraints and decomposes it if not, potentially yielding multiple valid decompositions without indicating which one is more storage-efficient [49]. In contrast, physical schema design is inherently cost-driven, focusing on which design reduces cost. While compression techniques [24] can reduce storage, they operate on the physical level without schema structure and thus cannot eliminate logical redundancy or improve consistency maintenance. Ideally, logical schema design should distinguish better decompositions from merely valid ones with respect to downstream physical storage, rather than treating all valid decompositions as equally acceptable. This allows logical design to explicitly incorporate physical objectives.



(a) Duplicates marked with the same color in each column.

	UID	City	PayType	ItemDetail	Merchant	Category	Amt	Tmstp
t_0	U1	SH	Huabei	ID1	Luckin	Latte	23.00	09:12
t_1	U1	SH	Huabei	**ID2**	Luckin	Flat White	**28.00**	09:35
t_2	U1	SH	Huabei	ID3	Mixue	Milk Tea	32.00	10:03
t_3	U1	SH	Huabei	ID4	Luckin	Flat White	32.00	10:27
t_4	**U2**	**SZ**	Yu'E Bao	ID3	Mixue	Milk Tea	32.00	11:11
t_5	**U2**	**SZ**	Yu'E Bao	ID4	Luckin	Flat White	32.00	11:56
t_6	**U2**	**HZ**	Yu'E Bao	**ID2**	Luckin	Flat White	**25.00**	12:22
t_7	**U3**	SZ	**Credit**	ID1	Luckin	Latte	23.00	12:22
t_8	**U3**	SZ	**Credit**	ID3	Mixue	Milk Tea	32.00	12:59
t_9	**U3**	SZ	**Yu'E Bao**	**ID2**	Luckin	Flat White	**28.00**	13:48

(b) Toy relation r sampled from payment-log relation.

Figure 1: A real-world payment-log relation.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819558

This misalignment manifests in two important ways. First, even exact FDs may be physically unprofitable for decomposition due to limited duplicates, repeated keys, or schema overhead. Worse still, the vast majority of duplicates are not captured by exact FDs. Instead, they follow an FD+Δ pattern, where most tuples conform to an FD, while a small fraction (Δ) violates it due to noise or semantic relaxations. Figure 1(a) visualizes a real-world payment-log relation, where columns correspond to attributes and rows correspond to tuples. It reveals many FD+Δ duplicate patterns at a coarse level. In particular, the highlighted rows share the same value on the left-side attribute(s), but differ on the right-side attribute(s), showing that the same determinant value may be associated with different dependent values. Further, Figure 1(b) provides a concrete zoom-in view of this phenomenon: *UID* → *City* holds for most tuples, but a few tuples violate, e.g., when *UID* = *U2*, *City* may be *SZ* or *HZ*.

Motivated by these, we study relaxed relation design (RelaxRD), a storage-centric extension of classical normalization. RelaxRD constructs lossless schemas guided by dependencies that are meaningful for redundancy reduction but are not required to hold for every tuple. An intuitive way to realize RelaxRD is to use approximate functional dependencies (AFDs) [2, 9] as candidate design signals, quantify their storage value, and select those that induce the most storage-efficient logical schema, where each determinant (Left-hand Side, LHS) specifies a schema that stores the attributes it functionally determines (Right-hand Side, RHS). Such AFDs can be obtained from existing discovery techniques or provided by domain experts (e.g., DBAs) [8, 17, 28, 32]. Database instances are then interpreted under this schema: tuples that satisfy all AFDs are stored in the corresponding normalized relations, whereas violating tuples remain in a base relation without being decomposed.

Achieving this goal raises several challenges: (1) Given an AFD set, how can we maximize the number of decomposable tuples? RelaxRD must determine which tuples to decompose and which to retain. This is challenging because tuple-level decomposability under different AFDs may conflict, and maximizing decomposition for one AFD can reduce feasible choices for others. (2) How to select a good AFD subset to minimize storage? A relation may admit many AFD candidates, but different AFD subsets may induce different storage costs, and "more" is not necessarily better due to conflicts among tuples. For instance, 30 AFDs are discovered in toy relation r of Figure 1(b): the subset in Figure 2(b) yields 65 cells, compared to 77 cells for the subset in Figure 2(a).

We developed RelaxRD to address these challenges. To address Challenge (1), we construct logical schemas for an AFD set and model tuple-level conflicts under the selected AFDs using a conflict graph, where nodes represent tuples and edges indicate that two tuples cannot be simultaneously decomposed. This reduces the problem to finding a maximal independent set in the conflict graph. Based on this formulation, we adopt a greedy strategy that prioritizes low-conflict for decomposition and further improve efficiency through lightweight degree maintenance. To address Challenge (2), we develop a cost-aware AFD selection framework. We first formulate the notion of *duplicate gain* to measure the storage benefit of an AFD subset. Our basic idea is to filter out non-duplicate AFDs by leveraging duplicate gain bounds. We observe that, in a good AFD set, each attribute appears at most once on the RHS, then propose a filtering method that leverages AFD errors to estimate bounds for all candidate LHSs and removes AFDs whose LHSs are guaranteed not to yield the maximal gain for that attribute. Furthermore, we utilize duplicate gain bounds of AFD subsets to skip many costly and unnecessary computations when selecting a good AFD subset.

The main contributions are as follows: (1) We propose RelaxRD, a storage-centric framework for relation design. To the best of our knowledge, this is the first work to shift schema design from rule-based logical to cost-driven physical design. (2) We are the first to study AFD-based schema design. We formulate relaxed decomposition under an AFD set as a conflict-aware optimization problem and develop an efficient decomposition algorithm. (3) We quantify the storage value of AFD subsets and develop an efficient AFD selection framework to identify the AFD subset that minimizes the storage cost of decomposed relations. (4) We offer an exploratory downstream-task perspective on AFD discovery through relation design. Our results show that not all mined AFDs are equally useful for downstream decomposition, suggesting a promising direction from exhaustive AFD enumeration toward utility-aware discovery. (5) We evaluate RelaxRD on several real datasets, and extensive experiments show that it achieves significant storage savings.

2 PRELIMINARIES

In this section, we introduce foundational concepts of relation design that are essential for understanding our work, including background and problem definition.

Let $U = \{A_1, A_2, \ldots, A_n\}$ be the set of attributes. The relation schema R over U defines the structure of the relation in the database, denoted as $R(U)$. The relation instance $r(R)$, or simply r, is a finite set of tuples $T = \{t_1, t_2, \ldots, t_m\}$, where each tuple t_i corresponds to the attributes in U. The total size of the relation, size(r), is simplified as $m \cdot n$, where $m = |T|$ (the number of tuples) and $n = |U|$ (the number of attributes). It is feasible to measure the size using the exact bytes of each attribute, but for simplicity, we uniformly approximate it by the number of cells ($m \cdot n$). For any attribute subset $X \subseteq U$, we use $\mathrm{d}(X)$ to denote the number of distinct tuples in r onto X, and $|X|$ to represent the number of attributes in set X.

Definition 2.1 (Approximate Functional Dependency). Given an error threshold ε, an attribute set X (the *left-hand side*, LHS) approximately determines the attribute A (the *right-hand side*, RHS) if the error $\mathrm{e}(X \to A) \leq \varepsilon$, denoted as $X \xrightarrow{\mathrm{e} \leq \varepsilon} A$, simply $X \xrightarrow{\varepsilon} A$. Here, $\mathrm{e}(X \to A)$ denotes the minimal fraction of tuples that must be removed from the relation r to make $X \to A$ hold.

For example, under $\varepsilon = 0.1$, *ItemDetail* → *Amt* holds in r (Figure 1(b)). For *ItemDetail* = ID2, tuples t_6, t_1, and t_9 share the same *ItemDetail* value but have different Amt values, thus violating this dependency (25.00 vs. 28.00). Removing t_6 leaves a consistent mapping ID2 → 28.00, satisfying the dependency within $\varepsilon = 0.1$.

Existing AFD discovery algorithms [12, 17] enumerate all minimal non-trivial AFDs of the form $X \xrightarrow{\varepsilon} A$ whose error does not exceed a given threshold ε, denoted as Σ. For any subset $\mathcal{F} \subseteq \Sigma$, we define its error rate $\mathrm{e}(\mathcal{F})$ as the minimum fraction of tuples that must be removed from the relation r to make all AFDs in $\mathcal{F}$ hold. Further, we define $\mathrm{M}(\mathcal{F})$ as an operation that groups AFDs with the same LHS into a single merged dependency of the form $X \xrightarrow{\varepsilon} Y$, where $Y = \{A \mid X \xrightarrow{\varepsilon} A \in \mathcal{F}\}$. An AFD subset $\mathcal{F} \subseteq \Sigma$ generally

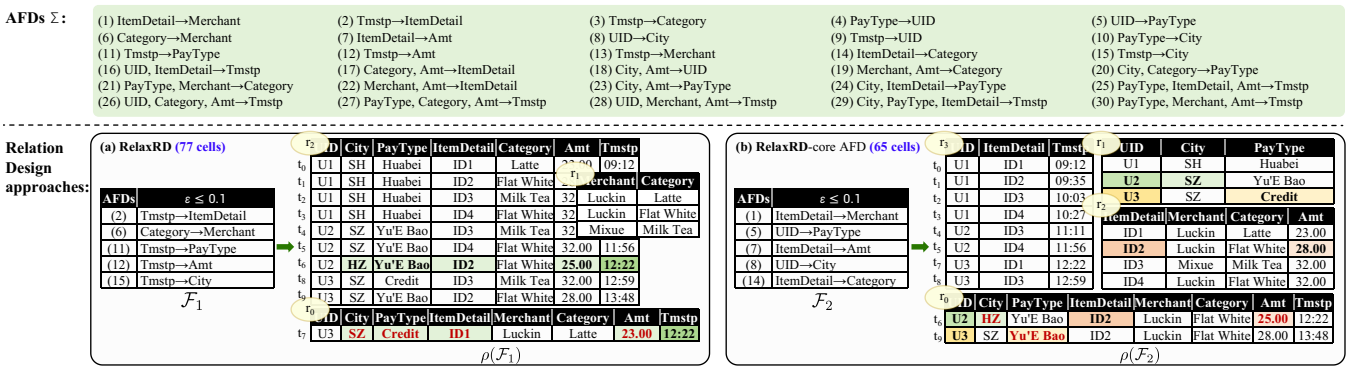


Figure 2: A running example comparing sampled original relation, 80 cells (10 tuples × 8 attrs) shown in Figure 1(b) and RelaxRD, where all AFDs Σ along with their IDs have been reported: (a) RelaxRD guided by random AFDs occupies 77 cells, (b) while RelaxRD guided by a good AFD subset only occupies 65 cells, achieving a 19% storage savings, and is considered the best.

does not satisfy Armstrong's axioms [5], such as Transitivity and Union, over the entire relation r. However, when restricted to the subset of tuples that satisfy all AFDs in $\mathcal{F}$, $\mathcal{F}$ can be treated as an exact FD set and satisfies Armstrong's axioms.

Relaxed relation design: given a relation instance r over schema R and an AFD subset $\mathcal{F}$, the relaxed decomposition constructs a set of relation schemas $\{R_1, \ldots, R_k\}$ based on $\mathcal{F}$. Tuples in r that satisfy all AFDs in $\mathcal{F}$ are decomposed into a set of relations $\{r_1, \ldots, r_k\}$, where each r_i is an instance of schema R_i, while tuples that violate at least one AFD are retained in relation r_0 over the original schema R. The resulting set of relations is denoted as $\rho(\mathcal{F}) = \{r_1, \ldots, r_k, r_0\}$. The decomposition is lossless, i.e., all tuples in r can be exactly reconstructed without information loss: joining $r_1, \ldots, r_k$ reconstructs all tuples in r that satisfy $\mathcal{F}$, and union with r_0 restores the remaining tuples. Formally, $(\bowtie_{i=1}^{k} r_i) \bigcup r_0 = r$. The total storage size induced by $\mathcal{F}$ is defined as $C(\mathcal{F}) = \sum_{r_i \in \rho(\mathcal{F})} \text{size}(r_i)$.

RelaxRD. Given a relation r over schema R and a candidate AFD set Σ over r, the goal of RelaxRD is to select an AFD subset $\mathcal{F}^* \subseteq \Sigma$ that minimizes the storage cost, i.e., $\mathcal{F}^* = \arg\min_{\mathcal{F} \subseteq \Sigma} C(\mathcal{F})$.

Theorem 2.1. *RelaxRD is an NP-hard problem, which can be reduced from the Set Cover Problem.*

3 RELAXED RELATION DESIGN

In this section, we study relaxed decomposition. We first present the schema construction process, then describe how tuples are decomposed into normalized relations, and finally prove the losslessness.

3.1 Schema Construction

We illustrate the schema construction process using a running example: decompose relation r in Figure 1(b) using the AFD subset shown in Figure 3(a), and obtain the resulting schema in Figure 3(b).

Given the AFD subset $\mathcal{F}$ reported in Figure 2(b), we first merge the AFDs with the same LHS, denoted as $\mathrm{M}(\mathcal{F}) = \{X_i \rightarrow Y_i\}_{i=1}^{k}$. For each dependency $X_i \rightarrow Y_i \in \mathrm{M}(\mathcal{F})$, we construct a relation schema $R_i = X_i \cup Y_i$. To ensure lossless reconstruction, we introduce a fact schema R_{k+1} that captures a global key of R and includes attributes not covered by the selected AFDs. Specifically, we construct R_{k+1} by first taking all LHS attributes, $R_{k+1} = \bigcup_{i=1}^{k} X_i$, and then removing

Algorithm 1: SchemaConstruction

Input: AFDs $\mathcal{F} = \{X_i \overset{\varepsilon}{\rightarrow} Y_i\}_{i=1}^{k}$, universal attribute set U.
Output: Relation Schema $\mathcal{R}(\mathcal{F})$.

```
1  Initialize R(F) ← ∅, R_{k+1} ← ∅, R_0 ← U;
2  foreach X_i →^ε Y_i ∈ F do
3      R_i = X_i ∪ Y_i;
4      Add R_i to R(F);
5      R_{k+1} ← R_{k+1} ∪ X_i;
6  foreach A ∈ R_{k+1} do
7      if R_{k+1} \ {A} → A can be inferred by F then
8          Remove A from R_{k+1};
9  Add attributes in U but not covered by F to R_{k+1};
10 Add R_{k+1} and R_0 to R(F);
11 return R(F)
```

any redundant attribute A such that $R_{k+1} \setminus \{A\} \rightarrow A$ holds under $\mathrm{M}(\mathcal{F})$, since such attributes can be inferred and need not be stored explicitly. Finally, attributes in R not covered by $\mathcal{F}$ are added to ensure full attribute coverage. We define a base schema R_0, which includes all attributes in U and will be used to store tuples that violate $\mathcal{F}$. Consequently, the resulting schema family consists of $k+2$ relation schemas, denoted as $\mathcal{R}(\mathcal{F}) = \{R_0, R_1, R_2, \cdots, R_{k+1}\}$.

Algorithm 1 describes the procedure for constructing the schema set $\mathcal{R}(\mathcal{F})$ from a given AFD subset $\mathcal{F}$. For each AFD $X_i \overset{\varepsilon}{\rightarrow} Y_i \in \mathcal{F}$, a relation schema R_i is created as the union of its left- and right-hand side attributes (Lines 2–4). Meanwhile, all LHS attributes are accumulated into a provisional fact schema R_{k+1}. Each attribute in R_{k+1} is then examined and removed if it is functionally determined by the remaining attributes under $\mathrm{M}(\mathcal{F})$ (Lines 5–7), ensuring that no redundant attributes are retained. Attributes in U that are not covered by any AFD are subsequently added to R_{k+1} (Line 8) to ensure completeness. Finally, the resulting relation schemas together with the base schema R_0 are returned as $\mathcal{R}(\mathcal{F})$. It runs in $O(|\mathcal{F}| \cdot n + n^2)$ time and uses $O(|\mathcal{F}| \cdot n)$ space, where n is the number of attributes. Since in a core AFD set each attribute appears on the right-hand side at most once, we have $|\mathcal{F}| \leq n$; therefore, both the time and space complexities reduce to $O(n^2)$.

Consider our running example in Figure 3(a–b). $\mathrm{M}(\mathcal{F})$ contains two AFDs, which yield two schemas: R_1 = {*UID, City, PayType*}

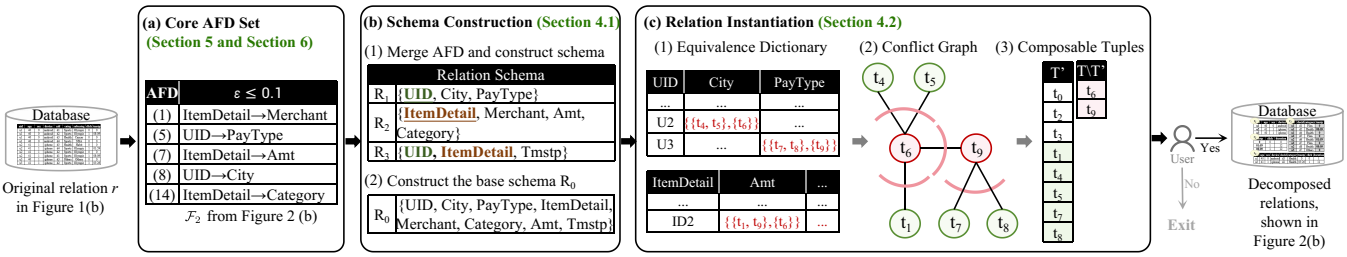


Figure 3: The process of RelaxRD under a given AFD subset (a) (corresponding to $\mathcal{F}_2$ in Figure 2(b)), including logical schema construction (b) and relation instantiation (c). The schema is proposed to users and executed only upon their approval. Tuple identifiers t_i refer to tuples in the original relation r (Figure 1(b)) and are preserved throughout the decomposition for clarity.

and $R_2 = \{ItemDetail, Merchant, Amt, Category\}$. To ensure lossless reconstruction, we further construct the fact schema R_3, which connects these schemas and covers attributes not captured by individual AFDs. It is initialized with all LHS attributes in $\mathcal{F}$, i.e., $R_3 = \{UID, ItemDetail\}$. In this example, no attribute in R_3 can be removed, since under $\mathcal{F}$, $UID \nrightarrow ItemDetail$ and $ItemDetail \nrightarrow UID$. Finally, attributes not covered by the two AFDs are added, i.e., *Tmstp*, to ensure full attribute coverage, resulting in $R_3 = \{UID, ItemDetail, Tmstp\}$. In addition, we define a base schema $R_0 = U$ to store tuples that violate $\mathcal{F}$. Therefore, $\mathcal{R}(\mathcal{F}) = \{R_0, R_1, R_2, R_3\}$.

Discussion. A canonical cover of $\mathcal{F}$ is a minimal equivalent set of dependencies without unnecessary attributes or FDs. Specifically, the canonical cover $\mathcal{F}_c$ of $\mathcal{F}$ is a dependency set logically equivalent to $\mathcal{F}$, meaning that $\mathcal{F}$ and $\mathcal{F}_c$ imply each other and satisfy the following properties [37]: (i) no dependency in $\mathcal{F}_c$ contains extraneous attributes; and (ii) there do not exist two distinct dependencies $X_1 \rightarrow Y_1$ and $X_2 \rightarrow Y_2$ in $\mathcal{F}_c$ such that $X_1 = X_2$. In traditional normalization, the canonical cover is computed prior to schema design to eliminate unnecessary FDs and attributes. Here, we do not need to explicitly compute the canonical cover of $\mathrm{M}(\mathcal{F})$ since: (i) if $\mathrm{M}(\mathcal{F})$ is already a canonical cover, then $\mathcal{R}(\mathcal{F})$ is in 3NF by the classical synthesis procedure [37]; (ii) otherwise, computing the canonical cover would require removing unnecessary AFDs, implying that there exists a better AFD subset $\mathcal{F}'$, which transforms the problem into an AFD selection problem (Section 4).

3.2 Relation Instantiation

In this subsection, we study how to decompose the relation r based on the schemas $\mathcal{R}(\mathcal{F})$ constructed in Section 3.1, as not all tuples in r satisfy the given AFDs $\mathcal{F}$. We first identify the maximal AFD-satisfying tuples in r, and then decompose them according to $\mathcal{R}(\mathcal{F})$.

Definition 3.1 (Maximal AFD-Satisfying Tuple Set). Given an AFD set $\mathcal{F}$ holding on the relation r, the tuple subset $T' \subseteq T$ is a maximal AFD-satisfying tuple set if all tuples in T' satisfy all AFDs in $\mathcal{F}$, and for every tuple $t \in T \setminus T'$, $T' \cup \{t\}$ violates at least one AFD in $\mathcal{F}$.

To achieve this, we construct an equivalence dictionary for each AFD $\varphi : X \xrightarrow{\varepsilon} A$ in $\mathcal{F}$, denoted as $\mathcal{E}_\varphi$. For each value of X, tuples are grouped by their A values; however, $\mathcal{E}_\varphi$ records only those X values that form multiple groups, i.e., those that violate AFD φ. For example, for AFD (5): $UID \rightarrow PayType$, only $UID = U3$ violates this dependency, since t_7, t_8, t_9 share $UID = U3$ but correspond to different $PayType$ values ($Credit$ in t_7, t_8 vs. $Yu'EBao$ in t_9) in Figure 1(b), yielding $\mathcal{E}_{(5)}(U3) = \{\{t_7, t_8\}, \{t_9\}\}$ (Figure 3(c)).

Definition 3.2 (Conflict Graph). Given an equivalence dictionary $\mathcal{E}$ constructed from an AFD set $\mathcal{F}$, the conflict graph $\mathcal{G}_T = (V, E)$ is an undirected graph where each node in V corresponds to a tuple in r, and an edge $(t_i, t_j) \in E$ exists if $\exists X \xrightarrow{\varepsilon} A \in \mathcal{F}$ such that $t_i[X] = t_j[X]$ but $t_i[A] \neq t_j[A]$. Isolated nodes are excluded by V.

For example, in Figure 3(c), $(t_8, t_9) \in E$ and $(t_7, t_9) \in E$, since they share the same UID value ($U3$) but have different $PayType$ values.

Theorem 3.1. *Identifying the maximal AFD-satisfying tuples is equivalent to finding the maximal independent set in conflict graph.*

Based on this equivalence, we reduce the problem to extracting a large independent set from the conflict graph and adopt a greedy strategy to find it. Given a relation instance r and an AFD subset $\mathcal{F}$, we construct the conflict graph $G_T = (V, E)$ based on the equivalence dictionaries. We then iteratively add the node with the minimum degree to T', while lazy-deleting it together with its neighbors from V. The algorithm returns two tuple-ID sets: T', containing tuples that satisfy all AFDs in $\mathcal{F}$, and $T \setminus T'$, containing the remaining violating tuples. Finally, tuples in $r[T']$ are projected onto each schema R_i to instantiate the decomposed relations $r_i = \pi_{R_i}(r[T'])$, where $r[T'] = \{ t_l \in r \mid l \in T' \}$, while tuples in $T \setminus T'$ are retained in $r_0 = r[T \setminus T']$. The decomposition is $\rho(\mathcal{F}) = \{r_0, r_1, \ldots, r_{k+1}\}$.

Consider our running example in Figure 3(c), the algorithm first adds t_1 (degree 1) to T' and removes its neighbor t_6, then sequentially adds t_4 and t_5 (each with degree 0) to T', followed by t_7 and t_8, while excluding their neighbor t_9. Tuples not in the conflict graph, t_0, t_2, t_3, are added directly to T'. Thus, the decomposable set is $T' = \{t_0, t_1, t_2, t_3, t_4, t_5, t_7, t_8\}$, and the removed tuples are $T \setminus T' = \{t_6, t_9\}$. Tuples indexed by T' are first collected from the original relation r to form $r[T']$, and are then materialized under schemas R_1, R_2, and R_3 (Section 4.1) via projection: $r_1 = \pi_{R_1}(r[T'])$, $r_2 = \pi_{R_2}(r[T'])$, and $r_3 = \pi_{R_3}(r[T'])$ (see r_1, r_2, and r_3 in Figure 2(b)). The remaining tuples indexed by $T \setminus T'$ are stored in $r_0 = r[T \setminus T']$ under the base schema R_0 (see r_0 in Figure 2(b)).

Algorithm 3 describes the relation instantiation procedure. Given a relation instance r, an AFD set $\mathcal{F}$, and the relation schemas $\mathcal{R}(\mathcal{F})$, it first constructs the equivalence dictionary $\mathcal{E}$ based on $\mathcal{F}$ and r (Line 1). It then invokes MaximalAFDTuples$(r, \mathcal{F}, \mathcal{E})$ to identify a large jointly decomposable tuple subset T', together with its

Algorithm 2: MaximalAFDTuples($r, \mathcal{F}, \mathcal{E}$)

1 Initialize tuple index $T' \leftarrow \emptyset$, conflict graph $\mathcal{G}_T$;
2 **foreach** $\varphi : X \xrightarrow{\varepsilon} A \in \mathcal{F}$ **do**
3 Construct the conflict graph $\mathcal{G}_T = (V, E)$ via $\mathcal{E}_\varphi$;
4 Add all tuple ID not in V to T';
5 **while** $V \neq \emptyset$ **do**
6 Add the node with minimum degree in $\mathcal{G}_T$ to T';
7 Lazy delete this node and its all neighbors;
8 **return** $T', T \setminus T'$

Algorithm 3: RelationDesign

Input: Relation instance r, AFDs $\mathcal{F}$, relation schemas $\mathcal{R}(\mathcal{F})$.
Output: Decomposed relations $\rho(\mathcal{F}) = \{r_0, r_1, \ldots, r_{k+1}\}$
1 Construct the equivalence dictionary $\mathcal{E}$ based on $\mathcal{F}$ and r;
2 $T', T \setminus T' \leftarrow$ MaximalAFDTuples($r, \mathcal{F}, \mathcal{E}$);
3 Initialize $r_0, r_1, \ldots, r_{k+1} \leftarrow \emptyset$;
4 **foreach** *schema* $R_i \in \mathcal{R}(\mathcal{F})$ **do**
5 $r_i \leftarrow \pi_{R_i}(r[T'])$
6 $r_0 \leftarrow r[T \setminus T']$;
7 **return** $r_0, r_1, \ldots, r_{k+1}$;

complement $T \setminus T'$ (Line 2). Next, the algorithm initializes the output relations $r_0, r_1, \ldots, r_{k+1}$ as empty (Line 3). For any index set $I \subseteq T$, we define the induced subrelation of r as $r[I] := \{\, t_i \in r \mid i \in I \,\}$. For each schema $R_i \in \mathcal{R}(\mathcal{F})$, the corresponding relation instance is constructed by projecting the tuples indexed by T' onto R_i, i.e., $r_i = \pi_{R_i}(r[T'])$ (Lines 4–5). Finally, the base relation is formed by the tuples indexed by $T \setminus T'$, namely $r_0 = r[T \setminus T']$ (Line 6). The algorithm returns the instantiated decomposition $\rho(\mathcal{F}) = \{r_0, r_1, \ldots, r_{k+1}\}$ (Line 7).

Algorithm 2 runs in $O(|\mathcal{F}|m^2 + m^2 \log m)$ time in the worst case, where $n = |U|$ and $m = |r|$: the first term comes from constructing tuple conflicts for all AFDs, and the second from greedily extracting a large independent set from the conflict graph using minimum-degree selection with lazy deletion.

Discussion: the effectiveness of RelaxRD. It is worth noting that the maximal AFD-satisfying tuples are not unique, and different selections may lead to different storage costs. This issue stems from focusing on AFD violations while overlooking value-level duplicates across tuples. Theoretically, further storage savings could be achieved by modeling value-level duplicates. However, such fine-grained optimization greatly increases complexity while offering limited gains (only a few cells), especially under small ε. Therefore, we tolerate slight duplicates and identify the maximal AFD-satisfying tuple set at the tuple level.

3.3 Losslessness and Consistency

Logical schema design concerns the correctness, including lossless reconstruction and reducing redundancy that would otherwise lead to anomalies. In this subsection, we show that RelaxRD can guarantee lossless reconstruction. In addition, its schema design follows the normalization principle of avoiding unnecessary redundancy, thereby mitigating update, insertion, and deletion anomalies.

Theorem 3.2 (Losslessness of RelaxRD). *Let r be a relation instance over $R(U)$, and $\mathrm{M}(\mathcal{F}) = \{X_i \to Y_i\}_{i=1}^{k}$ be a merged AFD subset. RelaxRD constructs schema $\mathcal{R}(\mathcal{F}) = \{R_0, R_1, \cdots, R_{k+1}\}$ and instantiates the relation instances $\rho(\mathcal{F}) = \{r_0, r_1, \cdots, r_{k+1}\}$. The relation r can be losslessly reconstructed as $r = \left(\bowtie_{i=1}^{k+1} r_i\right) \cup r_0$.*

Proof. Let $T' \subseteq T$ be the maximal set of tuples satisfying all AFDs in $\mathcal{F}$, and define $r' = r[T']$ and $r_0 = r[T \setminus T']$. Then $r = r' \cup r_0$ and $r' \cap r_0 = \emptyset$. $\mathcal{R}(\mathcal{F}) = \{R_0, R_1, \ldots, R_{k+1}\}$ guarantees (i) $\bigcup_{i=1}^{k+1} R_i = U$, and (ii) the fact schema R_{k+1} preserves a global candidate key K of R for tuples in r'. For each $i \le k$, the LHS X_i of the AFD $X_i \to Y_i$ (a primary key of R_i) is contained in R_{k+1} or is functionally determined by attributes in R_{k+1} under $\mathcal{F}$. Hence, joining $\pi_{R_{k+1}}(r')$ with $\pi_{R_i}(r')$ on the shared key attributes is lossless. Let $r_i = \pi_{R_i}(r')$ for all $i = 1, \ldots, k+1$. By iteratively joining r_{k+1} with $r_1, \ldots, r_k$, we obtain $r' = \bowtie_{i=1}^{k+1} r_i$. Combining with $r = r' \cup r_0$ yields $r = \left(\bowtie_{i=1}^{k+1} r_i\right) \cup r_0$. □

4 CONSTRUCTING GOOD AFDS

In this section, we study how to construct a good AFD set to perform decomposition with minimal storage. We define the core AFD set, formulate *duplicate gain* to quantify the benefits of AFD subsets, and then show two key properties that the core AFD set must satisfy (Section 4.1). We propose efficient filter algorithms to remove inefficient AFDs (Section 4.2 and Section 4.3).

4.1 Core AFD Set

Definition 4.1 (Core AFD Set). A subset of AFDs $\mathcal{F}^*$, w.r.t. a relation r, is called the core AFD set if it yields the minimal total size, i.e., $\mathcal{F}^* = \arg\min_{\mathcal{F} \subseteq \Sigma} \sum_{r_i \in \rho(\mathcal{F})} \mathrm{size}(r_i)$, where Σ denotes complete set of AFDs over relation r whose error does not exceed ε.

For example, the core AFD set $\mathcal{F}^*$ w.r.t. relation r in Figure 1(b) is {(1), (5), (7), (8), (14)}. The size of its resulting decomposition is 65, achieving the globally minimal storage among all AFD subsets.

Next, we quantify the storage savings of an AFD subset $\mathcal{F}$, which reflects the benefit of decomposing the relation r according to $\mathcal{F}$.

Definition 4.2 (Duplicate Gain). Given an AFD subset $\mathcal{F}$ holding on relation r, the duplicate gain of $\mathcal{F}$ in r quantifies the storage savings when decomposing r according to $\mathcal{F}$, and is defined as: $\mathrm{dg}(\mathcal{F}) = \sum_{X \xrightarrow{\varepsilon} Y \in \mathrm{M}(\mathcal{F})} (m_{\mathcal{F}} - \mathrm{d}_{\mathcal{F}}(X)) \cdot |Y| - \mathrm{d}_{\mathcal{F}}(X) \cdot |X|$. where $m = |r|$ denotes the number of tuples in r, and $m_{\mathcal{F}} = |r[T']|$ denotes the number of tuples that satisfy all AFDs in $\mathcal{F}$ (also denoted as $m \cdot (1 - \mathrm{e}(\mathcal{F}))$, $\mathrm{e}(\mathcal{F})$ is the error rate of $\mathcal{F}$). $\mathrm{d}_{\mathcal{F}}(X)$ denotes the number of distinct values of X among the tuples in $r[T']$.

For example, consider the relation r in Figure 1(b) with the AFD subset $\mathcal{F}$ in Figure 2(b). The relation contains $m = 10$ tuples, among which t_6 and t_9 violate $\mathcal{F}$, yielding $\mathrm{e}(\mathcal{F}) = 0.2$ and $m_{\mathcal{F}} = |T'| = 8$. For $UID \xrightarrow{\varepsilon} \{City, PayType\}$, the number of distinct UID values among the 8 satisfying tuples is $\mathrm{d}_{\mathcal{F}}(UID) = 3$. Substituting into Definition 4.2, we obtain $(8-3) \times 2 - 3 \times 1 = 7$. For $ItemDetail \xrightarrow{\varepsilon} \{Merchant, Amt, Category\}$, we have $\mathrm{d}_{\mathcal{F}}(ItemDetail) = 4$, yielding $(8-4) \times 3 - 4 \times 1 = 8$. Therefore, $\mathrm{dg}(\mathcal{F}) = 7 + 8 = 15$.

For each AFD $X \xrightarrow{\varepsilon} Y$ in $\mathrm{M}(\mathcal{F})$, the gains consist of two components: (i) Decomposing savings: $(m_{\mathcal{F}} - \mathrm{d}_{\mathcal{F}}(X)) \cdot |Y|$, representing the number of duplicates eliminated in the RHS Y among composable tuples; (ii) Primary key cost: $|X| \cdot \mathrm{d}_{\mathcal{F}}(X)$, accounting for the additional cost of materializing the LHS X as a primary key. Thus, the duplicate gain of this AFD in $\mathcal{F}$ is denoted as $\mathrm{dg}_{\mathcal{F}}(X \xrightarrow{\varepsilon} Y) = (m_{\mathcal{F}} - \mathrm{d}_{\mathcal{F}}(X)) \cdot |Y| - \mathrm{d}_{\mathcal{F}}(X) \cdot |X|$.

Property 4.1. *Given a set of AFDs $\mathcal{F}$ holding on a relation r, the duplicate gain and the size of the decomposed relations $\rho(\mathcal{F})$ satisfy:* $\mathrm{dg}(\mathcal{F}) + \sum_{r_i \in \rho(\mathcal{F})} \mathrm{size}(r_i) = m \cdot |U| + a \cdot m_{\mathcal{F}}$, *where* $a \in \mathbb{N}$.

Proof. Give an AFD subset $\mathcal{F}$ over a relation r, its $\mathrm{M}(\mathcal{F})$ is $\{X_i \xrightarrow{\varepsilon} Y_i\}_{i=1}^{k}$, and the decomposed relation based on it is $\rho(\mathcal{F}) = \{r_0, r_1, \ldots, r_{k+1}\}$ (see Section 3.2). We compute the total size of decomposed relation $\rho(\mathcal{F})$: $\sum_{i=0}^{k+1} \mathrm{size}(r_i) = (m - m_{\mathcal{F}}) \cdot |U| + \sum_{i=1}^{k} (|X_i| + |Y_i|) \cdot \mathrm{d}_{\mathcal{F}}(X_i) + m_{\mathcal{F}} \cdot |R_{k+1}|$. Adding both terms, we obtain: $\mathrm{dg}(\mathcal{F}) + \sum_{i=0}^{k+1} \mathrm{size}(r_i) = (m - m_{\mathcal{F}}) \cdot |U| + \sum_{i=1}^{k} \mathrm{d}_{\mathcal{F}}(X_i) \cdot (|X_i| + |Y_i|) + m_{\mathcal{F}} \cdot |R_{k+1}| + m_{\mathcal{F}} \cdot \sum_{i=1}^{k} |Y_i| - \sum_{i=1}^{k} \mathrm{d}_{\mathcal{F}}(X_i) \cdot (|X_i| + |Y_i|) = (m - m_{\mathcal{F}}) \cdot |U| + m_{\mathcal{F}} \cdot (\sum_{i=1}^{k} |Y_i| + |R_{k+1}|) = m \cdot |U| + a \cdot m_{\mathcal{F}}$, where $a = \sum_{i=1}^{k} |Y_i| + |R_{k+1}| - |U|$, is a non-negative integer. □

Consider our running example, decomposed relations $\rho(\mathcal{F}) = \{r_0, r_1, r_2, r_3\}$, whose size is 65. $a = 5 + 3 - 8 = 0$, and $\mathrm{dg}(\mathcal{F}) + \sum_{r_i \in \rho(\mathcal{F})} \mathrm{size}(r_i) = 15 + 65 = 8 \times 10 + 0$, verifying Property 4.1.

Lemma 4.1. *The core AFD set $\mathcal{F}^*$ w.r.t. relation r satisfies the following property:* $\mathrm{dg}(\mathcal{F}^*) + \sum_{r_i \in \rho(\mathcal{F}^*)} \mathrm{size}(r_i) = m \cdot |U|$.

Proof. Assume, for the sake of contradiction, that $\mathcal{F}^*$ is the core AFD set, its $\mathrm{M}(\mathcal{F}^*) = \{X_i \xrightarrow{\varepsilon} Y_i\}_{i=1}^{k}$ and the following holds: $a = \sum_{i=1}^{k} |Y_i| + |R_{k+1}| - |U| > 0$. Then, there exists $S_i \neq S_j \in \{Y_1, \ldots, Y_k, R_{k+1}\}$ such that $A \in S_i \cap S_j$. We now analyze all possible cases: Case 1: $A \in Y_i \cap Y_j$ for some $i \neq j$. That is, A appears as the RHS of both $X_i \xrightarrow{\varepsilon} A$ and $X_j \xrightarrow{\varepsilon} A$ in $\mathcal{F}^*$. Removing one of them (say, $X_j \xrightarrow{\varepsilon} A$) reduces the total size by at least $\mathrm{d}_{\mathcal{F}}(X_j)$, contradicting the optimality of $\mathcal{F}^*$. Case 2: $A \in Y_i \cap R_{k+1}$. That is, A appears both as the RHS of some AFD $X_i \xrightarrow{\varepsilon} A \in \mathcal{F}^*$ and in the LHS of another $X_j \xrightarrow{\varepsilon} Y_j \in \mathcal{F}^*$. If $R_{k+1} \setminus \{A\} \to A$, removing A from R_{k+1} reduces size by at least $m_{\mathcal{F}}$; otherwise, removing A from the RHS of $X_i \xrightarrow{\varepsilon} A$ reduces it by at least $\mathrm{d}_{\mathcal{F}}(X_i)$. In either case, this contradicts the optimality of $\mathcal{F}^*$. Case 3: It is not possible for A to appear multiple times in R_{k+1}. Hence, the core AFD set $\mathcal{F}^*$ must hold $a = 0$, i.e., $\mathrm{dg}(\mathcal{F}^*) + \sum_{r_i \in \rho(\mathcal{F}^*)} \mathrm{size}(r_i) = m \cdot |U|$. This completes the proof. □

Finding the core AFD set to minimize the decomposition is equivalent to finding the maximum duplicate gain under the constraint $a = |Y_i| + |R_{k+1}| - |U| = 0$, since $m \cdot |U|$ is a constant. This allows us to avoid physically evaluating the decomposition size for each candidate AFD subset. Instead, we can focus on those satisfying $a = 0$ and select the one with the highest duplicate gain.

Theorem 4.1. *The core AFD set is equivalent to the following problem:* $\mathcal{F}^* = \arg\max_{\mathcal{F} \subseteq \Sigma} \mathrm{dg}(\mathcal{F})$ *under the constraint of* $a = 0$.

Proof. By Lemma 4.1, for any AFD subset $\mathcal{F}$ we have $\mathrm{dg}(\mathcal{F}) + \sum_{r_i \in \rho(\mathcal{F})} \mathrm{size}(r_i) = m \cdot |U| + a \cdot m_{\mathcal{F}}$, where $a \geq 0$. For the core AFD set $\mathcal{F}^*$, $a = 0$, and therefore $\mathrm{dg}(\mathcal{F}^*) + \sum_{r_i \in \rho(\mathcal{F}^*)} \mathrm{size}(r_i) = m \cdot |U|$. Since $m \cdot |U|$ is a constant for a fixed relation r, maximizing $\mathrm{dg}(\mathcal{F})$ is equivalent to minimizing $\sum_{r_i \in \rho(\mathcal{F})} \mathrm{size}(r_i)$. According to Definition 5.1, the core AFD set $\mathcal{F}^*$ is defined as the AFD subset that minimizes the total storage size. Therefore, $\mathcal{F}^*$ must also maximize $\mathrm{dg}(\mathcal{F})$ among all $\mathcal{F} \subseteq \Sigma$ satisfying $a = 0$. Hence, $\mathcal{F}^* = \arg\max_{\mathcal{F} \subseteq \Sigma} \mathrm{dg}(\mathcal{F})$ under the constraint $a = 0$. This completes the proof. □

4.2 Filter Non-Duplicate AFDs

One naive way to select the core AFD set is to first discover Σ using existing methods, and then exhaustively enumerate all subsets that satisfy the constraint $a = 0$ required by Theorem 5.1. However, it is computationally expensive. Instead of directly enumerating all subsets, we use two key properties of $\mathcal{F}^*$ to filter out ineffective AFDs, significantly reducing the number of AFDs before enumeration.

Lemma 4.2. *Let $\Sigma_X = \{X \xrightarrow{\varepsilon} A \in \Sigma\}$ denote all AFDs in Σ sharing the same LHS X. If $\mathrm{dg}(\Sigma_X) < 0$, then none of them can be included in the core set; that is, $X \xrightarrow{\varepsilon} A \notin \mathcal{F}^*$ for all $X \xrightarrow{\varepsilon} A \in \Sigma_X$ if $\mathrm{dg}(\Sigma_X) < 0$.*

Proof. Fix an attribute set X and let $\Sigma_X = \{X \xrightarrow{\varepsilon} A \in \Sigma\}$. For any AFD subset $\mathcal{F} \subseteq \Sigma$, denote by $\mathcal{F}_X = \mathcal{F} \cap \Sigma_X$ the selected dependencies whose LHS is X. After merging, $\mathrm{M}(\mathcal{F}_X)$ contains a single dependency of the form $X \xrightarrow{\varepsilon} Y$, where $Y = \{A \mid X \xrightarrow{\varepsilon} A \in \mathcal{F}_X\}$. According to Definition 4.2, this merged dependency contributes $\mathrm{dg}_{\mathcal{F}}(X \to Y) = (m_{\mathcal{F}} - \mathrm{d}_{\mathcal{F}}(X)) \cdot |Y| - \mathrm{d}_{\mathcal{F}}(X) \cdot |X|$ to the objective $\mathrm{dg}(\mathcal{F})$. In particular, when taking $\mathcal{F} = \Sigma_X$, the merged dependency becomes $X \xrightarrow{\varepsilon} Y$ where Y contains all RHS attributes under X, and its duplicate gain equals $\mathrm{dg}(\Sigma_X)$. Now assume $\mathrm{dg}(\Sigma_X) < 0$. Consider any candidate solution $\mathcal{F}$ with $\mathcal{F}_X \neq \emptyset$. Removing these dependencies yields $\mathcal{F}' = \mathcal{F} \setminus \Sigma_X$, whose merged set $\mathrm{M}(\mathcal{F}')$ contains no dependency with LHS X. The only structural difference between $\mathcal{F}$ and $\mathcal{F}'$ is the presence of the merged dependency with LHS X. Since the merged dependency over Σ_X already yields negative duplicate gain, any non-empty subset of these RHS attributes cannot produce a positive contribution. Consequently, including dependencies from Σ_X cannot improve the objective (it decreases duplicate gain, equivalently increases storage). Hence, no optimal solution contains any dependency from Σ_X, that is, $\mathcal{F}^* \cap \Sigma_X = \emptyset$. □

By this Lemma, we can safely filter all AFDs in Σ_X if $\mathrm{dg}(\Sigma_X) < 0$. However, computing exact $\mathrm{dg}(\Sigma_X)$ is expensive, as it requires constructing a conflict graph to find composable tuples under Σ_X.

Lemma 4.3. *The error of $\mathcal{F}$, $\mathrm{e}(\mathcal{F})$, satisfies the following bounds:*

$$\max_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi) \leq \mathrm{e}(\mathcal{F}) \leq \sum_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi), \tag{1}$$

Briefly, we denote $\mathrm{e}^{\mathrm{up}}(\mathcal{F}) = \sum_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi)$ and $\mathrm{e}^{\mathrm{low}}(\mathcal{F}) = \max_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi)$.

Proof. For each AFD $\varphi \in \mathcal{F}$, let $V_\varphi \subseteq T$ denote a minimum set of tuples whose removal makes φ hold exactly. By definition, $\mathrm{e}(\varphi) = \frac{|V_\varphi|}{|T|}$. Similarly, let $V_{\mathcal{F}} \subseteq T$ denote a minimum set of tuples whose removal makes all AFDs in $\mathcal{F}$ hold simultaneously, so $\mathrm{e}(\mathcal{F}) = \frac{|V_{\mathcal{F}}|}{|T|}$.

We first prove the lower bound. Since $V_{\mathcal{F}}$ makes every $\varphi \in \mathcal{F}$ hold, it is a feasible removal set for each individual φ. By the minimality of V_φ, we must have $|V_{\mathcal{F}}| \geq |V_\varphi|$ for all $\varphi \in \mathcal{F}$. Dividing both sides by $|T|$ gives $\mathrm{e}(\mathcal{F}) \geq \mathrm{e}(\varphi)$ for all $\varphi \in \mathcal{F}$, and therefore $\mathrm{e}(\mathcal{F}) \geq \max_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi)$. Next, we prove the upper bound. Consider the union of all individual removal sets $V^* = \bigcup_{\varphi \in \mathcal{F}} V_\varphi$. Removing V^* makes every φ hold, hence V^* is a feasible removal set for $\mathcal{F}$. By the minimality of $V_{\mathcal{F}}$, we obtain:

$$\mathrm{e}(\mathcal{F}) \leq \frac{|V^*|}{|T|} = \frac{\left|\bigcup_{\varphi \in \mathcal{F}} V_\varphi\right|}{|T|} \leq \sum_{\varphi \in \mathcal{F}} \frac{|V_\varphi|}{|T|} = \sum_{\varphi \in \mathcal{F}} \mathrm{e}(\varphi).$$

Combining them yields $\max_{\varphi\in\mathcal{F}} e(\varphi) \le e(\mathcal{F}) \le \sum_{\varphi\in\mathcal{F}} e(\varphi)$. □

To avoid this, we evaluate upper bound of $dg(\Sigma_X)$ via Lemma 4.3, having $dg^{up}(\Sigma_X) = (1-e^{low}(\Sigma_X))\cdot m\cdot|\Sigma_X| - (|X|+|\Sigma_X|)\cdot d(X)$. If it is negative, we remove Σ_X from Σ (Algorithm 5 Lines 4-6). For example, $\Sigma_{\{UID,ItemDetail\}}$ includes one AFD, and $dg^{up}(\Sigma_{\{UID,ItemDetail\}}) < 0$, indicating that they all do not belong to $\mathcal{F}^*$.

4.3 Filter Inefficient AFDs Using RHS Uniqueness.

The enumeration space, $|\Sigma|$, may still be large, and we can further filter out inefficient AFDs by a corollary of Lemma 4.1.

Corollary 4.1. *In the core AFD set $\mathcal{F}^*$, each attribute $A \in U$ appears in the right-hand side of at most one AFD.*

Proof. Assume, for the sake of contradiction, that A appears as the RHS of both $X_i \xrightarrow{\varepsilon} A$ and $X_j \xrightarrow{\varepsilon} A$ in $\mathcal{F}^*$. Removing one of them (say, $X_j \xrightarrow{\varepsilon} A$) can reduce the total size of $\rho(\mathcal{F}^*)$ by at least $d_{\mathcal{F}}(X_j)$, contradicting the minimality of $\mathcal{F}^*$ in Definition 4.1. □

Based on Corollary 4.1, an intuitive way is to retain, for each attribute A, the candidate AFD $X \xrightarrow{\varepsilon} A$ with the highest duplicate gain. However, this is infeasible since the contribution of an AFD depends on the subset in which it appears. Specifically, the duplicate gain of an AFD is not independent: it is determined by the set $\mathcal{F}$ through both the number of decomposable tuples and the storage cost of attributes. The subset $\mathcal{F}$ determines which tuples are decomposable, thereby affecting both $m_{\mathcal{F}}$ and $d_{\mathcal{F}}(X)$. Moreover, AFDs in $M(\mathcal{F})$ determine which LHS attributes need to be materialized, and thus affect the storage. As a result, evaluating an AFD in isolation may be misleading. For example, the AFD $ItemDetail \xrightarrow{\varepsilon} Category$ has different numbers of decomposable tuples in Figure 2(a) (9 tuples) and Figure 2(b) (8 tuples), leading to different duplicate gains. Therefore, without knowing the exact subset $\mathcal{F}$, it is impossible to accurately assess the contribution of an AFD.

To address this challenge, we bound the duplicate gain that an AFD can achieve across all subsets containing it, rather than computing its exact gain. Our key idea is to compare AFDs by their worst- and best-case gains: if the worst-case gain of one AFD exceeds the best-case gain of another, the latter can be safely filtered.

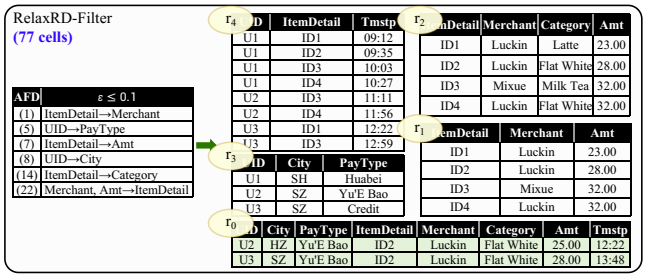


Figure 4: RelaxRD using Σ after twice filtering (77 cells).

To derive such bounds, we first estimate the possible error range of any subset containing a given AFD. By Corollary 4.1, for any AFD $\varphi \in \Sigma$, the maximum error of any AFD subset containing it equals its own error plus the maximal errors of all other RHS attributes in $U \setminus \{A\}$, while the minimum error is $e(\varphi)$ (see Equation (2)).

$$\begin{aligned} &\min_{\mathcal{F}\subseteq\Sigma,\,\varphi\in\mathcal{F}} e^{low}(\mathcal{F}) = e(\varphi), \\ &\max_{\mathcal{F}\subseteq\Sigma,\,\varphi\in\mathcal{F}} e^{up}(\mathcal{F}) = e(\varphi) + \sum_{B\in U\setminus\{A\}} \max_{X'\to B\in\Sigma} e(X'\to B). \end{aligned} \tag{2}$$

Using these bounds, we derive the worst- and best-case duplicate gains of an AFD across all subsets containing it.

Lemma 4.4. *For any AFD $\varphi : X \xrightarrow{\varepsilon} A$, its duplicate gain in any subset $\mathcal{F}$ lies within the following range:*

$$dg_{\mathcal{F}}(X \xrightarrow{\varepsilon} A) \in \Big[\,(1 - \max_{\mathcal{F}\subseteq\Sigma,\,\varphi\in\mathcal{F}} e^{up}(\mathcal{F}))\cdot m - (1+|X|)\cdot d(X),$$
$$(1 - \min_{\mathcal{F}\subseteq\Sigma,\,\varphi\in\mathcal{F}} e^{low}(\mathcal{F}))\cdot m - (1+|X|)\cdot (d(X) - m\cdot \min_{\mathcal{F}\subseteq\Sigma,\,\varphi\in\mathcal{F}} e^{low}(\mathcal{F}))\,\Big).$$

where m is the number of tuples in the relation; $\mathcal{F}$ is any AFD subset containing $\varphi : X \xrightarrow{\varepsilon} A$; $d(X)$ is the number of distinct values of X; $|X|$ is the number of attributes in X; and $e^{up}(\mathcal{F})$ and $e^{low}(\mathcal{F})$ are the error bounds defined in Equation (2).

This reduces subset-unknown evaluation to pairwise AFD comparison, allowing many inefficient AFDs to be filtered out early. More importantly, filtering can tighten the duplicate-gain bounds of the remaining AFDs. Once some AFDs are removed, the set of feasible subsets containing each remaining AFD becomes smaller, which leads to more accurate error bounds and changes the storage cost of primary-key attributes when they are no longer shared across multiple AFDs. This, in turn, enables further filtering of inefficient AFDs. For instance, after one round of filtering, only 7 AFDs remain in Σ; after a second round, Σ is further reduced to 6 AFDs, yielding a smaller decomposition (77 cells), as shown in Figure 4.

Algorithm 5 proceeds in three phases. First, it discovers AFDs using DAFDiscovery and removes non-duplicate AFDs using the pruning rule from Lemma 4.2 and Lemma 4.3 (Lines 2–5). Next, the algorithm iteratively filters inefficient AFDs using the RHS uniqueness property (Corollary 4.1) (Lines 6–14). It builds an inverted index $\mathcal{I}$ over the current AFD set Σ, grouping candidate AFDs by RHS attribute. Each entry $\mathcal{I}[A]$ contains triplets (X, l, u), where $X \xrightarrow{\varepsilon} A$ is a candidate incoming AFD and l, u are its lower and upper duplicate-gain bounds from Lemma 4.4. For each RHS attribute A, candidates whose upper bound is smaller than the maximum lower bound among candidates with the same RHS are pruned. If only one candidate remains for A, it is confirmed as the best incoming AFD for A, and all other AFDs with RHS A are removed from Σ. This process repeats until no further pruning is possible. This process converges in at most $|U|$ iterations, since each nontrivial iteration confirms at least one new RHS attribute, and there are at most $|U|$ attributes; otherwise, no further change occurs, and the loop terminates. After filtering, the remaining candidates in Σ are merged with the confirmed AFDs in C_{fd} to form the final candidate space (Line 14). Finally, it enumerates candidate subsets $\mathcal{F} \subseteq \Sigma$ (Lines 16–19). Only subsets satisfying $\sum_{i=1}^{k} |Y_i| + |R_{k+1}| - |U| = 0$ are considered, corresponding to the $a = 0$ condition in Theorem 4.1. For each valid subset, ComputeDG computes its exact duplicate gain, and the current best solution $(\mathcal{F}^*, dg^*)$ is updated online if a larger gain is found. The algorithm returns the valid subset with the maximum duplicate gain. ComputeDG identifies the maximal tuple

subset satisfying all AFDs in $\mathcal{F}$ and computes the exact duplicate gain; its pseudocode is provided in Algorithm 4.

Algorithm 4: ComputeDG($\mathcal{F}$, r)

```
Construct the equivalence dictionary E based on F and r;
T', T \ T' ← MaximalAFDTupleSet(r, F, E);
m_F ← |T'|;
dg ← 0;
foreach X_i -ε-> Y_i ∈ M(F) do
    d_F(X_i) ← number of distinct values of X_i in r[T'];
    dg ← dg + (m_F − d_F(X_i)) · |Y_i| − d_F(X_i) · |X_i|;
return dg;
```

Algorithm 5 runs in $O(n|\Sigma| + (\sum_{l=1}^{\min(|\Sigma|,n)} \binom{|\Sigma|}{l}) \cdot m^2(n + \log m))$ time in the worst case, where $n = |U|$, $m = |r|$. The $O(n|\Sigma|)$ cost comes from iterative filtering, which converges in at most n rounds. The remaining cost comes from enumerating candidate subsets of size at most $\min(|\Sigma|, n)$, as Corollary 5.1 implies that any valid AFD subset contains at most n dependencies, and evaluating each valid subset using ComputeDG. Since $|\mathcal{F}| \leq n$, each invocation of ComputeDG costs $O(m^2(n + \log m))$. The auxiliary space complexity is $O(m^2 + |\Sigma|)$, where the $O(|\Sigma|)$ accounts for storing the current candidate AFDs and the inverted index built over them.

Algorithm 5: ExactCore

```
Input: relation r over attribute set U;
Output: the core AFD set F*;
Initialize C_fd ← ∅, conf ← ∅, Σ_prev ← ∅, n ← |U|;
Σ ← DAFDiscovery(r);
foreach X_i such that ∃(X_i -ε-> A) ∈ Σ do
    if dg^up(Σ_{X_i}) < 0 then
        Remove all AFDs in Σ_{X_i} from Σ; // ExactCore_1
while Σ ≠ ∅ and Σ ≠ Σ_prev do
    Σ_prev ← Σ;
    I ← InvertedIndex(Σ, conf);
    foreach A ∈ U \ conf do
        I[A] ← {(X, l, u) | u ≥ max{l' | (X', l', u') ∈ I[A]}};
        if I[A] = {(X, l, u)} ;        // the only candidate for A
        then
            add (X -ε-> A) to C_fd and A to conf;
    Remove all AFDs from Σ whose RHS attribute belongs to conf;
Σ ← Σ ∪ C_fd; // ExactCore_2
Initialize F* ← ∅, dg* ← 0, S ← 0; // S_ExactCore
for length l ← 1; l ≤ min(|Σ|, n); l++ do
    foreach subset F ⊆ Σ with |F| = l do
        if Σ_{i=1}^{k} |Y_i| + |R_{k+1}| − |U| = 0 then
            Update F*, dg* if ComputeDG(F, r) > dg*; S++;
return F*;
```

Discussion: the performance of ExactCore. In Figure 4, efficiency could be further improved by enumerating merged dependencies $\mathrm{M}(\mathcal{F})$ instead of the original AFD set $\mathcal{F}$. However, this assumes that AFDs in $\mathcal{F}$ satisfy Armstrong's axioms in relation r, which may not hold in practice due to conflicting approximation errors. Therefore, we focus on exact enumeration over singleton-RHS AFDs. We also integrate RelaxRD's pruning rules into lattice-based AFD discovery to reduce the search space.

Table 1: Statistics of the public datasets.

Dataset	Attributes	Tuples	Domains
School [21]	27	14,384	Education
Adult [21]	15	32,561	Census demographics
NCVoter [30]	19	1,000,000	Voter registration
Fraud [36]	23	1,296,675	Financial transactions
CrimeLA [34]	26	1,048,574	Crime reports
DIGIX [26]	36	1,000,000	Recommendation
IndusData	15	4,000,000	Financial transactions
Iowa[7]	24	12,591,077	Retail sales

5 EXPERIMENTS

In this section, we evaluate RelaxRD on eight real-world datasets in terms of storage reduction, robustness, sensitivity, efficiency, and query performance.

5.1 Experiment Setup

5.1.1 Datasets. We evaluate RelaxRD on eight real-world datasets. Seven of them are publicly available datasets, while one is an industrial dataset that cannot be released due to privacy constraints. These datasets are selected to cover diverse application domains and data characteristics. In particular, School, Adult, and NCVoter are commonly used benchmarks in AFD discovery [12, 17, 21, 30], providing standard baselines. The description is in Table 1.

5.1.2 Baselines. We introduce baselines from five aspects (the top-down workflow of relation design): AFD discovery, AFD selection, logical schema design, physical storage models, and compression.
(1) AFD Discovery Baselines. To examine whether RelaxRD is robust to different AFD discovery algorithms, we evaluate it using AFD sets discovered by four representative methods: Tane [17], DAFDiscover [12], Hyfd [30], and Pyro [21].
(2) AFD Selection Baselines. To the best of our knowledge, RelaxRD is the first work to study dependency selection. Thus, to evaluate the importance of selecting high-quality AFDs for relation design, we compare ExactCore with two simple AFD selection baselines: Random, which selects k AFDs at random, and Minimal, which selects the k AFDs with the smallest error, where k is the size of the core AFD subset produced by RelaxRD.
(3) Storage Model Baselines. To evaluate whether the benefits of RelaxRD persist on different storage models, we evaluate storage under three storage models: row-store, column-store, and key–value store. We use PostgreSQL (abbr. PG) [38] and PolarDB (abbr. PD) [10] as row-store systems, DuckDB [33] as a column-store system, and RocksDB (abbr. RD) [13] as a key–value store system.
(4) Compression Baselines. To examine whether the storage benefits of RelaxRD remain effective under different compression techniques, we evaluate four representative compression methods: Zlib [24], LZ4 [22], ZSTD [45], and DuckDB's native compression.

5.1.3 Implementations. All experiments are conducted on a machine with an Apple M3 Ultra (28-core CPU, 96 GB unified memory). The implementations of AFD discovery algorithms are obtained from public repositories [12]. PostgreSQL 17.2, DuckDB v1.5.0, and RocksDB (via MariaDB 12.2.2) are used with default configurations. PolarDB 8.0.13 is deployed on an Alibaba Cloud instance with 2 CPU cores and 2 GB memory. Compression uses each system's built-in implementation. Unless otherwise specified, $\varepsilon = 0.05$.

Table 2: Storage size comparison (MB) across eight datasets under row-store, column-store, and KV-store settings.

Data	Method	Row-store			Col-store	KV-store		
		PD	PD-Zlib	PG	DuckDB	RD	RD-LZ4	RD-ZSTD
School	Large	4.81	2.41	4.62	**1.00**	3.35	1.57	1.13
	Random	8.66	3.82	8.36	1.50	4.42	2.04	1.43
	Minimal	8.47	3.73	8.21	1.70	4.85	2.33	1.69
	RelaxRD	**4.80**	**2.37**	**4.55**	1.30	**3.27**	**1.51**	**1.11**
	Savings	0.2%	1.7%	1.5%	-30.0%	2.4%	3.8%	1.8%
Adult	Large	**7.03**	**3.52**	**6.31**	**0.75**	4.24	1.50	1.16
	Random	10.13	4.55	8.23	1.20	5.42	1.92	1.47
	Minimal	7.09	3.55	6.55	1.00	**4.11**	**1.45**	**1.13**
	RelaxRD	7.09	3.55	6.45	1.00	**4.11**	**1.45**	**1.13**
	Savings	-0.9%	-0.9%	-2.2%	-33.3%	3.1%	3.3%	2.6%
NCVoter	Large	215.31	98.15	203.54	18.7	163.77	50.90	38.70
	Random	755.42	349.76	691.49	54.5	409.05	142.28	95.47
	Minimal	204.58	93.28	173.17	20.5	151.14	49.48	37.06
	RelaxRD	**126.23**	**57.61**	**103.48**	**17.7**	**75.67**	**32.01**	**22.35**
	Savings	41.4%	41.3%	49.2%	5.3%	53.8%	37.1%	42.2%
Fraud	Large	305.42	185.75	268.62	70.0	267.07	218.19	179.19
	Random	463.66	267.88	351.29	81.7	314.71	248.14	202.16
	Minimal	613.97	344.54	403.78	146.7	412.16	342.07	250.93
	RelaxRD	**185.00**	**102.00**	**136.91**	**47.2**	**128.08**	**95.53**	**77.90**
	Savings	39.4%	45.1%	49.0%	32.6%	52.0%	56.2%	56.5%
CrimeLA	Large	279.41	126.69	215.00	44.2	226.17	135.83	110.04
	Random	524.11	246.05	489.24	55.2	296.19	169.34	127.74
	Minimal	529.02	245.01	495.13	57.0	299.98	164.59	125.71
	RelaxRD	**189.64**	**86.31**	**124.07**	**38.5**	**139.79**	**89.69**	**71.62**
	Savings	32.1%	31.9%	42.3%	12.9%	38.2%	34.0%	34.9%
DIGIX	Large	279.42	126.70	222.43	28.0	213.08	83.89	57.43
	Random	295.25	134.59	301.42	29.2	211.67	83.28	57.89
	Minimal	260.89	118.44	210.73	26.7	196.89	76.87	53.97
	RelaxRD	**249.77**	**113.37**	**195.43**	**20.0**	**170.34**	**59.34**	**40.87**
	Savings	10.6%	10.5%	12.1%	28.6%	20.1%	29.3%	28.8%
IndusData	Large	556.64	253.32	452.00	75.2	356.17	147.91	92.26
	Random	656.98	302.48	597.08	87.5	375.95	178.52	109.01
	Minimal	880.81	407.43	781.2	151.0	480.57	258.93	154.85
	RelaxRD	**515.80**	**238.91**	**306.19**	**70.2**	**261.91**	**141.76**	**82.40**
	Savings	7.3%	5.7%	32.3%	6.6%	26.5%	4.2%	10.7%
Iowa	Large	4499.88	1851.94	3924.17	542.7	7373.52	5962.62	1685.50
	Random	4002.41	1810.72	3658.32	485.2	5992.51	3863.50	1438.74
	Minimal	3958.14	1791.05	3521.89	453.5	5934.86	2034.29	1541.24
	RelaxRD	**3007.59**	**1364.30**	**2361.74**	**329.0**	**3016.25**	**1794.78**	**1051.42**
	Savings	33.2%	26.3%	39.8%	39.4%	59.09%	69.90%	37.6%

PD-Zlib denote PolarDB with Zlib; RD-LZ4/-ZSTD denote RocksDB with LZ4/ZSTD. DuckDB uses its native compression. Savings = (Large − RelaxRD)/Large × 100%.

5.2 Storage Reduction Performance

Different AFD selection strategies. Table 2 reports the storage of the original relation (Large) and decompositions induced by different AFD selection strategies (Random, Minimal, core AFD) across eight datasets and four storage engines (PD, PG, RD, and DuckDB). We can find: (1) RelaxRD substantially reduces storage over Large, with the largest savings on Iowa (up to 59.09% under RocksDB) and smaller gains on low-redundancy datasets such as School and Adult. (2) Compared with Random and Minimal, RelaxRD achieves lower storage cost, indicating that neither random selection nor lowest-error AFDs is necessarily effective for storage-oriented schema design. (3) RelaxRD, as a schema design technique, remains effective across row-, column-, and KV-store systems.

Combined with compression techniques. Table 2 reports the storage of the original relation (Large) and RelaxRD under different compression settings across four storage engines. We observe that: (1) Compared with compressed Large, RelaxRD without compression can still achieve additional savings on some datasets. For example, on Fraud, it reduces storage by up to 41.3%, and on Iowa by 49.4% under RD-LZ4. However, on datasets with limited FD-based redundancy (e.g., DIGIX), RelaxRD may be worse than compressed Large. (2) When combined with compression, RelaxRD consistently yields smaller storage than compressed Large on most datasets. For example, under RD-LZ4, it achieves up to 69.9% reduction on Iowa, 56.5% on Fraud, and 42.2% on NCVoter. Similar trends are observed on PolarDB and DuckDB, while the benefit is limited on very small datasets such as School and Adult. This is because compression and RelaxRD reduce different forms of redundancy. Compression primarily shrinks byte-level duplicates, whereas RelaxRD removes schema-level redundancy by eliminating values that need not be repeatedly stored, while preserving the structural relationships among attributes. Thus, compression is better at shrinking bytes, whereas RelaxRD is more favorable for relational reasoning, consistency maintenance, and also inherits the benefits of classical normalization, such as reducing anomalies. They are orthogonal and can be combined for stronger storage reduction.

5.3 Sensitivity and Robustness Analysis

We examine how the decomposition results change under different approximation thresholds ε, and whether they remain stable when the input AFD set is generated by different discovery algorithms.

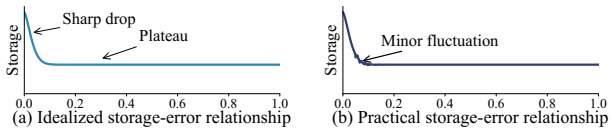


Figure 5: Trends among storage–error curve in RelaxRD.

5.3.1 Varying error thresholds ε. We vary ε from 0 to 0.2 and report, across five datasets, the number of discovered AFDs (#AFDs), the number of AFDs in the core AFD set ($|\mathcal{F}^*|$), the violating-tuple ratio $\frac{|r_0|}{|r|}$, the storage of RelaxRD (MB), and the resulting storage savings (Table 3). As the error threshold increases, the storage of RelaxRD follows an approximately monotonic decreasing trend: a sharp drop at small thresholds and then gradual stabilization (see Figure 5(a)). This is because: (1) At small error thresholds, only a small fraction of tuples violate the core AFD set, so most tuples can be decomposed by RelaxRD, leading to rapid storage reduction. (2) Low-error AFDs are more likely to capture meaningful semantic regularities rather than merely statistical pseudo-dependencies, and therefore reveal more removable duplicates, just like existing AFD discovery algorithms only focus on AFDs with small thresholds, such as 0.01 and 0.05 [12, 17, 21, 30]. (3) As ε increases, the candidate AFD space becomes larger, but the newly admitted AFDs usually have larger error and fewer decomposable tuples (fewer gain). As a result, the core AFD set remains largely dominated by low-error (more decomposable tuples) AFDs. Consequently, the $\mathcal{F}^*$ becomes increasingly similar to that under smaller thresholds, resulting in similar decompositions and thus gradual stabilization of the storage.

The storage is strictly non-increasing as ε increases if Σ_ε contains all AFDs whose error does not exceed ε (Figure 5(a)). To be

specific, for $\varepsilon_1 < \varepsilon_2$, let Σ_1 and Σ_2 denote the sets of all AFDs satisfying the two thresholds, respectively. Then $\Sigma_1 \subseteq \Sigma_2$. Let $\mathcal{F}_1^* \subseteq \Sigma_1$ and $\mathcal{F}_2^* \subseteq \Sigma_2$ be the corresponding optimal core AFD sets. Since RelaxRD minimizes storage cost, we have $C(\mathcal{F}_1^*) = \min_{\mathcal{F} \subseteq \Sigma_1} C(\mathcal{F})$ and $C(\mathcal{F}_2^*) = \min_{\mathcal{F} \subseteq \Sigma_2} C(\mathcal{F})$, which implies $C(\mathcal{F}_2^*) \leq C(\mathcal{F}_1^*)$. Therefore, the storage is non-increasing as ε increases. In practice, the storage is approximately non-increasing as ε grows, with only minor fluctuations (see Figure 5(b)). For instance, on the School dataset, the storage cost increases only slightly from 3.27 to 3.29 when ε increases from 0.05 to 0.1. This is because most existing AFD discovery algorithms [12, 17, 21, 30] adopt level-wise search over the attribute lattice with necessary early pruning. As a result, some AFDs that should become feasible at larger thresholds may be pruned before being fully explored, so the ideal inclusion $\Sigma_1 \subseteq \Sigma_2$ may not strictly hold. Consequently, RelaxRD may optimize over different candidate spaces and select slightly different core AFD sets, leading to minor storage fluctuations. For example, suppose $ABC \rightarrow D$ is selected under a smaller threshold ε_1, while under a larger threshold ε_2, a relaxed dependency $AB \rightarrow D$ becomes valid and prunes $ABC \rightarrow D$. If $AB \rightarrow D$ is then selected, the schema stores one fewer key attribute C, reducing schema storage but enlarging r_0 due to more violations. The final storage cost is determined by the trade-off between these two effects, which explains the minor fluctuations observed in practice.

5.3.2 Varying AFD discovery algorithms. To evaluate the robustness of RelaxRD with respect to different AFD discovery algorithms, i.e., Tane [17], DAFDiscovery [12], Hyfd [30], and Pyro [21], we first obtain the discovered AFD set Σ from each method, then run RelaxRD to select the core AFD subset $\mathcal{F}^*$ and perform the decomposition. Table 4 reports the number of discovered AFDs by AFD discovery algorithms, the size of the core AFD set $|\mathcal{F}^*|$ via RelaxRD, and the storage (MB) via RelaxRD. Overall, the core AFD sets selected by RelaxRD and the resulting decompositions are highly similar across different discovery algorithms and are often even the same. Small storage differences may still occur due to minor variations in Σ caused by different pruning strategies and search orders, but their impact remains limited (e.g., 169.87–170.65 MB on DIGIX and less than 1 MB on Iowa). These results indicate that RelaxRD is robust to the upstream AFD discovery algorithms.

5.4 Efficiency of RelaxRD

We use DAFDiscovery to extract AFDs. For large datasets, we first mine AFDs on a small sample and retain only those whose errors remain below 0.05 after validation on the full dataset.

Filtering power of ExactCore. Here, we evaluate the filtering power of the two strategies in ExactCore, namely ExactCore_1 and ExactCore_2. Table 5 reports the candidate sizes before and after filtering, together with the corresponding filtering rates. Overall, both strategies remove most candidate AFDs, while ExactCore_2 consistently achieves stronger pruning. For example, both methods filter out over 99% of candidates on School and Adult, and over 94% and 99% on NCVoter and Fraud, respectively. Its advantage becomes more evident on CrimeLA, IndusData, and Iowa.

Runtime of ExactCore. Figure 6(a), across all datasets, reports the runtime of ExactCore_x. Overall, both methods incur very low overhead. In particular, ExactCore_2 runs in only 0.0129 s on School and 0.0005 s on Adult, while remaining below 0.25 s on datasets such as CrimeLA and NCVoter. Even on larger datasets, the runtime remains small: on DIGIX, the two variants take 5.97 s and 3.83 s, respectively, and on Iowa only 5.26 s and 2.67 s. These results show that ExactCore introduces negligible runtime overhead.

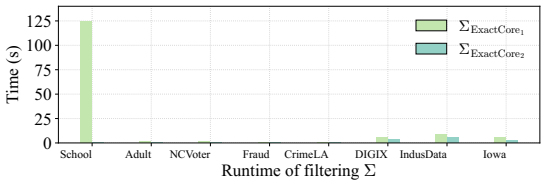


Figure 6: Runtime of filtering.

5.5 Query Performance.

To evaluate how RelaxRD affects end-to-end query latency in practical workloads, we construct a representative query workload over both the original large relation and the decomposed relations. We construct a workload of 12 representative SQL statements, including 3 point queries, 3 range queries, 3 analytical queries, and 3 write operations (INSERT, UPDATE, and DELETE), covering diverse data access patterns. The workload is generated with the assistance of ChatGPT using the domain description and attribute semantics of each dataset, together with explicit instructions on the desired query types, and require it to simulate realistic application requirements when generating queries. The model outputs SQL queries together with natural-language descriptions of the hypothetical business scenarios motivating them. All queries are first defined over the original (large) relation. We then develop a query transformation tool that automatically rewrites them into equivalent queries over the decomposed relations produced by RelaxRD, while preserving query semantics. The complete query workload and transformation tool are publicly available in our GitHub repository.

We evaluate the workload on two storage systems: PolarDB and RocksDB, comparing query latency between the original (Large) relation and the decomposed schema produced by RelaxRD. For PolarDB and RocksDB, we further consider multiple compressions to evaluate how schema decomposition interacts with compression mechanisms. The results are shown in Figures 7–8. Overall, the query latency of RelaxRD is determined by the trade-off between reduced I/O and the additional CPU overhead of join operations. Decomposition can improve performance by reducing data access, while the original large relation may be faster by avoiding join costs. To explain this trade-off, we model the latency difference of a query Q as $\Delta\text{Latency}(Q) = \text{Latency}_{\text{Large}}(Q) - \text{Latency}_{\text{RelaxRD}}(Q) \approx \alpha \cdot \Delta\text{IO}(Q) - \beta \cdot J(Q)$, where $\alpha, \beta > 0$ are scaling factors that map data access and computation cost to latency. Here, $\Delta\text{IO}(Q) = \text{IO}(r) - \text{IO}\big(\rho_{\text{needed}}(\mathcal{F}^*)\big)$ denotes the reduction in data access brought by decomposition, where $\rho_{\text{needed}}(\mathcal{F}^*)$ is the minimal set of decomposed relations required to answer Q. The term $J(Q)$ denotes the CPU cost of joining these relations. Thus, $\Delta\text{Latency}(Q) > 0$ indicates that RelaxRD is faster than the original relation. In particular, we divide workloads into three categories.

(1) *Join-free queries (see Q1–Q3 in Figure 7 and Figure 8).* These queries can be answered from a single decomposed relation and

Table 3: Sensitivity analysis under different error thresholds ε.

Dataset	$\varepsilon = 0$					$\varepsilon = 0.01$					$\varepsilon = 0.05$					$\varepsilon = 0.1$					$\varepsilon = 0.2$				
	#AFDs	$\|\mathcal{F}^*\|$	$\frac{\|r_0\|}{\|r\|}$	RelaxRD	% ↑	#AFDs	$\|\mathcal{F}^*\|$	$\frac{\|r_0\|}{\|r\|}$	RelaxRD	% ↑	#AFDs	$\|\mathcal{F}^*\|$	$\frac{\|r_0\|}{\|r\|}$	RelaxRD	% ↑	#AFDs	$\|\mathcal{F}^*\|$	$\frac{\|r_0\|}{\|r\|}$	RelaxRD	% ↑	#AFDs	$\|\mathcal{F}^*\|$	$\frac{\|r_0\|}{\|r\|}$	RelaxRD	% ↑
School	0	0	0.000	3.35	0.0%	40174	2	0.004	3.31	1.2%	55219	3	0.065	3.27	2.4%	46347	3	0.072	3.29	1.8%	53227	3	0.081	3.28	2.1%
Adult	0	0	0.000	4.24	0.0%	713	1	0.008	4.11	3.1%	884	1	0.008	4.11	3.1%	901	2	0.047	4.00	5.7%	1124	2	0.047	4.00	5.7%
Fraud	3044	12	0.000	144.37	45.9%	2973	14	0.011	128.55	51.9%	3012	14	0.013	128.08	52.0%	2291	14	0.013	128.08	52.0%	2253	14	0.013	128.08	52.0%
CrimeLA	131	6	0	139.79	38.2%	3227	6	0	139.79	38.2%	4399	6	0	139.79	38.2%	5820	8	0.032	139.88	38.2%	5937	8	0.032	139.88	38.2%
IndusData	24	4	0.000	331.52	6.9%	57	8	0.003	266.80	25.1%	96	8	0.013	261.91	26.5%	73	8	0.071	259.51	27.1%	72	9	0.108	235.14	34.0%

Table 4: Comparison across different AFD discovery methods ($\varepsilon = 0.05$). Storage is measured using RD (MB).

Methods	School			Adult			NCVoter			Fraud			CrimeLA			DIGIX			IndusData			Iowa		
	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD	#AFDs	$\|\mathcal{F}^*\|$	RelaxRD
Tane	55213	3	3.27	884	1	4.11	893	10	75.67	3078	14	128.08	4399	6	129.79	201	17	170.34	102	8	261.91	104	11	3016.25
DAFDiscovery	55219	3	3.27	884	1	4.11	897	10	75.67	3012	14	128.08	4399	6	129.79	205	17	170.34	96	8	261.91	106	11	3016.25
Hyfd	55734	3	3.27	901	1	4.11	814	10	75.67	3056	14	128.08	4075	6	129.83	198	17	170.65	105	8	262.34	113	11	3016.25
Pyro	55184	3	3.74	891	1	4.11	902	10	76.12	2987	14	127.56	4413	6	130.41	211	17	169.87	101	8	262.44	109	11	4731.88

Table 5: Filter power with filtering rate.

Datasets	Σ	ExactCore$_1$		ExactCore$_2$	
	$\|\Sigma\|$	$\|\Sigma\|_{\text{ExactCore}_1}$	$\frac{\|\Sigma\|-\|\Sigma\|_{\text{ExactCore}_1}}{\|\Sigma\|}$	$\|\Sigma\|_{\text{ExactCore}_2}$	$\frac{\|\Sigma\|-\|\Sigma\|_{\text{ExactCore}_2}}{\|\Sigma\|}$
School	55,219	31	99.9%	13	99.9%
Adult	884	1	99.9%	1	99.9%
NCVoter	897	51	94.3%	48	94.6%
Fraud	3,012	28	99.1%	27	99.1%
CrimeLA	4,399	27	69.7%	6	93.3%
DIGIX	205	43	79.0%	38	81.5%
IndusData	96	45	53.1%	27	71.9%
Iowa	106	63	40.6%	24	77.4%

therefore incur no join overhead, i.e., $J(Q) = 0$. Hence, $\Delta\text{Latency}(Q) \approx \alpha \cdot \Delta\text{IO}(Q)$. Since $\Delta\text{IO}(Q) = \text{IO}(r) - \text{IO}(\rho_{\text{needed}}(\mathcal{F}^*)) \geq 0$, any single decomposed relation is no larger than the original relation r, and join-free queries typically achieve lower latency under RelaxRD. Therefore, join-free queries typically achieve lower latency under RelaxRD, regardless of whether they are point, range, or analytical queries. (2) *Join-based queries (see Q4–Q9 in Figure 7 and Figure 8).* For these queries, latency is determined by the trade-off between I/O reduction $\Delta\text{IO}(Q)$ and join cost $J(Q)$. We consider three representative cases. ① Point queries. Point queries are highly selective and produce only small intermediate results, so the join cost $J(Q)$ is typically negligible. Since decomposition still reduces data access, $\Delta\text{IO}(Q)$ remains positive. Hence, $\Delta\text{IO}(Q) \gg J(Q)$ typically holds, and RelaxRD usually achieves lower latency. ② Range queries. Range queries have moderate selectivity and produce larger intermediate results, making $J(Q)$ non-negligible. At the same time, $\Delta\text{IO}(Q)$ remains beneficial due to reduced redundancy. As a result, latency depends on the balance between these two terms, and RelaxRD may either outperform or slightly underperform the original relation. ③ Analytical queries. Analytical queries, such as low-selectivity scans or full-table aggregations, often involve large intermediate results and access most decomposed relations. In this case, the I/O reduction is limited while the join cost becomes dominant. Therefore, $J(Q) \gg \Delta\text{IO}(Q)$ typically holds, and the original large relation generally achieves lower latency. (3) *Write operations (see Q10–Q12 in Figure 7 and Figure 8).* ① INSERT (Q10). INSERT is efficient under RelaxRD, since a new tuple is directly appended to r_0 without requiring joins or tuple reconstruction. ② UPDATE (Q11). UPDATE first requires locating the target tuple, and therefore follows the same cost pattern as the query cases analyzed above: its latency depends on whether tuple identification is join-free or join-based. Once identified, the modified tuple is written into r_0. ③ DELETE (Q12). DELETE also first requires locating the target tuple and thus follows the same identification cost pattern as UPDATE. After identification, deletion is applied only to r_0 and r_{k+1}, since these two relations preserve tuple identity through the original primary key. Other decomposed relations are refreshed periodically rather than directly updated to avoid anomalies and ensure consistency.

6 RELATED WORKS

Traditional Schema Design. Database normalization is the classical foundation of logical schema design in the relational model [16, 37, 39, 40, 43, 47, 48]. Normalization restructures database schemas into progressively stricter normal forms, enforcing functional dependencies as constraints to eliminate redundancy and guarantee correctness [18, 20, 23, 41]. Specifically, First Normal Form (1NF) eliminates nested relations and multivalued attributes by requiring that each attribute contains a single, atomic value. Second Normal Form (2NF) [4] eliminates partial dependencies by requiring that every non-key attribute be fully functionally dependent on the entire candidate key. Third Normal Form (3NF) [6] removes transitive dependencies by requiring that every non-key attribute be functionally dependent on a key, while Boyce–Codd Normal Form (BCNF) [3] further strengthens this requirement by ensuring that the left-hand side of every functional dependency is a key. Recently, Zhang et al. [49] proposed a 3NF schema synthesis approach that parameterizes candidate schemata based on the number of keys and non-key dependencies to reduce maintenance overhead while preserving all dependencies. These classical normal forms guarantee logical consistency and maintainability.

Functional Dependency Discovery. A large body of research has focused on identifying exact functional dependencies [15, 17, 27–30]. However, these methods become infeasible when real-world data contains some tuples that violate exact FDs. Recently, approximate functional dependency (AFD) discovery techniques have been proposed [11, 21, 28, 32]. These methods relax dependency conditions by allowing limited violations or errors [25, 31, 35], enhancing their robustness in noisy yet real datasets. These works can

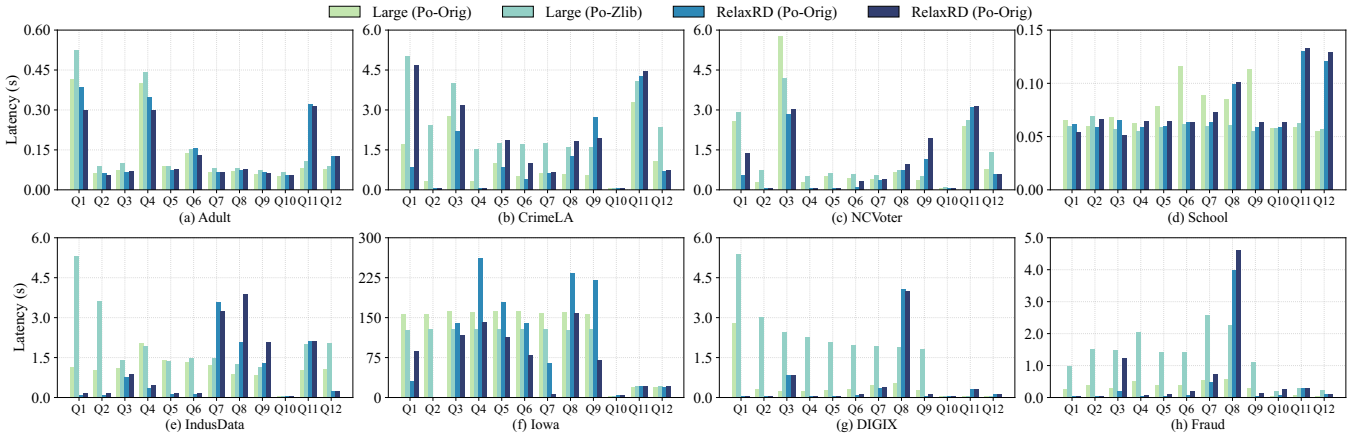


Figure 7: Latency comparison between Large and RelaxRD on PolarDB under different compressions (PD and PD-Zlib).

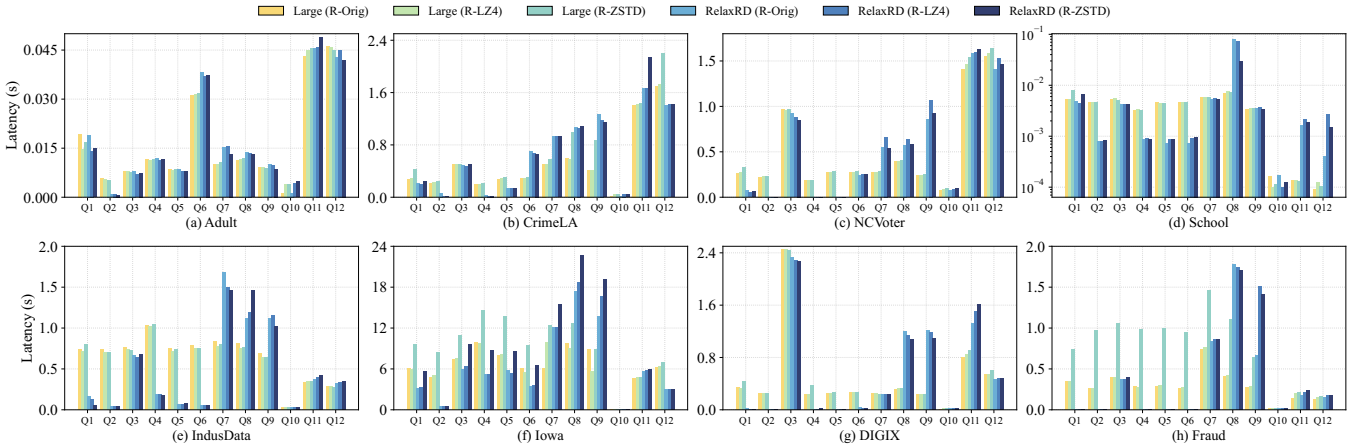


Figure 8: Latency comparison between Large and RelaxRD on RocksDB under different compressions (Orig, LZ4, and ZSTD).

be classified as follows. (i) *Lattice-based approaches*, which model the search space as a power set lattice of attribute combinations and employ a level-wise bottom-up traversal strategy that builds upon the *apriori-gen* candidate generation principle [1], e.g., Tane [17] and DAFDiscovery [12]. They generate new AFD candidates and validate them using a stripped equivalence partition. Notably, their pruning rules usually exploit the minimality criterion of AFDs and have significant overlap [29]: almost all algorithms in lattice-based methods use and extend the pruning rules of Tane. (ii) *Probabilistic-based approaches*, which model the distribution that AFDs impose over pairs of tuples and leverage dependencies to recover AFDs [14], e.g., FDX [46], and RFI [28]. However, the number of AFDs can be extremely large, and not all AFDs are useful.

7 CONCLUSION AND FUTURE WORKS

In this paper, we present RelaxRD, a storage-centric schema design approach that leverages high-quality AFDs to reduce redundancy. RelaxRD constructs schemas from AFDs, models violating tuples via a conflict graph, and formulates tuple selection as a maximal independent set problem. We further develop an efficient algorithm, ExactCore, to select core AFDs for decomposition. Experiments demonstrate that RelaxRD achieves substantial storage reduction, making it a complement to classical normalization.

Discussion and future directions. RelaxRD suggests a broader shift in relational schema design: from rule-based normalization to cost-driven design. Traditional schema design focuses on logical validity, while future schema design may explicitly optimize physical objectives such as storage, maintenance, and query performance. Second, AFD mining may also shift from discovery-driven to objective-driven. Rather than exhaustively finding statistically valid dependencies, future methods may focus on dependencies that are useful for downstream tasks such as schema optimization.

ACKNOWLEDGMENTS

The work was supported by the National Key Research and Development Program of China (No. 2024YFF0617702); the National Natural Science Foundation of China (Nos. U22A2025, 62232007, and U23A20309); the 111 Project (No. B16009); and the Ant Group Research Program (No. 2025021900003).

REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. 1994. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the 20th International Conference on Very Large Data Bases, VLDB*. Morgan Kaufmann, 487–499.

[2] Munqath Alatar and Attila Sali. 2024. Approximate Keys and Functional Dependencies in Incomplete Databases With Limited Domains-Algorithmic Perspective. *CoRR* abs/2402.05161 (2024). arXiv:2402.05161

[3] Marcelo Arenas. 2009. Boyce-Codd Normal Form. In *Encyclopedia of Database Systems*. Springer US, 264–265.

[4] Marcelo Arenas. 2018. Second Normal Form (2NF). In *Encyclopedia of Database Systems, Second Edition*. Springer.

[5] Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. 1984. On the Structure of Armstrong Relations for Functional Dependencies. *J. ACM* 31, 1 (1984), 30–46.

[6] Philip A. Bernstein. 1976. Synthesizing Third Normal Form Relations from Functional Dependencies. *ACM Trans. Database Syst.* 1, 4 (1976), 277–298.

[7] Aleksey Bilogur. 2018. Iowa Liquor Sales. https://www.kaggle.com/datasets/residentmario/iowa-liquor-sales. Kaggle dataset, accessed 2025.

[8] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. 2020. Hitting Set Enumeration with Partial Information for Unique Column Combination Discovery. *Proceedings of the VLDB Endowment, PVLDB* 13, 11 (2020), 2270–2283.

[9] Tobias Bleifuß, Susanne Bülow, Johannes Frohnhofen, Julian Risch, Georg Wiese, Sebastian Kruse, Thorsten Papenbrock, and Felix Naumann. 2016. Approximate Discovery of Functional Dependencies for Large Datasets. In *Proceedings of the 25th ACM International Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. ACM, 1803–1812.

[10] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *SIGMOD '21: International Conference on Management of Data*. ACM, 2477–2489.

[11] Xiaoou Ding, Yingze Li, Hongzhi Wang, Chen Wang, Yida Liu, and Jianmin Wang. 2024. TSDDISCOVER: Discovering Data Dependency for Time Series Data. In *Proceedings of the 40th IEEE International Conference on Data Engineering, ICDE*. IEEE, 3668–3681.

[12] Xiaoou Ding, Yixing Lu, Hongzhi Wang, Chen Wang, Yida Liu, and Jianmin Wang. 2024. DAFDiscover: Robust Mining Algorithm for Dynamic Approximate Functional Dependencies on Dirty Data. *Proceedings of the VLDB Endowment, PVLDB* 17, 11 (2024), 3484–3496.

[13] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4 (2021), 26:1–26:32.

[14] Liang Duan, Xinran Wu, Xinhui Li, Lixing Yu, and Kun Yue. 2025. Probabilistic Semantics Guided Discovery of Approximate Functional Dependencies. In *Conference on Uncertainty in Artificial Intelligence (Proceedings of Machine Learning Research)*, Silvia Chiappa and Sara Magliacane (Eds.), Vol. 286. PMLR, 1121–1134.

[15] Peter A. Flach and Iztok Savnik. 1999. Database Dependency Discovery: A Machine Learning Approach. *AI Commun.* 12, 3 (1999), 139–160.

[16] Daokun Hu, Quanqing Xu, and Chuanhui Yang. 2025. OLTP Engines on Modern Storage Architectures. In *Companion of the 2025 International Conference on Management of Data (SIGMOD-Companion '25)*.

[17] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An Efficient Algorithm for Discovering Functional and Approximate Dependencies. *Comput. J.* 42, 2 (1999), 100–111.

[18] Batya Kenig, Pranay Mundra, Guna Prasaad, Babak Salimi, and Dan Suciu. 2020. Mining Approximate Acyclic Schemes from Relations. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD*. ACM, 297–312.

[19] Heekyong Kim. 2024. Samsung hikes memory chip prices by up to 60% as shortage worsens. *Reuters* (November 2024).

[20] Henning Köhler and Sebastian Link. 2016. SQL Schema Design: Foundations, Normal Forms, and Normalization. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. ACM, 267–279.

[21] Sebastian Kruse and Felix Naumann. 2018. Efficient Discovery of Approximate Dependencies. *Proceedings of the VLDB Endowment, PVLDB* 11, 7 (2018), 759–772.

[22] Maximilian Kuschewski, David Sauerwein, Adnan Alhomssi, and Viktor Leis. 2023. BtrBlocks: Efficient Columnar Compression for Data Lakes. *Proc. ACM Manag. Data* 1, 2 (2023), 118:1–118:26.

[23] Sebastian Link and Ziheng Wei. 2021. Logical Schema Design that Quantifies Update Inefficiency and Join Efficiency. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD*. ACM, 1169–1181.

[24] Chunwei Liu, Anna Pavlenko, Matteo Interlandi, and Brandon Haynes. 2023. A Deep Dive into Common Open Formats for Analytical DBMSs. *Proc. VLDB Endow.* 16, 11 (2023), 3044–3056.

[25] Jinqi Liu, Anzhen Zhang, Jiajia Li, Na Guo, and Jing Zhang. 2024. Approximate Functional Dependencies Discovery Using Markov Blanket. In *Proceedings of Workshops at the 50th International Conference on Very Large Data Bases*.

[26] Chen Lixi. 2020. 2020 DIGIX Advertisement CTR Prediction. https://www.kaggle.com/datasets/lousichen/2020-digix-advertisement-ctr-prediction.

[27] Stéphane Lopes, Jean-Marc Petit, and Lotfi Lakhal. 2000. Efficient Discovery of Functional Dependencies and Armstrong Relations. In *Proceedings of the 7th International Conference on Extending Database Technology EDBT (Lecture Notes in Computer Science)*, Vol. 1777. Springer, 350–364.

[28] Panagiotis Mandros, Mario Boley, and Jilles Vreeken. 2017. Discovering Reliable Approximate Functional Dependencies. In *Proceedings of the 23rd ACM International Conference on Knowledge Discovery and Data Mining*. ACM, 355–363.

[29] 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. *Proceedings of the VLDB Endowment, PVLDB* 8, 10 (2015), 1082–1093.

[30] Thorsten Papenbrock and Felix Naumann. 2016. A Hybrid Approach to Functional Dependency Discovery. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD*. ACM, 821–833.

[31] Marcel Parciak, Sebastiaan Weytjens, Niel Hens, Frank Neven, Liesbet M. Peeters, and Stijn Vansummeren. 2024. Measuring Approximate Functional Dependencies: A Comparative Study. In *Proceedings of the 40th IEEE International Conference on Data Engineering, ICDE*. IEEE, 3505–3518.

[32] Frédéric Pennerath, Panagiotis Mandros, and Jilles Vreeken. 2020. Discovering Approximate Functional Dependencies using Smoothed Mutual Information. In *Proceedings of the 26rd ACM International Conference on Knowledge Discovery and Data Mining, SIGKDD*. ACM, 1254–1264.

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD*. ACM, 1981–1984.

[34] Shubhra Rana. 2023. CrimeDataSetLA. Kaggle Dataset. https://www.kaggle.com/datasets/shubhrarana/crimedatasetla Derived from Los Angeles Open Data.

[35] Sanjivni Rana, Junya Ogawa, Suraj Shetiya, Senjuti Basu Roy, and Gautam Das. 2025. Anytime Algorithms for Approximate Functional Dependencies. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. ACM, 2386–2397.

[36] Kartik Shenoy. 2019. Credit Card Transactions Fraud Detection Dataset. https://www.kaggle.com/datasets/kartik2112/fraud-detection. Kaggle dataset.

[37] Avi Silberschatz, Henry F. Korth, and S. Sudarshan. 2011. *Database System Concepts, Sixth Edition*. McGraw-Hill Book Company.

[38] Michael Stonebraker and Lawrence A. Rowe. 1986. The Design of Postgres. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*. ACM Press, 340–355.

[39] Regina P. Ticona-Herrera, Joe Tekli, Richard Chbeir, Sébastien Laborie, Irvin Dongo, and Renato Guzman. 2015. Toward RDF Normalization. In *Proceedings of the 34th International Conference on Conceptual Modeling ER (Lecture Notes in Computer Science)*, Vol. 9381. Springer, 261–275.

[40] Millist W. Vincent, Jixue Liu, and Chengfei Liu. 2004. Strong functional dependencies and their application to normal forms in XML. *ACM Trans. Database Syst.* 29, 3 (2004), 445–462.

[41] Ziheng Wei and Sebastian Link. 2019. Embedded Functional Dependencies and Data-completeness Tailored Database Design. *Proceedings of the VLDB Endowment, PVLDB* 12, 11 (2019), 1458–1470.

[42] Monica J. White. 2025. It's Not Just RAM—SSDs Could Soon Cost Way More Too, and It's All Downhill from Here. *PC Gamer* (2025).

[43] Quanqing Xu, Chuanhui Yang, and Aoying Zhou. 2024. Native Distributed Databases: Problems, Challenges and Opportunities. *Proc. VLDB Endow.* 17, 12 (2024), 4217–4220.

[44] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million tpmC Distributed Relational Database System. *Proceedings of the VLDB Endowment, PVLDB* 15, 12 (2022), 3385–3397.

[45] Feng Zhang, Weitao Wan, Chenyang Zhang, Jidong Zhai, Yunpeng Chai, Haixiang Li, and Xiaoyong Du. 2022. CompressDB: Enabling Efficient Compressed Data Direct Processing for Various Databases. In *SIGMOD '22: International Conference on Management of Data*. ACM, 1655–1669.

[46] Yunjia Zhang, Zhihan Guo, and Theodoros Rekatsinas. 2020. A Statistical Perspective on Discovering Functional Dependencies in Noisy Data. In *Proceedings of the 2020 International Conference on Management of Data*. ACM, 861–876.

[47] Zhuoxing Zhang, Wu Chen, and Sebastian Link. 2023. Composite Object Normal Forms: Parameterizing Boyce-Codd Normal Form by the Number of Minimal Keys. *Proc. ACM Manag. Data* 1, 1 (2023), 13:1–13:25.

[48] Zhuoxing Zhang and Sebastian Link. 2025. Mixed covers: optimizing updates and queries using minimal keys and functional dependencies. *VLDB J.* 34, 3 (2025), 29.

[49] Zhuoxing Zhang and Sebastian Link. 2025. Synthesizing Third Normal Form Schemata that Minimize Integrity Maintenance and Update Overheads: Parameterizing 3NF by the Numbers of Minimal Keys and Functional Dependencies. *Proc. ACM Manag. Data* 3, 3 (2025), 225:1–225:25.