Semantically Meaningful Sample Databases for Constraint Verification

A thesis presented in partial fulfilment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science

at Massey University, Manawatu,
New Zealand

Wendy(Weimin) YE

2026

Contents

1 Introduction **1**
- 1.1 Basic Definitions . . . 1
- 1.2 Armstrong Database . . . 3
- 1.3 Additional Concepts . . . 5
- 1.4 Related work . . . 7
- 1.5 Broader Context within Database and Data Management Research . . . 9
- 1.6 Research Aims and Objectives . . . 9

2 Foundations **11**
- 2.1 Existence of Armstrong Databases . . . 11
- 2.2 Construction of Armstrong database . . . 12
  - 2.2.1 Armstrong databases for keys . . . 13
  - 2.2.2 Armstrong Databases for Functional Dependencies . . . 13
  - 2.2.3 Armstrong Databases for Inclusion Dependencies (INDs) . . . 16
  - 2.2.4 Armstrong Databases for Multiple Classes of Dependencies . . . 16
- 2.3 NULL Values . . . 18
  - 2.3.1 NULL Values and Keys . . . 18
  - 2.3.2 NULL Values and Functional Dependencies . . . 19

3 Combining partial Armstrong databases **20**
- 3.1 Disjoint Union . . . 20
- 3.2 Proof of faithfulness with respect to Disjoint Union . . . 22

4 Small Armstrong tables for keys and functional dependencies **29**
- 4.1 Build Armstrong table . . . 29
  - 4.1.1 Armstrong Table for Keys . . . 29
  - 4.1.2 Armstrong Tables for Functional Dependencies (FDs) . . . 30
- 4.2 Build Armstrong Table in Sublinear Size . . . 31

5 Implementation for Sublinear Armstrong Table **35**
- 5.1 Basic Algorithm . . . 35
- 5.2 Alternative paradigm . . . 40
- 5.3 Optimization efforts . . . 40
- 5.4 Complexity Analysis and Heuristic approaches . . . 42
  - 5.4.1 Theoretical Complexity and Algorithm Analysis . . . 42
  - 5.4.2 Heuristic approaches . . . 43
- 5.5 Experiments . . . 45
  - 5.5.1 Experiment 1: Single Dataset Analysis . . . 46
  - 5.5.2 Experiment 2: Cross-Dataset Evaluation . . . 46
  - 5.5.3 Experiment 3: Evaluation on Randomly Generated Tables . . . 47
  - 5.5.4 Summary . . . 48

6 Implementation for Informative Sublinear Armstrong Table **49**
6.1 Algorithm principle . . . 49
6.2 Methods to eliminate duplicates . . . 51
6.2.1 Edge Number (EN) . . . 53
6.2.2 Progressive Vertex Expansion (PVE) . . . 54
6.2.3 Edge Rank (ER) . . . 55
6.2.4 Weighted Edge Rank (WER) . . . 56
6.2.5 Weighted Vertex Rank (WVR) . . . 57
6.2.6 Domination . . . 58
6.3 Experiments . . . 60

7 Keys with NULLs **62**
7.1 Possible keys and certain keys . . . 62
7.2 Building Armstrong tables for possible and certain keys . . . 64
7.2.1 Existence of Armstrong table . . . 64
7.2.2 Constructing Armstrong table . . . 66

8 Sublinear Armstrong table for possible and certain keys **69**
8.1 Algorithm Principle . . . 70
8.2 Approaches for Finding Sublinear Armstrong Tables for Possible and Certain Keys . . . 73
8.3 Experiments . . . 74

9 Informative Sublinear Armstrong Tables for Possible and Certain Keys **76**
9.1 Algorithm for Constructing Smaller Informative Armstrong Tables with `NULL` Values . . . 76
9.2 Strategies for removing duplicates . . . 77
9.2.1 Edge Selection . . . 77
9.2.2 Vertex-Centric Strategy . . . 78
9.2.3 Agree Set Occurrence Frequency . . . 78
9.2.4 Vertex Metric . . . 79
9.3 Experiments . . . 81

10 Conclusions and Future Directions **82**
10.1 Discussion and Reflection on Research Aims . . . 83
10.2 Future Research Directions . . . 84

Appendix A: Glossary **85**

Abstract

To ensure high-quality relational database design, theoretical developments are crucial, particularly in measuring constraints. Keys, functional dependencies (FDs), and inclusion dependencies (INDs) play pivotal roles and are practical for conveying data semantics in relational databases. However, ensuring the validity of constraints can be challenging when specified manually or mined from existing database instances.

Example databases serve as valuable tools for inspecting the properties of proposed designs. Known in the literature as Armstrong databases, these databases precisely adhere to a specified set of dependencies and their logical consequences, excluding any additional dependencies not implied by them.

Research efforts have focused on proving the existence and exploring the structure of Armstrong databases. The size of Armstrong databases is equally important; larger databases can be impractical as overlooked constraints become harder to detect. Moreover, when NULL values are involved, constructing Armstrong databases becomes more complex, requiring additional steps. Therefore, constructing a compact Armstrong database is challenging but essential for practical utility.

Our study aims to improve existing methods by investigating and constructing Armstrong databases from scratch. This includes adapting existing real-world databases into Armstrong databases to enhance their utility and effectiveness in practical applications. By focusing on reducing the size of Armstrong databases, we aim to streamline the process of constraint validation and database design optimization.

This research contributes to advancing the field by addressing the practical challenges of database constraint management and fostering the development of more efficient and reliable relational database designs.

Chapter 1

Introduction

Relational databases are built upon the relational model, first proposed by Edgar Frank Codd in 1970[1], which revolutionized the way data is structured and queried. In this model, data is organized into tables (relations) consisting of rows (tuples) and columns (attributes). Each row represents a unique record, typically identified by a key, while each column defines a particular attribute of the data. The tabular organization of data, along with a formal mathematical foundation, makes relational databases a powerful and widely adopted tool for representing and manipulating structured information. Relational databases are engineered to reduce data redundancy and maintain data integrity, thereby enabling efficient storage, retrieval, and management of large volumes of structured data.

In the traditional paradigm of relational database design, the process typically begins with database designers manually determining the attributes of interest and specifying data dependencies among them[2]. These dependencies guide the schema design and ensure data consistency and integrity. However, in many practical settings, such as legacy systems or rapidly evolving domains, such dependencies may be incomplete, or entirely missing. In such cases, dependencies may be discovered (or mined) from existing database instances through automated data profiling and constraint discovery techniques. Still, challenges remain - valid dependencies may be overlooked, which can negatively affect the efficiency, accuracy, and reliability of the database system.

To address these challenges, the concept of Armstrong databases provide a robust and interpretable representation of dependency sets. Named after William Ward Armstrong, who introduced the well-known Armstrong's axioms for reasoning about FDs in 1974[3], Armstrong databases offer a model-theoretic framework for illustrating the implications of a given set of dependencies. The formal concept of an Armstrong relation - a special kind of relation that satisfies exactly the dependencies entailed by a given set - was introduced by Ronald Fagin in 1980[4]. Further foundational work by Fagin, Beeri, and Howard in the late 1970s and early 1980s helped establish the theoretical basis for Armstrong-style reasoning in relational databases. In its simplest form, an Armstrong relation is a single-table instance that exactly captures the logical consequences of a given set of constraints - no more, no less.

1.1 Basic Definitions

Database constraints are essential mechanisms for preserving the integrity, accuracy, and consistency of data within relational database systems. These constraints enforce rules on the data stored in tables to ensure that it adheres to the logical structure and business semantics intended by database designers. By automating data validation, constraints reduce the potential for human error during data insertion and updates, helping maintain data quality over time. Moreover, they play a crucial role in ensuring logical consistency, supporting query optimization, and facilitating schema design by defining how data can be structured and interrelated.

Below, we introduce the most commonly used types of constraints and some related definitions:

Definition 1.1.1 (Structured Query Language/SQL)**.** Structured Query Language (SQL) is a domain-specific language designed for managing and querying data in relational database systems. SQL provides a declarative syntax for specifying data retrieval, insertion, deletion, and update operations, as well as defining and enforcing

schema-level constraints such as keys, data types, and integrity rules. It is especially well-suited for working with structured data, where relationships among data items are expressed through attributes and relational operators. △

Definition 1.1.2 (Key)**.** In a SQL table schema, a key is a set of one or more attributes that uniquely identify each tuple in a relation. A candidate key ensures that no two tuples have the same combination of values across the key attributes. SQL supports UNIQUE constraints, which enforce that values in specified columns (or combinations of columns) are distinct across all rows. A Primary Key is a special kind of candidate key that must additionally satisfy a NOT NULL constraint, ensuring every record has a value for the key attributes. A table can have multiple UNIQUE constraints but only one Primary Key, reflecting the main identifier for tuples within the relation. △

Definition 1.1.3 (Functional Dependency/FD)**.** A functional dependency(FD) describes a relationship between two sets of attributes in a database: the determinant and the dependent. Specifically, for a functional dependency, the determinant uniquely determines the values of the dependent. Let R be a relation schema, and let $X, Y \subseteq R$. The FD $X \to Y$ holds in a relation r over R if, for any two tuples $t_1, t_2 \in r$, $t_1[X] = t_2[X]$ implies $t_1[Y] = t_2[Y]$. Here, X is the determinant, and Y is the dependent. Functional dependencies generalize the concept of keys: a key is essentially a determinant that functionally determines all other attributes in the relation[5]. △

Definition 1.1.4 (Lossless Decomposition)**.** A decomposition of a relation r into sub-relations r_1 and r_2 is said to be lossless if the natural joining of r_1 and r_2, denoted $r_1 \bowtie r_2$, reconstructs the original relation exactly, i.e., $r = r_1 \bowtie r_2$. Lossless decomposition is crucial for ensuring that no information is lost during normalization or schema restructuring. Conversely, a lossy decomposition introduces spurious tuples not present in the original relation. Table 1.1 below demonstrates such a counterexample where extra tuples are produced[5].

Original relation r:

A	B	C
1	3	5
1	3	6
2	3	6
2	4	6

Decomposed relations:

r_1:

A	B
1	3
2	3
2	4

r_2:

B	C
3	5
3	6
4	6

$r_1 \bowtie r_2$:

A	B	C
1	3	5
1	3	6
2	3	5
2	3	6
2	4	6

Table 1.1: Counter-example of Lossless Decomposition

△

Definition 1.1.5 (Multi-valued Dependency/MVD)**.** A multi-valued dependency (MVD) captures the idea that sets of values in one part of a relation can vary independently of values in another part, given a fixed set of attributes. Formally, for a relation schema R, the MVD $X \twoheadrightarrow Y$ holds in a relation r if the decomposition of r into projections on XY and $X(R \backslash Y)$ is lossless. This is formalized as follows: for all $t, t' \in r$ with $t[X] = t'[X]$ there exist rows $u, u' \in r$ such that:

$$\begin{array}{ll} u[X] = t[X], & u'[X] = t'[X]; \\ u[Y] = t[Y], & u'[Y] = t'[Y]; \text{ and} \\ u[R \backslash XY] = t'[R \backslash XY], & u'[R \backslash XY] = t[R \backslash XY]. \end{array}$$

MVDs generalize FDs: every FD $X \to Y$ is also an MVD $X \twoheadrightarrow Y$, but not vice versa[5].

△

Definition 1.1.6 (Join Dependency/JD)**.** A join dependency (JD)is a generalization of both MVDs and FDs. For a relation schema U, a JD is an expression of the form $\bowtie [X_1, ..., X_n]$, where each $X_i \subseteq U$ and $\bigcup_{i=1}^{n} X_i = U$. A relation r over U satisfies $\bowtie [X_1, ..., X_n]$ if $r = \bowtie_{i=1}^{n} \{\pi x_i(r)\}$, i.e., the natural join of the projections over $X_1, ..., X_n$ reconstructs r exactly. A JD σ is n-ary if the number of attribute sets involved in σ is n, while a MVDs is a special case of JD (2-ary JD)[6]. △

Definition 1.1.7 (Inclusion Dependency/IND)**.** An inclusion dependency (IND) expresses that values appearing in one relation must also appear in another, capturing referential integrity. Let R and S be relation schemes. and $X = \{A_1, ..., A_n\} \subseteq R$, $Y = \{B_1, ..., B_n\} \subseteq S$. The IND $R[X] \subseteq S[Y]$holds if, for every tuple $t \in r$ (an instance of R), there exists a tuple $t' \in s$ (an instance of S) such that $t[A_i] = t'[B_i]$ for all $1 \leq i \leq n$[5]. INDs are foundational in defining foreign key constraints, which ensure consistency across related tables and are especially important in multi-relational database systems. △

Definition 1.1.8 (Closure)**.** Let R be a relation schema and Σ a set of functional dependencies (FDs) over R. For a set of attributes $X \subseteq R$, the *closure* of X under Σ, denoted by X^* (or X^+), is the set of all attributes $A \in R$ such that $\Sigma \vDash X \to A$, i.e., $X \to A$ is logically implied by Σ[5].

More generally, the *closure* of a set of constraints Σ, denoted Σ^* (or Σ^+), is the set of all constraints that are logically implied by Σ.

Let $\Sigma = \{A \to B, B \to C\}$ be a set of FDs over the relation schema $R(A, B, C)$. The dependency $A \to B$ indicates that the value of attribute A uniquely determines the value of B, and $B \to C$ implies that B uniquely determines C. From these two dependencies, it follows by transitivity that A also uniquely determines C, i.e., $A \to C$. Therefore, the closure of Σ, denoted Σ^*, includes all functional dependencies logically implied by Σ:

$$\Sigma^* = \{A \to B, B \to C, A \to C\}.$$

△

To support the formal definitions and discussions throughout the chapters, a list of key symbols is included in Appendix A.

1.2 Armstrong Database

Armstrong databases are powerful conceptual and practical tool in the design and analysis of relational database schemas, particularly for understanding and managing data dependencies. Their primary utility lies in providing a minimal and exact representation of a given set of constraints along with their logical consequences. By systematically constructing a database that satisfies only the dependencies logically implied by a given set, Armstrong databases enable designers to isolate and explore the precise semantics of constraint sets. This makes them indispensable in tasks such as constraint validation.

Definition 1.2.1 (Armstrong Database and Armstrong Table)**.** An *Armstrong database* is a synthetic database instance that satisfies *exactly* those dependencies that are logically implied by a given set Σ, i.e., its closure Σ^*, and violates all other dependencies of the same type that are *not* implied by Σ [4]. Formally, for a database schema $\mathcal{R}$, an Armstrong database for Σ is a collection of relation instances such that:

- Every dependency $\varphi \in \Sigma^*$ holds in the database.
- Every dependency $\varphi' \notin \Sigma^*$ is violated in the database.

Let Σ be a set of heterogeneous dependencies defined over a set of relation schemas $\{R_1, R_2, \ldots, R_n\}$. In this setting, an Armstrong database is a multi-relation instance that satisfies precisely the dependencies in Σ^* and violates all others of the same kind.

An *Armstrong relation* is a special case of an Armstrong database in which only a *single relation* is considered. It is often referred to as an *Armstrong table*.

Armstrong databases and tables are essential tools in database theory and practice. Their utility stems from the fact that they present minimal but sufficient data examples that are consistent with a known constraint set - facilitating error detection, explanation, and communication between domain experts and data engineers. While constructing Armstrong databases becomes more challenging when dealing with multiple classes of constraints and inter-relational dependencies, they provide a precise and faithful representation of schema-level semantics.

△

Suppose we now have two relation schemas: $R(A, B, C)$ and $S(A, D)$, and a set of constraints

$$\Sigma = \{A \to BC, R[A] \subseteq S[A]\},$$

which includes both functional dependencies (FDs) and an inclusion dependency (IND). In this case, the closure Σ^* is equal to Σ, as no additional dependencies can be inferred.

An *Armstrong database* in this context consists of a pair of relation instances r over R and s over S, such that all dependencies in Σ^* are satisfied, and every dependency of the same type that is not entailed by Σ is violated. This requires careful coordination between the data in r and s: for instance, to satisfy the IND $R[A] \subseteq S[A]$, all values that appear in the A attribute of r must also appear in the A attribute of s. On the other hand, to intentionally violate the IND $S[A] \subseteq R[A]$ (which is not implied by Σ), at least one value must appear in $s[A]$ that does not appear in $r[A]$.

r:

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_3	b_1	c_2
a_4	b_2	c_2

s:

A	D
a_1	d_1
a_2	d_2
a_3	d_3
a_4	d_4
a_5	d_4
a_5	d_5

Table 1.2: Example of Armstrong Database for mixed constrains set

In practice, Armstrong databases are particularly useful early in the design phase, where dependencies are being specified, refined, or discovered. By examining these representative instances, domain experts - who may not be database specialists - can inspect concrete examples of how data behaves under a proposed schema. This aids in the identification of missing or incorrect constraints, fosters better communication between domain experts and database designers, and helps ensure that the resulting schema reflects the true business rules or application logic. As such, Armstrong databases bridge the gap between theoretical constraint reasoning and practical schema development.

To illustrate the concept, consider a simple example. Suppose a database designer creates a relation schema R = [Item(I), Source(S), CostMethod(C)] and specifies that Item (I) is a key. This means the intended set of functional dependencies is $\Sigma = \{I \to SC\}$, indicating that the item uniquely determines both its source and cost method. An Armstrong relation r over R is constructed to satisfy exactly the dependencies in Σ^* - in this case, all dependencies that logically follow from $I \to SC$ - and to violate any FD not implied by this dependency. Table 1.3 provides a concrete example of such an Armstrong relation, serving as a minimal and precise test case for evaluating the designer's assumptions.

Item(I)	Source(S)	CostMethod(C)
Mattress King(i_1)	Manufactured(s_1)	Standard(c_1)
Base Queen(i_2)	Manufactured(s_1)	FIFO(c_2)
Frame Double(i_3)	Transfered(s_2)	Average(c_3)
Foam Block(i_4)	Purchased(s_3)	Average(c_3)

Table 1.3: Example of Armstrong Table

A domain expert who is familiar with company finance policy would immediately recognize a subtle error in the example schema. Specifically, although the database designer initially specified only the functional dependency $\Sigma = \{I \to SC\}$, the domain expert knows that, in practice, items supplied by the same source always use the

same cost method. This rule exists to simplify the work of accountants and reduce ambiguity in financial reporting. Upon inspecting the Armstrong relation constructed from Σ, the domain expert would observe a counterexample violating this implicit policy - namely, that tuples with the same value for S (source) may have differing values for C (cost method). This observation enables the database expert to identify a previously omitted constraint: the functional dependency $S \to C$. The constraint can then be added to the schema, ensuring alignment with domain policies and improving the overall accuracy of the data model.

Importantly, domain experts are not required to possess technical knowledge of database theory or constraint logic. Armstrong databases function as intuitive, representative instances that visualize how a system of constraints behaves. Because these databases are designed to satisfy only the dependencies in a given set (and no others), domain experts can easily detect violations of business rules in concrete data without needing to understand formal definitions of FDs, INDs, or other constraint types. Once these issues are flagged, it is the responsibility of the database expert to interpret and formalize the missing constraints accordingly.

Armstrong databases also play a critical role in the maintenance and refinement of existing databases. In real-world applications, it is common for data to be imported from external sources - such as spreadsheets or legacy systems - where constraint metadata may be missing or incomplete. In such cases, constraints can be discovered through data profiling or mining techniques. However, due to the presence of dirty or noisy data, valid (intended) constraints can be violated, leading to incomplete or misleading results. To address this, a small Armstrong database can be constructed from the mined or hypothesized constraints to act as a clean, minimal model that exactly reflects the intended semantics of those dependencies. By visualizing the Armstrong relation, designers and domain experts can collaboratively assess whether the hypothesized constraints capture the true structure of the data, thereby helping to identify missing but valid constraints that might be obscured by data quality issues. Rather than flagging "errors" in the data, these visualizations assist in recovering the intensional structure of the schema and improving the accuracy, interpretability, and usability of the derived constraints.

1.3 Additional Concepts

Definition 1.3.1 (Standard and Non-Standard Keys and Functional Dependencies)**.** In relational database theory, a *key* is considered *standard* if it is defined over a non-empty set of attributes. That is, a standard key consists of one or more attributes whose values uniquely identify each tuple in a relation. In contrast, an *empty key* - i.e., the key over the empty set of attributes - is regarded as *non-standard*. The empty key can only be satisfied if the relation contains *at most one tuple*, since no attribute values are available to distinguish tuples.

Similarly, a *standard functional dependency (FD)* requires that the left-hand side (LHS) of the dependency be non-empty. That is, expressions of the form $\emptyset \to Y$ are not permitted under standard FD semantics. However, under a broader interpretation - often referred to as *non-standard* or *unrestricted* FDs - such dependencies are allowed and carry a specific meaning. In this case, a non-standard FD of the form $\emptyset \to Y$ implies that all attributes in Y must have a single constant value throughout the relation. Formally, for a relation r to satisfy $\emptyset \to Y$, it must hold that for all $A \in Y$ and for all tuples $t_1, t_2 \in r$, we have $t_1[A] = t_2[A]$[7].

Such constraints are sometimes useful in data analysis or integrity enforcement, especially in contexts where global constants (e.g., fixed currency codes, version tags, or policy flags) are expected to remain invariant across all rows of a dataset.

△

Definition 1.3.2 (Acyclic Inclusion Dependencies)**.** A set of inclusion dependencies (INDs) Σ over a relational schema $\mathcal{R}$ is said to be *acyclic* if there exists no sequence of INDs of the form

$$R_1[X_1] \subseteq R_2[Y_1],\ R_2[X_2] \subseteq R_3[Y_2],\ \ldots,\ R_n[X_n] \subseteq R_1[Y_n]$$

such that each inclusion dependency $R_i[X_i] \subseteq R_{i+1}[Y_i]$ connects relation R_i to R_{i+1} for $i = 1, \ldots, n-1$, and the final dependency $R_n[X_n] \subseteq R_1[Y_n]$ completes the cycle by returning to the starting relation R_1[6].

An acyclic IND graph ensures that no attribute dependency paths loop back to the original relation. This is desirable because cyclic INDs introduce complexity in schema design, integrity checking, and query evaluation.

Example (Cyclic INDs): Consider a schema $\mathcal{R} = \{R_1, R_2, R_3\}$ and a set of INDs:

$$\Sigma = \{R_1[D] \subseteq R_2[D],\ R_2[M] \subseteq R_3[M],\ R_3[E] \subseteq R_1[E]\}$$

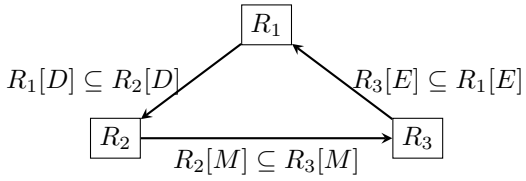
with the following relation descriptions:

– $R_1(E, D)$: `Employee (EmpID, DeptID)`

– $R_2(D, M)$: `Department (DeptID, ManagerID)`

– $R_3(M, E)$: `Manager (ManagerID, EmpID)`

This configuration forms a cycle in the IND graph, as shown below:



This cycle $R_1 \to R_2 \to R_3 \to R_1$ violates the acyclicity condition. Such cyclic dependencies can cause complications, especially when determining the correct order of tuple insertions, deletions, or updates in a database. In many database management systems (DBMSs), referential integrity constraints are enforced incrementally - after each modification rather than at the end of a transaction. In this setting, cycles can lead to violations that are difficult to resolve without deferring constraint checks or reordering operations.

Acyclic INDs, by contrast, simplify these challenges. They allow for more straightforward integrity checking, and enable efficient query planning. Acyclicity ensures that the inclusion dependency graph forms a directed acyclic graph (DAG), which makes reasoning about data flow, referential paths, and join strategies more tractable. Consequently, many schema design algorithms and normalization procedures either assume or enforce acyclicity of INDs to guarantee consistency and correctness[8].

△

Definition 1.3.3 (Typed Inclusion Dependencies)**.** An inclusion dependency (IND) of the form $R[X] \subseteq S[Y]$ is called *typed* if $|X| = |Y|$ and, for each $i \in [1, |X|]$, the i-th attribute in X has the same name and domain as the i-th attribute in Y. That is, the attributes in X and Y can be matched positionally, and must be identical in name.

Typed INDs permit attribute renaming, but not arbitrary renaming of individual attributes since attribute names within a relation must remain distinct.

For example, consider two relations `R(A, B, C)` and `S(D, E, F)`. Let

$$\Sigma = \{R[AB] \subseteq S[DE],\ R[BC] \subseteq S[EF]\}.$$

This set Σ is a set of *typed* INDs, since both INDs satisfy the typedness condition: in each case after renaming (e.g., the relation `S` is renamed as `S(A, B, C)`), the left- and right-hand sides consist of two attributes of matching positions and names. Most INDs that arise in practice are typed. For instance, all INDs in the acyclicity example earlier were typed.

In contrast, an *untyped* IND does not satisfy the typedness condition. Consider the relation `Employee(EmpName, Manager)`. The IND

`Employee[Manager]` $\subseteq$ `Employee[EmpName]`

is *untyped*, since the attribute names on the left and right differ. Although this IND may still carry semantic meaning (e.g., asserting that every manager is also an employee), it violates the typedness condition: we cannot rename either attribute to match the other, as this would result in duplicate attribute names within the same schema.

Typed INDs play a crucial role in theoretical and practical aspects of database dependency reasoning. Notably, they prevent arbitrary permutations of attributes between relations, which can lead to undecidability in implication problems and increase computational complexity[9].

△

1.4 Related work

The investigation of Armstrong databases has attracted significant scholarly interest, primarily due to their combinatorial complexity and the noteworthy characteristic that they are not guaranteed to exist. This inherent intricacy, coupled with their conditional existence, positions Armstrong tables as a compelling object of study.

Early research on Armstrong databases, particularly the seminal work by Ronald Fagin and Moshe Y. Vardi [7], laid the groundwork for understanding the interplay between different classes of dependencies. Their investigation focused on the existence of Armstrong databases for constraint sets that combine inclusion dependencies (INDs) and functional dependencies (FDs). Fagin and Vardi demonstrated that the existence of such databases depends on certain restrictions, such as limiting consideration to standard FDs. Their finding underscored the complexity and delicate balance required when combining heterogeneous types of dependencies and spurred further research into identifying conditions under which Armstrong databases can be guaranteed.

Subsequent studies revealed that determining implications for FDs and INDs is generally undecidable. However, when the problem is restricted to cases involving acyclic INDs and FDs, or typed INDs and acyclic FDs, the implication problem becomes both decidable and tractable. A key contribution in this direction was made by Stavros S. Cosmadakis and Paris C. Kanellakis, where they employed a graph-theoretic framework to analyze the interaction between FDs and INDs. Their approach provided valuable insight into the structural properties that govern the decidability of dependency implication. By representing dependencies as graphs, they were able to characterize cycles and establish criteria under which inference remains computationally manageable. In particular, they showed that if the set of INDs forms an acyclic graph and interacts with a restricted form of FDs, the implication problem avoids the general undecidability result and instead admits efficient algorithms. Their work laid the foundation for later studies that further explored tractable fragments of dependency implication and contributed significantly to our theoretical understanding of data dependency inference in relational databases[10].

Further research has explored the construction of Armstrong databases with increasing focus on both the theoretical foundations and practical algorithms for generating instances that precisely capture specific dependency sets. For single classes of constraints, such as keys, work by Van Bao Tran Le, Sebastian Link, and Mozhgan Memari introduced methods for discovering and validating keys directly from SQL tables, demonstrating that Armstrong databases can support the reverse engineering of database constraints from real-world data[11]. Similarly, for functional dependencies (FDs), foundational work by Beeri, Dowd, Fagin, and Statman analyzed the structural properties of Armstrong relations, providing deep insights into how FDs can be represented using minimal instances. These studies confirmed that, for individual dependency types, Armstrong databases can be constructed algorithmically, and their structure can be understood and controlled[12]. Extending beyond single constraint classes, Ronald Fagin's earlier work on the use of Horn clauses to represent database dependencies offered a unifying formalism for reasoning about multiple types of constraints simultaneously, including FDs and more expressive forms. His results demonstrated how logical inference systems could be employed to determine the implications of complex dependency sets and, under certain restrictions, support the construction of Armstrong-like databases that accurately reflect mixed constraint environments. These efforts collectively laid the groundwork for modern techniques in schema design, dependency inference, and data quality assurance by showing both the potential and the limits of using Armstrong databases as precise tools for dependency representation[13].

Another line of research addresses scenarios involving NULL values, which are common in real-world databases but introduce significant complexity into dependency reasoning. In particular, the presence of NULLs undermines the classical interpretation of database constraints, as standard semantics for keys and functional dependencies (FDs) no longer apply straightforwardly. To handle this, researchers have introduced the notions of possible and certain semantics, which provide a framework for interpreting constraints under uncertainty. A possible world represents one of the many complete instances that could result from treating NULLs as placeholders for actual values, while a certain semantic captures what holds true across all such interpretations. This paradigm shift complicates the existence and construction of Armstrong tables, since it is no longer sufficient to build a single instance that satisfies exactly the given dependencies - one must also account for the range of interpretations that arise from NULL values. The work by Koehler, Leck, Link, and Zhou exemplifies this by developing formal definitions for possible and certain keys in SQL databases, and by providing algorithms for identifying them in the presence of NULLs. Their findings reveal that constructing Armstrong databases in such settings requires additional logical

and computational steps, including reasoning over multiple potential interpretations of the data and ensuring that constraints are respected in all - or at least in some - relevant worlds. As a result, this line of inquiry not only extends the theoretical boundaries of Armstrong databases but also enhances their applicability in practical environments where incomplete or uncertain information is the norm[14].

Recent studies have also focused on data profiling, which aims to extract structural information and hidden constraints from datasets. Wei and Link introduced Dataprof, a semantic profiling framework designed to support iterative data cleansing and business rule acquisition [15]. Their approach emphasizes both structural constraint detection and semantic interpretation to improve data quality and schema understanding. While Armstrong database construction follows a top-down approach from a dependency set, data profiling operates bottom-up by inferring dependencies from existing data. Together, these approaches contribute to schema discovery, data integration, and data quality assurance.

Keys remain among the most fundamental constraints in relational databases, yet key definitions are often missing or incomplete in practice. To address this problem, BirnickF, Bläsius, and colleagues proposed Unique Column Combinations (UCCs) as candidate keys and introduced HPIValid, a hitting set-based UCC discovery algorithm [16]. By integrating partial information, dynamic validation, and minimal hitting set enumeration, HPIValid substantially improves runtime and memory efficiency when discovering minimal UCCs in large datasets. Since Armstrong table construction relies on agree sets derived from keys and functional dependencies, efficient UCC discovery techniques provide a strong foundation for identifying candidate constraints.

Functional dependency discovery has also been extensively studied due to its importance in schema normalization, data cleaning, and query optimization. Papenbrock and Naumann proposed HyFD, a hybrid FD discovery algorithm that combines row-efficient sampling with column-efficient validation [17]. This approach enables scalable discovery of minimal and non-trivial FDs in large real-world datasets while reducing memory consumption through dynamic pruning techniques.

More recent work has further improved the scalability of FD discovery algorithms. Bleifuss et al. proposed a method based on hitting set enumeration that achieves notable runtime improvements and substantial memory reduction compared to previous approaches [18]. Their work addresses major scalability bottlenecks in dependency discovery and enables more efficient profiling of large datasets. Once candidate dependencies are discovered, Armstrong tables can support human-in-the-loop validation by providing concrete examples that satisfy or violate specific constraints. This connection strengthens the role of Armstrong databases in modern data profiling and schema design workflows.

Existing research on integrity constraint discovery has largely focused on identifying exact and approximate key constraints from relational data, particularly when database schemas are incomplete or outdated. Early methods relied on exhaustive search and pruning strategies to discover minimal keys, but often produced many accidental keys in high-dimensional datasets. More recent research has introduced arity restrictions and statistical approaches that analyze attribute correlations to distinguish meaningful keys from coincidental uniqueness. However, in the process of filtering accidental constraints, some genuinely meaningful constraints may inevitably be misclassified and discarded. Additional studies addressed noisy and incomplete data through approximate, weak, and partial key semantics, while measures such as dirtiness and violation rates were proposed to evaluate how closely candidate keys satisfy dependency properties [19]. In this context, Armstrong tables can serve as a complementary mechanism to recover meaningful constraints that may have been overlooked during the discovery process.

An early recognized drawback of Armstrong tables is their potential to grow exponentially in size, both with respect to the number of attributes and the number of dependencies, making them impractical for use in large or complex schemas. Even modest sets of functional dependencies could lead to Armstrong relations of prohibitive size[2]. Such exponential growth not only limits the scalability of Armstrong tables in real-world applications but also complicates their role in tasks such as dependency verification. The root of this challenge lies in the requirement that Armstrong tables capture all and only the consequences of a given dependency set - ensuring completeness and minimality - which often necessitates a large number of carefully constructed tuples. To address this issue and enhance the practical utility of Armstrong tables, our research focuses on strategies aiming at producing smaller, more manageable Armstrong relations without sacrificing their representational fidelity.

1.5 Broader Context within Database and Data Management Research

Constraint management remains a foundational topic in relational database research. Integrity constraints such as keys, functional dependencies, and inclusion dependencies are central to schema design, normalization, data integration, data cleaning, and data quality assurance. These constraints provide the semantic foundations that allow database systems to reason about data correctness, redundancy, and consistency[1, 6]. Within this broader context, Armstrong tables provide a concrete mechanism for interpreting and validating the semantics represented by database constraints.

In practice, constraints arise through multiple processes. During schema design, dependencies are manually specified by database designers based on application semantics and domain knowledge. In modern data-driven environments, however, constraints are increasingly inferred automatically through dependency discovery, data profiling, and metadata extraction techniques[20]. Although these methods can identify useful constraints, the discovered dependencies are often noisy, incomplete, or semantically ambiguous. Armstrong tables complement such approaches by providing compact example instances that enable users to inspect, validate, and communicate discovered constraints more effectively.They are particularly valuable when distinguishing meaningful constraints from accidental patterns that arise from limited or noisy datasets.

Constraints also play a central role in schema normalization and decomposition theory. Functional dependencies form the theoretical foundation of normalization algorithms, including decomposition into Boyce–Codd Normal Form (BCNF) and Third Normal Form (3NF) [21, 22, 23]. Since the correctness and effectiveness of these decompositions depend heavily on accurately specified dependencies, Armstrong tables provide a practical mechanism for validating intended semantics and discovering missing constraints during schema refinement.

Beyond schema design, constraints are essential for maintaining data quality and supporting data cleaning processes. Modern data quality frameworks use constraints to detect anomalies, inconsistencies, duplicate records, and incomplete information [24, 25]. This is particularly important in large-scale enterprise systems, heterogeneous integration environments, and data lakes, where data errors can propagate across multiple downstream applications. Because Armstrong tables provide concrete examples that satisfy exactly a given set of constraints, they offer an interpretable and example-driven approach for validating and communicating data quality rules before deployment.

More broadly, this research aligns with ongoing developments in explainable and human-centered data management. As database systems become increasingly automated and data ecosystems continue to grow in complexity, there is an increasing need for interpretable representations of metadata, constraints, and semantic assumptions. Compact Armstrong tables contribute to this goal by transforming abstract logical specifications into concrete and understandable examples. Consequently, the techniques developed in this thesis contribute not only to dependency theory, but also to broader areas including schema engineering, data governance, data profiling, data cleaning, and trustworthy data management systems.

1.6 Research Aims and Objectives

The primary aim of this thesis is to investigate practical approaches for constructing compact and semantically meaningful Armstrong tables for relational database constraints. Although Armstrong databases are well established in database theory, their practical adoption remains limited due to the excessive size. This research therefore focuses on improving both scalability and interpretability in Armstrong table construction.

Scalability is a critical concern because large Armstrong tables become difficult for database designers and domain experts to analyse, often obscuring missing constraints within large volumes of data. At the same time, interpretability is equally important. Compact and semantically informative Armstrong tables can enhance schema validation, dependency analysis, and communication between technical and non-technical stakeholders.

The specific objectives of this research are as follows:

1. To study the theoretical foundations of Armstrong databases and their role in representing relational constraints such as keys and functional dependencies.

2. To investigate existing Armstrong table construction techniques and identify limitations related to scalability and practical usability.
3. To evaluate optimization strategies for combining multiple classes of constraints while minimizing the resulting table size.
4. To develop efficient methods for constructing smaller Armstrong tables, based solely on a given set of constraints.
5. To build smaller informative Armstrong tables that improve semantic understanding for database designers and domain experts.
6. To extend Armstrong table construction techniques to databases containing incomplete information, particularly schemas involving `NULL` values.

Chapter 2

Foundations

The existence of Armstrong databases is not guaranteed in general. In this chapter, we focus on the construction of Armstrong databases in cases where their existence is ensured. That is, we only consider scenarios where it is known that Armstrong databases do exist.

2.1 Existence of Armstrong Databases

For constraint sets that contain only a single class of dependencies - such as keys, Functional Dependencies (FDs), or Inclusion Dependencies (INDs) - Armstrong databases are guaranteed to exist. However, when a constraint set contains a "mixture" of different types of dependencies, the existence of an Armstrong database is not always assured.

Notably, in cases involving combinations of FDs and INDs, the existence of Armstrong databases depends on certain restrictions. Fagin and Vardi [7] have established results under which Armstrong databases do or do not exist:

- If the constraint set includes only standard FDs and typed INDs, then an Armstrong database always exists.
- If the constraint set includes unrestricted FDs and INDs, then an Armstrong database may not exist. The non-existence can be demonstrated through the following example:
  - Let the database schema be $R = \{R_1, R_2, R_3, R_4\}$, where each relation schema contains exactly one attribute: $R_1(A_1), R_2(A_2), R_3(A_3), R_4(A_4)$
  - Let the constraint set be: $\Sigma = \{\emptyset \to R_1[A_1],\ R_2[A_2] \subseteq R_1[A_1],\ R_2[A_2] \subseteq R_3[A_3]\}$
  - Let $\sigma_1 = \{R_1[A_1] \subseteq R_3[A_3]\}$ and $\sigma_2 = \{R_2[A_2] \subseteq R_4[A_4]\}$. Both of these INDs are not implied by Σ.
  - Now consider a database instance $r = \{r_1, r_2, r_3, r_4\}$ that satisfies Σ. We examine two possible cases:
    * Case 1: r_2 is empty.
      In this case, σ_2 is trivially satisfied because an empty set is a subset of any relation.
    * Case 2: r_2 is not empty.
      Then r_1 and r_2 must each contain exactly one tuple, and they must be equal: $r_1 = r_2$.
      Since $r_2 \subseteq r_3$ by the constraint in Σ, it follows that $r_1 \subseteq r_3$, which means that σ_1 is satisfied.

  In both cases, at least one of σ_1 or σ_2 is satisfied, despite neither being implied by Σ. This contradicts the basic requirement of an Armstrong database: that only the dependencies in the closure of Σ are satisfied. Therefore, an Armstrong database cannot exist in this case.

2.2 Construction of Armstrong database

Constructing Armstrong tables is, in general, an undecidable problem. That is, given certain combinations of constraints and dependencies, it may be fundamentally impossible to determine whether a valid Armstrong table exists. Even in cases where an Armstrong table can, in principle, be constructed, the process is often computationally intractable. The time and space complexity can grow exponentially with the number of attributes and constraints involved, rendering the task impractical for large schemas.

The construction of an Armstrong table requires carefully synthesizing data that satisfies all semantically valid constraints - those that are logically implied by a given set - while simultaneously violating those that are not. This dual requirement ensures the resulting table precisely characterizes the logical closure of the initial constraint set, thereby acting as a definitive model of the dependency theory. Such synthesis demands sophisticated reasoning about dependencies and non-dependencies, making the process not only resource-intensive but also algorithmically challenging.

Before delving into the construction methodology, it is essential to clarify several foundational definitions and concepts that underpin Armstrong tables and their role in formal data modeling.

Definition 2.2.1 (Agree Sets)**.** Let r be a relation instance over a relation schema R. Given two tuples $u, v \in r$, the *agree set* of u and v, denoted $ag(u, v)$, is the set of attributes in R on which the two tuples agree, i.e.,

$$ag(u, v) = \{A \in R \mid u[A] = v[A]\}.$$

The *agree set of the relation* r, denoted $ag(r)$, is the collection of all agree sets of pairs of tuples from r r:

$$ag(r) = \{ag(u, v) \mid u, v \in r\}[5].$$

△

Agree sets are a useful tool in reasoning about dependencies in relational databases. Sepcially in the construction of *Armstrong databases* - databases that satisfy exactly a given set of keyss and functional dependencies (FDs) and no others. The strategic use of agree sets allows precise control over which dependencies are satisfied or violated in the constructed instance.

Definition 2.2.2 (Chase)**.** The *chase* is a fixed-point algorithm used to test whether a database instance satisfies a set of data dependencies and, when necessary, to enforce those dependencies[5]. The chase operates by repeatedly applying rules that enforce functional dependencies (FDs), inclusion dependencies (INDs), or combinations of both, until no further violations exist.

- **Chase for Functional Dependencies (FDs)**
  Let Σ be a set of FDs over schema R, and suppose Σ includes a dependency $X \to Y$. Let r be a relation over R. If there exist tuples $t, t' \in r$ such that $t[X] = t'[X]$, but $t[A] \neq t'[A]$ for some attribute $A \in Y$, then modify the relation by setting both $t[A]$ and $t'[A]$ to a common value. One typical strategy is to replace the larger value with the smaller one[5].

- **Chase for Inclusion Dependencies (INDs) - Specialized Chase**
  Let $R = (R_1, \ldots, R_k)$ be a database schema and let r be a database instance over R. Suppose Σ contains an IND of the form $R_i[X] \subseteq R_j[Y]$. If a tuple $t \in r_i$ violates this IND (i.e., $t[X] \notin r_j[Y]$), then insert a new tuple $s \in r_j$ such that $s[Y] = t[X]$, and assign fixed new values to all other attributes of s. This approach guarantees termination even in the presence of cyclic INDs.

- **Chase for FDs and INDs - General Chase**
  The specialized chase described above is insufficient when both FDs and INDs are present, as using fixed default values to repair INDs can lead to violations of FDs. In such cases, the *general chase* is applied. FD violations are resolved similarly as in the FD-only case, but IND violations are repaired by introducing new fresh values from the domain for attributes not involved in the inclusion. This avoids conflicts with existing FD constraints and enables consistent repairs. △

When generating Armstrong databases for INDs, a different approach is typically required. Initially, the database is constructed using *tableaux* that intentionally violate all INDs in the set Σ. The chase procedure is then applied iteratively to repair the violations while ensuring that only the intended INDs are eventually satisfied. This staged repair process ensures that the resulting instance serves as an Armstrong database, satisfying exactly the INDs implied by Σ and no others.

2.2.1 Armstrong databases for keys

To construct Armstrong databases for keys, we need to distinguish between two important concepts: the sets of attributes that function as keys, and those that do not. The latter are called *anti-keys*. Furthermore, the notion of a *transversal* helps us systematically generate anti-keys from the given set of keys. We begin with formal definitions of these terms.

Definition 2.2.3 (Anti-Key)**.** An *Anti-Key* is a set of attributes in a relation schema that fails to serve as a uniqueness constraint - that is, it does **not** functionally determine all other attributes within the relation. △

Definition 2.2.4 (Transversal)**.** In mathematics, a *transversal* for a family of sets $\mathcal{F} = \{S_1, S_2, \ldots, S_n\}$ is a set that contains exactly one element from each set S_i.

For example, given $\mathcal{F} = \{\{A\}, \{B, C\}\}$, the transversals of $\mathcal{F}$ are $\{A, B\}$ and $\{A, C\}$; that is,

$$\mathcal{T}_r\big(\{\{A\}, \{B, C\}\}\big) = \{\{A, B\}, \{A, C\}\}.$$

Alternatively, using a compact notation, we can write $\mathcal{T}_r\big(\{A, BC\}\big) = \{AB, AC\}$. △

An Armstrong table for a given set of keys will satisfy precisely the logical consequences of those keys and will violate all other keys not implied by them, known as anti-keys. For a set of keys Σ, the anti-keys are the maximal set of keys that are not implied by Σ.

To start constructing an Armstrong table over schema R from a given set of keys Σ, we follow the method outlined by Mannila and Raiha[2]:

1. Compute the set of anti-keys
   - Construct the *transversal* of the set of keys.
   - For each element in the transversal, compute its *complement* with respect to the full attribute set R; the resulting set of complements forms the anti-keys.
2. Generate rows (tuples) for the Armstrong table
   - For each anti-key, construct a pair of tuples that agree on the anti-key but differ elsewhere. This demonstrates that the anti-key does not functionally determine the rest of the attributes.

This process yields an Armstrong table that captures precisely the implications of the key set Σ. Additional implementation details will be discussed in Chapter 4.

2.2.2 Armstrong Databases for Functional Dependencies

Before describing the construction of Armstrong databases for Functional Dependencies(FDs), we introduce several foundational concepts, including Closure Operator, Closed Set, Anti-Functional Dependency (Anti-FD), Maximal Set, and Generator Set.

Definition 2.2.5 (Closure Operator)**.** A *closure operator* on a set S is a function $\mathrm{cl} : \mathcal{P}(S) \to \mathcal{P}(S)$, mapping the power set of S to itself, that satisfies the following three properties:

- *Extensivity:* For all $A \subseteq S$, $A \subseteq \mathrm{cl}(A)$.
- *Monotonicity:* If $A \subseteq B \subseteq S$, then $\mathrm{cl}(A) \subseteq \mathrm{cl}(B)$.
- *Idempotency:* For all $A \subseteq S$, $\mathrm{cl}(\mathrm{cl}(A)) = \mathrm{cl}(A)$.

An important instance of a closure operator arises in the context of relational databases. Let R be a relation schema, and let Σ be a set of functional dependencies (FDs) over R. For any set of attributes $X \subseteq R$, the *attribute closure* of X under Σ, denoted $\mathrm{cl}_\Sigma(X)$ or X^*, is defined as:

$$\mathrm{cl}_\Sigma(X) = \{A \in R \mid \Sigma \vDash X \to A\},$$

where $\Sigma \vDash X \to A$ means that $X \to A$ is logically entailed by Σ, i.e., it holds in every relation that satisfies all dependencies in Σ[5].

This attribute closure captures all attributes in R that are functionally determined by X according to Σ, and the function cl_Σ satisfies the three properties above.

△

Definition 2.2.6 (Closed Set)**.** Given a relation schema R and a set of functional dependencies Σ, a subset $V \subseteq R$ is said to be *closed* under the closure operator cl_Σ if

$$\mathrm{cl}_\Sigma(V) = V,$$

that is, V contains all attributes that are logically implied by Σ given V as the set of known attributes.

Equivalently, V is closed if for every attribute $A \in R$, we have

$$V \to A \quad \text{if and only if} \quad A \in V.$$

The set of all such closed subsets of R (with respect to Σ) is denoted by $\mathrm{CL}_\Sigma(R)$.

△

Definition 2.2.7 (Anti-Functional Dependency (Anti-FD))**.** Let R be a relation schema, and let $X, Y \subseteq R$ be sets of attributes. An *anti-functional dependency*, denoted $X \nrightarrow Y$, holds in a relation instance over R if the attributes in X do **not** functionally determine the attributes in Y. That is, there exists at least one pair of tuples t_1, t_2 in some relation instance r over R such that:

$$t_1[X] = t_2[X] \quad \text{but} \quad t_1[Y] \neq t_2[Y].$$

In other words, knowing the values of X is not sufficient to uniquely determine the values of Y. The anti-functional dependency $X \nrightarrow Y$ is the logical negation of the functional dependency $X \to Y$.

△

Definition 2.2.8 (Maximal Set)**.** Let Σ be a set of functional dependencies (FDs) over a relation schema R, X a subset of R and $A \in X$. A set $Y \subseteq X$ is called a *maximal set for* A if the following two conditions hold:

1. $\Sigma \not\models Y \to A$ (i.e., Y does not functionally determine A; this is an anti-FD),
2. For every strict superset $Z \supset Y$ (i.e., $Z \subseteq X$ and $Y \subset Z$), we have $\Sigma \models Z \to A$.

In other words, Y is a maximal subset of attributes that fails to determine A, such that adding any attribute from $R \setminus Y$ to Y results in a set that *does* determine A. Thus, Y is the left-hand side of an anti-functional dependency (anti-FD) for A, and it is *maximal* with respect to this property[5].

We denote the collection of all such maximal sets for a given attribute A by:

$$MAX\_lhs_\Sigma(A) = \{Y \subseteq R \mid Y \neq \emptyset,\ Y \nrightarrow A, \text{ and } \forall Z \supset Y,\ \Sigma \models Z \to A\}.$$

For brevity, when the context is clear, we may write this as $MAX_\Sigma(A)$.

The set of all maximal sets across all attributes of the schema R is defined as:

$$MAX_\Sigma(R) = \bigcup_{A \in R} MAX_\Sigma(A).$$

△

Definition 2.2.9 (Generator Sets)**.** Let Σ be a set of functional dependencies (FDs) over a relation schema R, and let $CL_\Sigma(R)$ denote the collection of all closed sets under the closure operator induced by Σ. A subfamily $GEN_\Sigma(R) \subseteq CL_\Sigma(R)$ is called the *generator set* of $CL_\Sigma(R)$ if the following conditions hold:

1. For every $C \in CL_\Sigma(R)$, there exists a subset $\mathcal{G} \subseteq GEN_\Sigma(R)$ such that:
$$C = \bigcap_{G \in \mathcal{G}} G.$$
2. The set $GEN_\Sigma(R)$ is *minimal* with respect to inclusion; that is, no proper subfamily of $GEN_\Sigma(R)$ satisfies the above property.

In other words, every closed set in $CL_\Sigma(R)$ can be represented as an intersection of generator sets, and this collection of generators is the smallest possible family with this property.

A fundamental result by Mannila and Raiha[2] shows that the set of maximal sets with respect to anti-functional dependencies coincides with the generator sets. Formally,

$$MAX_\Sigma(R) = GEN_\Sigma(R).$$

△

Equipped with the above theoretical foundations, we can now begin the process of constructing *Armstrong databases* for a given set of functional dependencies (FDs) Σ.

By definition, an *Armstrong relation* for Σ is a relation instance r over schema R that satisfies exactly those FDs that are logically implied by Σ - that is, it satisfies all FDs in the closure Σ^*, and violates all FDs not in Σ^*.

According to a key result by Mannila and Raiha[2], a relation r is an Armstrong relation for Σ if and only if:

$$GEN_\Sigma(R) \subseteq ag(r) \subseteq CL_\Sigma(R),$$

where:

- $ag(r) = \{X \subseteq R \mid \exists t_1, t_2 \in r,\ t_1 \neq t_2 \wedge t_1[X] = t_2[X]\}$ denote the set of attribute subsets X over the schema R on which at least two distinct tuples in r agree.
- $CL_\Sigma(R)$ is the closure system of attribute sets corresponding to FDs implied by Σ,
- $GEN_\Sigma(R)$ is the minimal generator set (i.e., the base) of this closure system.

The condition $GEN_\Sigma(R) \subseteq ag(r)$ ensures that all *anti-FDs* are violated in the relation r, by enforcing tuples agree on the corresponding attribute sets. In the mean time, the condition $ag(r) \subseteq CL_\Sigma(R)$ guarantees that all FDs logically implied by Σ are satisfied in r.

Based on these principles, the construction of an Armstrong relation proceeds through the following steps:

- Step 1: Compute the generator set $GEN_\Sigma(R)$.
  Since it is known that $GEN_\Sigma(R) = MAX_\Sigma(R)$[2], we can compute $GEN_\Sigma(R)$ by computing $MAX_\Sigma(R)$. This is done incrementally, as follows:
  1. Initialization (Empty FD Set):
     When $\Sigma = \emptyset$, for each attribute $A \in R$, we have:
     $$MAX_\emptyset(A) = \{R \setminus \{A\}\}.$$
  2. Incremental Update with Each FD:
     Let $\Sigma = \Sigma' \cup \{Y \to Y'\}$, and let $X \in MAX_{\Sigma'}(A)$. For each $W \in MAX_\Sigma(A)$:
     – If the left-hand side (LHS) attributes $Y \not\subseteq X$, or the right-hand side (RHS) attributes $Y' \subseteq X$, then $W \in MAX_{\Sigma'}(A)$.

- Otherwise, remove W from $MAX_\Sigma(A)$ and add each set $W \cap Z$, for every $B \in Y$, and $Z \in MAX_{\Sigma'}(B)$.

3. Repeat the process iteratively for all FDs in Σ until the full $MAX_\Sigma(R)$ is computed.

- Step 2: Construct the Armstrong relation.
  Use the computed generator set $GEN_\Sigma(R) = MAX_\Sigma(R)$ to construct a relation r such that:

$$GEN_\Sigma(R) \subseteq ag(r) \subseteq CL_\Sigma(R).$$

  This ensures that r satisfies all FDs in Σ^* and violates all those not logically implied by Σ, making it an Armstrong relation[12].

2.2.3 Armstrong Databases for Inclusion Dependencies (INDs)

When the set of constraints Σ consists solely of inclusion dependencies (INDs), we can use a variant of the Chase algorithm - specifically adapted for INDs - to construct an Armstrong database. Such a database satisfies all INDs that are explicitly present in Σ or logically implied by them, while intentionally violating any IND not implied by Σ.

The process for constructing an Armstrong database under INDs proceeds as follows:

1. Initialization: For each relation (table) in the schema, initialize it with at least one tuple to ensure it is not empty. Ensure that all initial attribute values across all tables are distinct to prevent accidental satisfaction of INDs that are not entailed by Σ.
2. Chase Step for Inclusion Dependencies: For each inclusion dependency in Σ, verify whether it is satisfied in the current instance. If a dependency is violated, add one or more tuples to the appropriate relation(s) to enforce it. When creating such tuples:
   - Assign matching values to the attributes on the right-hand side (RHS) of the IND using values from the corresponding attributes on the left-hand side (LHS).
   - For the remaining attributes in the new tuple, assign fresh values that do not appear elsewhere in the database. This ensures that these tuples do not unintentionally satisfy other inclusion dependencies not implied by Σ.
3. Repeat Until Fixed Point: Continue applying the Chase steps iteratively until no further tuples need to be added. At this point, the instance will satisfy all INDs in Σ^*, while violating all others.

This construction guarantees an instance that behaves as an Armstrong database for the given set of inclusion dependencies.

2.2.4 Armstrong Databases for Multiple Classes of Dependencies

When the constraint set Σ contains multiple classes of dependencies - such as a combination of functional dependencies (FDs), inclusion dependencies (INDs), multivalued dependencies (MVDs), or join dependencies (JDs) - the existence of an Armstrong database is not guaranteed. Even when Armstrong databases do exist, such as for sets involving standard FDs and typed INDs, there is no straightforward or practical method for constructing them.

Nonetheless, Fagin introduced a foundational result in the early 1980s[13], which provides an indirect method to construct Armstrong databases for such mixed sets of dependencies. This method relies on two key concepts: the Direct Product of relations and the Faithfulness of dependencies.

Definition 2.2.10 (Direct Product)**.** A *direct product* of objects is often constructed by defining a new object whose underlying set is the Cartesian product of the component sets, with a structure induced component-wise from the original objects.

For example, consider the *direct product of groups*. Let G and H be groups. Their direct product, denoted $G \times H$, is the set of all ordered pairs (g, h), where $g \in G$ and $h \in H$, equipped with the group operation defined component-wise:

$$(g_1, h_1) \cdot (g_2, h_2) = (g_1 \cdot_G g_2,\ h_1 \cdot_H h_2)$$

for all $g_1, g_2 \in G$ and $h_1, h_2 \in H$. Here, $\cdot_G$ and $\cdot_H$ denote the group operations in G and H, respectively.

In this context, the *direct product* refers to an operation on relations, defined in terms of tuple-wise combination. Let R be a relation schema consisting of attributes $A_1, \ldots, A_n$. Suppose that for each $i \in [1, k]$, the relation r_i is over the schema R, meaning all r_i share the same set of attributes.

We begin by defining the *direct product of tuples*. Given tuples $t_1, \ldots, t_k$, where $t_i \in r_i$, their direct product is a tuple t over the schema R, defined as follows: for each attribute $A_j \in R$,

$$t[A_j] = (t_1[A_j], \ldots, t_k[A_j]).$$

That is, for each attribute, the value in the resulting tuple is a k-tuple consisting of the corresponding attribute values across the input tuples.

Based on this, the *direct product of relations* $r_1, \ldots, r_k$ is the set of all such tuple-wise products:

$$\bigotimes\{r_i \mid i \in [1, k]\} = \{t \mid t = t_1 \otimes \cdots \otimes t_k,\ t_i \in r_i \text{ for all } i \in [1, k]\}.$$

The resulting relation has the same schema R as each r_i, but for each attribute A_j, its domain in the result is the Cartesian product of the respective attribute domains across all r_i:

$$\mathrm{Dom}(A_j) = \mathrm{Dom}_{r_1}(A_j) \times \cdots \times \mathrm{Dom}_{r_k}(A_j).$$

Example:

r_1:

A	B	C
a_1	b_1	c_1
a_1	b_1	c'_1

r_2:

A	B	C
a_2	b_2	c_2
a_2	b'_2	c_2

$r_1 \otimes r_2$:

A	B	C
$\langle a_1, a_2\rangle$	$\langle b_1, b_2\rangle$	$\langle c_1, c_2\rangle$
$\langle a_1, a_2\rangle$	$\langle b_1, b'_2\rangle$	$\langle c_1, c_2\rangle$
$\langle a_1, a_2\rangle$	$\langle b_1, b_2\rangle$	$\langle c'_1, c_2\rangle$
$\langle a_1, a_2\rangle$	$\langle b_1, b'_2\rangle$	$\langle c'_1, c_2\rangle$

Table 2.1: Example of a Direct Product of Relations

△

Definition 2.2.11 (Faithfulness with Respect to Direct Product)**.** A dependency is said to be *faithful* with respect to the Direct Product if it holds for each member of a non-empty family of non-empty relations if and only if it holds for their Direct Product [13].

△

Based on this definition, if a dependency σ is faithful and holds for every relation r_i in a family $r = (r_1, \ldots, r_k)$, then σ also holds for the Direct Product $\bigotimes\{r_i\}$. Fagin proved that many common dependency types are faithful with respect to the Direct Product, including:

- Functional Dependencies (FDs), including keys
- Join Dependencies (JDs), including MVDs
- Inclusion Dependencies (INDs)

This property of faithfulness can be exploited to construct Armstrong databases for sets Σ that contain multiple classes of such dependencies.

Let Θ denote the set of all dependencies that are not logically implied by Σ, i.e.,

$$\Theta = \{\pi_j \mid \Sigma \not\models \pi_j\}$$

To construct an Armstrong database for Σ, the following strategy can be used:

1. For each dependency $\pi_j \in \Theta$, construct a *partial Armstrong database* - a relation that satisfies all constraints in Σ but violates π_j.

2. Compute the Direct Product of all these partial Armstrong databases.

The resulting database will:

- Satisfy every dependency in Σ (and all its logical consequences, Σ^*),
- Violate every dependency in Θ (i.e., all those not implied by Σ),

thus behaving as a valid Armstrong database for Σ.

Limitations. Despite its theoretical elegance, this approach has practical drawbacks:

- Constructing the partial Armstrong databases for each π_j may still be challenging or infeasible in practice.
- The size of the final Armstrong database, resulting from the Direct Product, may grow exponentially with the number of partial databases, making it too large for practical use.

Therefore, while this method provides a foundational theoretical solution for mixed dependencies, its application is often constrained by computational and scalability issues.

2.3 NULL Values

`NULL` is a special marker used in Structured Query Language (SQL) to represent missing, unknown, or inapplicable data values. It was introduced by Edgar F. Codd, the creator of the relational database model, to ensure that relational databases could faithfully represent incomplete information. According to Codd's vision, any true relational database management system (RDBMS) must support the concept of "missing information and inapplicable information."

`NULL` is not a value in itself but rather a placeholder indicating the absence of a value. Its semantics have been widely debated and criticized. As pointed out by computer science professor Ron van der Meyden, the handling of `NULL` in the SQL standard is logically inconsistent to the extent that no intuitive semantics can be universally ascribed to it [26]. Although numerous proposals have been made to resolve these inconsistencies, most alternatives are too complex for practical adoption in commercial systems.

In the era of rapidly expanding data - commonly referred to as *Big Data* - databases must handle enormous volumes of structured, semi-structured, and unstructured information. Traditional RDBMS often struggle with such scale and diversity, particularly in the presence of uncertainty and incomplete information. While *NULL* values offer a practical means of accommodating uncertain or absent data, they also present challenges to relational normalization, which assumes idealized conditions where *NULL*s and duplicates are not present. Consequently, recent research efforts have focused on extending the foundational theories of relational databases - originally developed over four decades ago - to better address the requirements of modern data-intensive applications[23].

2.3.1 NULL Values and Keys

In SQL database systems, *primary keys* are required to be both `UNIQUE` and `NOT NULL`. This ensures that each row in a table can be uniquely identified without ambiguity. On the other hand, the `UNIQUE` constraint alone permits `NULL` values. Since `NULL` represents the absence of a known value, it is not considered equal to any other value, including another `NULL`. Consequently, multiple `NULL`s are not inherently treated as duplicates under the SQL standard. However, certain database systems implement stricter interpretations and allow only a single `NULL` value within a column that is constrained by `UNIQUE`.

To address the complexities introduced by `NULL` in enforcing key constraints, Thalheim proposed the concept of a *key set*[27]. A relation is said to satisfy a key set if, for every pair of distinct tuples, there exists at least one key in the key set such that the tuples are both defined (i.e., non-`NULL`) and distinct on that key. This generalization aims to better accommodate incomplete information in relational schemas.

Building on this, further research has introduced a *possible world semantics* to rigorously define key constraints in the presence of `NULL` values[14]. Under this framework, each *possible world* is derived from a relation by independently substituting each `NULL` with a value from its domain. Based on this interpretation, two types of keys are defined:

- A *possible key* $p\langle X\rangle$ is satisfied by a relation r if there exists at least one possible world of r in which the attribute set X forms a valid key (i.e., all tuples are uniquely identified).
- A *certain key* $c\langle X\rangle$ is satisfied by a relation r if in *every* possible world of r, the attribute set X forms a valid key.

These formal definitions aim to reconcile the logical inconsistencies of `NULL` handling in SQL by providing a principled approach to reasoning about keys under incomplete information.

2.3.2 NULL Values and Functional Dependencies

`NULL` values in SQL are subject to two predominant interpretations: (1) the value exists but is currently unknown, and (2) there is no information available about the value[14]. These interpretations significantly influence the semantics of relational constraints, particularly Functional Dependencies(FDs), when applied to incomplete relations.

When the first interpretation is adopted - i.e., the value exists but is unknown - two variations of FDs have been introduced to capture their behavior under uncertainty: Strong Functional Dependencies(SFDs) and Weak Functional Dependencies(WFDs). An SFD is said to hold in an incomplete relation if, in *every* possible world (i.e., every instantiation of the relation where `NULL` values are replaced by concrete values from their domains), the FD is satisfied in the traditional sense. In contrast, a WFD holds if there exists *at least one* such possible world where the FD holds.

Mark Levene and George Loizou investigated the logical properties of SFDs and WFDs, focusing on their axiomatization and implication[28]. Their results show that the combined class of SFDs and WFDs does not admit a finite axiomatization. Moreover, it has been shown that for certain sets of SFDs and WFDs, Armstrong relations - used to characterize implication - may not exist[14].

Under the second interpretation of `NULL`, which encompasses both unknown and genuinely non-existent values, Lien proposed an alternative semantics for FDs[29]. In this framework, a functional dependency is said to hold if *strong similarity* on the left-hand side (LHS) of the dependency implies equality on the right-hand side (RHS). This formulation is better suited to real-world scenarios where incomplete information must still support meaningful constraints.

Building upon these foundations, further research has proposed generalized notions of dependencies that integrate Functional Dependencies, Keys, and `NOT NULL` constraints into a unified framework. Specifically, the concepts of Possible Functional Dependencies(p-FDs) and Certain Functional Dependencies(c-FDs) have been introduced[23].

A relation r satisfies a *possible functional dependency* (p-FD) $X \to Y$ if, for all tuples $t, t' \in r$, whenever t and t' have *identical and complete* values on all attributes in X, they also have identical values on all attributes in Y.

Two tuples $t, t' \in r$ are said to be *weakly similar* on an attribute A if either $t(A) = t'(A)$, or one of $t(A)$ or $t'(A)$ is `NULL`. A relation r satisfies a *certain functional dependency* (c-FD) $X \to Y$ if, whenever two tuples have same values on all attributes in X, they are weakly similar on all attributes in Y. In this way, c-FDs tolerate the presence of `NULL` values while still supporting meaningful dependency reasoning.

These generalized dependency models provide a robust foundation for enforcing data integrity in modern databases, where uncertainty, incompleteness, and partial information are increasingly the norm.

Chapter 3

Combining partial Armstrong databases

As discussed earlier, Armstrong databases do not always exist, particularly in complex scenarios involving combined classes of constraints such as functional dependencies (FDs) and inclusion dependencies (INDs). For the general case where both FDs and INDs are unrestricted, it has been shown that Armstrong databases may not exist. Furthermore, even when FDs are restricted to standard forms, and INDs limited to binary, the implication problem remains undecidable[30]. Without a sound axiomatization or alternative systematic approaches, reasoning about implication and constructing Armstrong databases becomes highly non-trivial.

However, if we impose additional restrictions - namely, limiting FDs to standard and INDs to typed - both of which are faithful in the sense defined by Fagin[13], the situation improves considerably. Under these constraints, it has been proven that Armstrong databases not only always exist[7], but are also computable[13].

In Section 2.2.4, we introduced the concept of Direct Product, a construction employed to generate Armstrong databases for mixed classes of dependencies. This method, however, comes with a downside: the resulting Armstrong databases can be exponentially large, due to the reliance on Cartesian Product operations during construction.

To address this scalability issue, we explored optimization strategies that significantly reduce the size of the resulting database. Specifically, we developed a method that reduces the construction size from exponential to polynomial, making the generation of Armstrong databases more practical for real-world applications.

3.1 Disjoint Union

We adopt the *Disjoint Union* as the underlying operator in the construction of Direct Products (see Definition 2.2.10), in contrast to the conventional use of the Cartesian Product. The *Disjoint Union* is defined as follows:

Given a family of relations that share the same schema, the Disjoint Union is formed by first transforming each relation into an isomorphic copy such that all data values across different relations are pairwise disjoint. That is, no value appearing in one relation occurs in any of the others. Once this transformation is complete, a new relation is constructed by taking the set-theoretic union of all tuples from these modified relations. This approach ensures that the provenance or source of each tuple remains distinguishable, even though the original attribute names and structure are preserved.

To illustrate this idea, consider the following example. Let schema $R = [\text{Item}(I), \text{Source}(S), \text{CostMethod}(C)]$ (Table 1.3 on page 4 is a relation over R), and let $T = [\text{CostMethod}(C), \text{Update}(U)]$. Suppose the constraint set is $\Sigma = \{I \to SC,\ R[C] \subseteq T[C]\}$. Using a standard chase procedure, it can be verified that the following dependencies are *not* implied by Σ:

$$\pi_1 : S \to C, \quad \pi_2 : C \to S, \quad \pi_3 : T[C] \subseteq R[C]$$

These dependencies (among others) form part of the set Θ, which contains all dependencies not implied by Σ. For simplicity, we select only these three dependencies to demonstrate the impact on the size of Armstrong databases.

Below are partial Armstrong relations that satisfy Σ but violate each π_j:

r_1:

Item(I)	Source(S)	CostMethod(C)
i_1	s_1	c_1
i_2	s_1	c_2

t_1:

CostMethod(C)	Update(U)
c_1	u_1
c_2	u_2

Table 3.1: Partial Armstrong Relations on $\pi_1(S \to C)$

r_2:

Item(I)	Source(S)	CostMethod(C)
i_3	s_2	c_3
i_4	s_3	c_3

t_2:

CostMethod(C)	Update(U)
c_3	u_3

Table 3.2: Partial Armstrong Relations on $\pi_2(C \to S)$

r_3:

Item(I)	Source(S)	CostMethod(C)
i_5	s_4	c_4
i_6	s_5	c_5

t_3:

CostMethod(C)	Update(U)
c_4	u_4
c_5	u_5
c_6	u_6

Table 3.3: Partial Armstrong Relations on $\pi_3(T[C] \subseteq R[C])$

If we use the *Cartesian product* as the operator, the resulting Armstrong database is significantly larger:

r_4:

Item(I)	Source(S)	CostMethod(C)
$\langle i_1, i_3, i_5 \rangle$	$\langle s_1, s_2, s_4 \rangle$	$\langle c_1, c_3, c_4 \rangle$
$\langle i_1, i_3, i_6 \rangle$	$\langle s_1, s_2, s_5 \rangle$	$\langle c_1, c_3, c_5 \rangle$
$\langle i_1, i_4, i_5 \rangle$	$\langle s_1, s_3, s_4 \rangle$	$\langle c_1, c_3, c_4 \rangle$
$\langle i_1, i_4, i_6 \rangle$	$\langle s_1, s_3, s_5 \rangle$	$\langle c_1, c_3, c_5 \rangle$
$\langle i_2, i_3, i_5 \rangle$	$\langle s_1, s_2, s_4 \rangle$	$\langle c_2, c_3, c_4 \rangle$
$\langle i_2, i_3, i_6 \rangle$	$\langle s_1, s_2, s_5 \rangle$	$\langle c_2, c_3, c_5 \rangle$
$\langle i_2, i_4, i_5 \rangle$	$\langle s_1, s_3, s_4 \rangle$	$\langle c_2, c_3, c_4 \rangle$
$\langle i_2, i_4, i_6 \rangle$	$\langle s_1, s_3, s_5 \rangle$	$\langle c_2, c_3, c_5 \rangle$

t_4:

CostMethod(C)	Update(U)
$\langle c_1, c_3, c_4 \rangle$	$\langle u_1, u_3, u_4 \rangle$
$\langle c_1, c_3, c_5 \rangle$	$\langle u_1, u_3, u_5 \rangle$
$\langle c_1, c_3, c_6 \rangle$	$\langle u_1, u_3, u_6 \rangle$
$\langle c_2, c_3, c_4 \rangle$	$\langle u_2, u_3, u_4 \rangle$
$\langle c_2, c_3, c_5 \rangle$	$\langle u_2, u_3, u_5 \rangle$
$\langle c_2, c_3, c_6 \rangle$	$\langle u_2, u_3, u_6 \rangle$

Table 3.4: Result by Cartesian product

Now, compare this with the result of using the *Disjoint Union*:

r_5:

Item(I)	Source(S)	CostMethod(C)
i_1	s_1	c_1
i_2	s_1	c_2
i_3	s_2	c_3
i_4	s_3	c_3
i_5	s_4	c_4
i_6	s_5	c_5

t_5:

CostMethod(C)	Update(U)
c_1	u_1
c_2	u_2
c_3	u_3
c_4	u_4
c_5	u_5
c_6	u_6

Table 3.5: Result by Disjoint union

As demonstrated in above discussion, Armstrong databases constructed using the *Disjoint Union* approach maintain satisfaction of the original constraint set Σ while explicitly violating only the selected non-implied dependencies π_j. Importantly, this method results in a significant reduction in the size of the database when compared to those constructed using the *Cartesian Product* method. The key advantage of the Disjoint Union approach lies in its ability to simplify the database structure while still adhering to the required dependencies, thus avoiding the

exponential growth in size typically associated with Cartesian products.

This benefit becomes even more pronounced in scenarios involving a large number of non-implied dependencies, which is common in complex cases where both functional dependencies (FDs) and inclusion dependencies (INDs) are present. In such combined constraint settings, the number of dependencies that are not implied by the given set can be substantial, leading to a corresponding increase in the potential size of Armstrong databases. The Disjoint Union method mitigates this issue by offering a more compact representation, reducing both the storage requirements and computational overhead associated with database construction. Most importantly, a small size is critical for manual inspection.

3.2 Proof of faithfulness with respect to Disjoint Union

In Chapter 1, we introduced the concept of classifying constraints as either standard or non-standard. This distinction is particularly important when analyzing keys and various types of dependencies in database theory. For instance, a standard key is defined as a key that is non-empty - meaning it includes at least one attribute. In contrast, an empty key, though theoretically possible, is considered non-standard due to its limited practical applicability.

A similar classification applies to functional dependencies (FDs). A standard FD is one where the left-hand side (i.e., the set of determining attributes) is non-empty. If the left-hand side is empty, the dependency is considered non-standard. This pattern of distinguishing standard from non-standard cases extends to other forms of dependencies as well, such as multivalued dependencies, join dependencies, and inclusion dependencies.

In general, standard dependencies tend to be more meaningful and applicable in practical database design and analysis. They align more closely with real-world data modeling requirements and are better supported by existing theoretical tools.

Before we can rigorously demonstrate the faithfulness of dependency preservation under disjoint union, we must first establish some additional formal definitions and notational conventions. These will provide the foundation necessary for the upcoming proofs and discussions.

Definition 3.2.1 (Disjoint Union of Relations)**.** Let $\{r_i\}_{i\in I}$ be a family of relations defined over the same set of attributes A. Assume that the data domains of different relations are disjoint - this can be achieved by systematically relabelling or tagging the data values in each r_i according to the index $i \in I$, so that no value occurs in more than one relation.

The *disjoint union* $\oplus_{i\in I}\{r_i\}$ is defined as the relation r with attributes A whose set of tuples is the union of the transformed relations:

$$r = \bigcup_{i\in I} \tilde{r}_i,$$

where each $\tilde{r}_i$ is an isomorphic copy of r_i in which the data values have been renamed to enforce disjointness across all $i \in I$. △

Remark 3.2.1. The construction of the disjoint union ensures that tuple values uniquely identify their origin. Let $r = \oplus_{i\in I}\{r_i\}$, and consider two tuples $t_1, t_2 \in r$. If t_1 and t_2 agree on the value of any attribute, then it follows that both tuples must have originated from the same relation r_i for some $i \in I$. This is a direct consequence of the disjointness of value domains: overlap in attribute values implies common provenance. This property enables tracking the source relation of any tuple in the union without needing explicit provenance annotations.

Definition 3.2.2 (Upward Faithful Dependency)**.** A dependency σ is said to be *upward faithful* with respect to *disjoint union* if the following condition holds:

Let $\{r_i\}_{i\in I}$ be a family of relation instances over a common schema. If the dependency σ holds in each individual relation r_i for all $i \in I$, then σ also holds in their disjoint union $\bigoplus_{i\in I} r_i$.

That is, if σ is satisfied independently in every component relation, then it remains valid when these relations are combined via disjoint union.

△

Definition 3.2.3 (Downward Faithful Dependency)**.** A dependency σ is said to be *downward faithful* with respect to *disjoint union* if the following condition holds:

Let $\{r_i\}_{i\in I}$ be a family of relation instances over a common schema, and let $r = \bigoplus_{i\in I} r_i$ be their disjoint union. If the dependency σ holds in r, then it also holds in each individual relation instance r_i for every $i \in I$.

In other words, if σ holds for the union of the data, then it is guaranteed to hold for each part.

△

Definition 3.2.4 (Faithful Dependency)**.** A dependency σ is said to be *faithful* with respect to *disjoint union* if it is both upward faithful and downward faithful.

That is, the dependency is preserved when moving in both directions between a set of disjoint relation instances and their union.

△

Theorem 3.2.1. *Standard functional dependencies are faithful with respect to disjoint union.*

Proof. Let $\sigma\colon X \to Y$ be a standard functional dependency over a schema R, and let $\{r_i\}_{i\in I}$ be a family of relation instances over R. Define the disjoint union of these relations as $r = \bigoplus_{i\in I} r_i$.

Upward Faithfulness. Assume for contradiction that σ holds in each r_i (for all $i \in I$), but σ does not hold in r. Then, by definition, there exist tuples $t_1, t_2 \in r$ such that:

$$t_1[X] = t_2[X] \quad \text{but} \quad t_1[Y] \neq t_2[Y].$$

Since r is a disjoint union, any two tuples in r must originate from the same or different component relations. However, since X is non-empty (as σ is standard), the equality $t_1[X] = t_2[X]$ implies that t_1 and t_2 must have come from the *same* relation r_j for some $j \in I$. This is because in a disjoint union, identical values over a non-empty attribute set X cannot appear across components due to disjointness in identifiers. But then $t_1, t_2 \in r_j$, and σ fails in r_j, contradicting the assumption that σ holds in each r_i. Thus, σ must hold in r, and so σ is upward faithful.

Downward Faithfulness. Assume for contradiction that σ holds in r, but there exists some index $j \in I$ such that σ does not hold in r_j. Then there exist tuples $t_1, t_2 \in r_j$ such that:

$$t_1[X] = t_2[X] \quad \text{but} \quad t_1[Y] \neq t_2[Y].$$

Since r_j is a subset of r (as part of the disjoint union), t_1 and t_2 are also in r. This contradicts the assumption that σ holds in r. Therefore, σ must hold in each r_i, and hence it is downward faithful.

Since σ is both upward and downward faithful, it follows that σ is faithful. □

Remark 3.2.2. *Non-standard* functional dependencies can fail to be upward faithful.

Consider the following example. Let r_1 and r_2 be two relations over the schema $\{Y\}$ (i.e., a single attribute). Suppose:

$$r_1\colon \begin{array}{|c|}\hline \mathrm{Y} \\ \hline y_1 \\ \hline \end{array} \qquad r_2\colon \begin{array}{|c|}\hline \mathrm{Y} \\ \hline y_2 \\ \hline \end{array}$$

and consider the dependency $\sigma\colon \emptyset \to Y$, which asserts that all tuples in any relation must have the same value of Y.

Clearly, σ holds in both r_1 and r_2 individually, since each contains only one tuple. However, in the disjoint union $r = r_1 \oplus r_2$, we have:

$$r = \oplus\{r_1, r_2\}\colon \begin{array}{c} \mathrm{Y} \\ \hline y_1 \\ y_2 \end{array}$$

so σ no longer holds, as the Y values differ. Hence, σ is not upward faithful.

This example demonstrates that the restriction to standard functional dependencies is necessary for the theorem to hold.

Definition 3.2.5 (Standard and Non-Standard Multivalued Dependency)**.** Let $X \twoheadrightarrow Y$ be a multivalued dependency (MVD) over a relational schema R.

- The dependency is called *standard* if the left-hand side X is non-empty, i.e., $X \neq \emptyset$.
- The dependency is called *non-standard* if the left-hand side is empty, i.e., $X = \emptyset$.

The distinction between standard and non-standard multivalued dependencies is important from both a theoretical and practical perspective. Standard MVDs express meaningful constraints about how values in Y vary independently of values in $R \setminus (X \cup Y)$, conditioned on shared values of X. Non-standard MVDs, in contrast, make a global statement about value independence across the entire relation, without any conditioning context, and are generally considered less useful in practice.

△

Theorem 3.2.2. *Standard multivalued dependencies are faithful with respect to disjoint union.*

Proof. Let $\sigma\colon X \twoheadrightarrow Y$ be a *standard* multivalued dependency over a relational schema R. Let $U = R \setminus (X \cup Y)$ denote the remaining attributes. Let $\{r_i\}_{i \in I}$ be a family of relation instances over R, and let $r = \bigoplus_{i \in I} r_i$ denote their disjoint union.

Upward Faithfulness. Assume that σ holds in every r_i for all $i \in I$. We aim to show that σ also holds in r.

Suppose, for contradiction, that σ does not hold in r. Then there exist tuples $t_1, t_2 \in r$ such that:

$$t_1[X] = t_2[X] = x_1, \quad t_1[Y] = y_1 \neq y_2 = t_2[Y], \quad t_1[U] = u_1, \quad t_2[U] = u_2.$$

Since $X \neq \emptyset$ and $t_1[X] = t_2[X] = x_1$, and the union is disjoint, both t_1 and t_2 must originate from the same component relation r_j for some $j \in I$.

Because σ holds in r_j, the MVD condition requires that the tuples

$$t_3 = (x_1, y_2, u_1) \quad \text{and} \quad t_4 = (x_1, y_1, u_2)$$

must also be present in r_j. Since $r_j \subseteq r$, these tuples are also in r, and thus the condition required by σ is satisfied in r. This contradicts the assumption that σ does not hold in r.

Hence, σ must hold in r, and therefore it is upward faithful.

Downward Faithfulness. Assume that σ holds in $r = \bigoplus_{i \in I} r_i$, and we want to show that σ holds in each individual r_j.

Suppose, for contradiction, that there exists some $j \in I$ such that σ does not hold in r_j. Then there exist tuples $t_1, t_2 \in r_j$ such that:

$$t_1[X] = t_2[X] = x_1, \quad t_1[Y] = y_1 \neq y_2 = t_2[Y], \quad t_1[U] = u_1, \quad t_2[U] = u_2,$$

but at least one of the tuples

$$t_3 = (x_1, y_2, u_1) \quad \text{or} \quad t_4 = (x_1, y_1, u_2)$$

is not present in r_j.

Since $r_j \subseteq r$ and $t_1, t_2 \in r$, and σ holds in r, then both t_3 and t_4 must be in r. But recall that $X \neq \emptyset$ and $t_3[X] = x_1$; thus, all tuples with $X = x_1$ must belong to the same component r_j due to the disjointness of the union on X values. Therefore, t_3 and t_4 must also be in r_j.

This contradicts the assumption that σ does not hold in r_j. Hence, σ must hold in every r_i, and therefore it is downward faithful.

Since σ is both upward and downward faithful, it follows that σ is faithful with respect to disjoint union. □

Remark 3.2.3. *Non-standard* multivalued dependencies may fail to be upward faithful.

Consider the non-standard dependency $\sigma\colon \emptyset \twoheadrightarrow Y$ over the schema $R = \{Y, U\}$, and let the relation instances r_1 and r_2 be defined as follows:

r_1:

Y	U
y_1	u_1

r_2:

Y	U
y_2	u_2

In both r_1 and r_2, the dependency $\emptyset \twoheadrightarrow Y$ holds trivially, since there is only one tuple in each relation.

However, in their disjoint union:

$r = \oplus\{r_1, r_2\}$:

Y	U
y_1	u_1
y_2	u_2

the dependency no longer holds. According to the semantics of $\emptyset \twoheadrightarrow Y$, for every pair of tuples, all combinations of Y and U values should be present. That would require the tuples (y_1, u_2) and (y_2, u_1) to also exist in r, which they do not.

Thus, σ does not hold in the union, demonstrating that non-standard multivalued dependencies are not necessarily upward faithful.

Definition 3.2.6 (Standard and Non-Standard Join Dependency)**.** Let σ be a join dependency over a relational schema $R = \{A_1, A_2, \ldots, A_m\}$, expressed as:

$$\sigma\colon X_1 \bowtie X_2 \bowtie \cdots \bowtie X_n,$$

where each $X_j \subseteq R$ for $1 \leq j \leq n$ denotes a subset of attributes from R. That is, σ requires that a relation over R can be reconstructed by taking the natural join of its projections over the sets $X_1, X_2, \ldots, X_n$.

The structure of σ can be represented using a *hypergraph*, where:

- Each attribute A_l $(1 \leq l \leq m)$ is a vertex.
- Each subset X_j $(1 \leq j \leq n)$ is a hyperedge connecting the corresponding attributes.

We define the standardness of the join dependency σ based on the connectivity of this hypergraph:

- σ is called *standard* if the hypergraph is connected - that is, there is a path between every pair of attributes via the hyperedges.
- σ is called *non-standard* if the hypergraph is disconnected - meaning the schema can be partitioned into isolated attribute subsets that do not interact in the join structure.

Let Γ be a relation over R, and let $\gamma_1, \gamma_2, \ldots, \gamma_n$ be its projections over $X_1, X_2, \ldots, X_n$, respectively. That is:

$$\gamma_j = \Pi_{X_j}(\Gamma), \quad \text{for } 1 \leq j \leq n.$$

We say that Γ *satisfies* the join dependency σ if:

$$\Gamma = \gamma_1 \bowtie \gamma_2 \bowtie \cdots \bowtie \gamma_n,$$

i.e., Γ is equal to the *lossless join* of its projections. △

Theorem 3.2.3. *Standard join dependencies are faithful with respect to disjoint union.*

Proof. Let $\sigma = \bowtie \{X_1, X_2, \ldots, X_n\}$ be a standard join dependency over a relational schema R, and let $\{r_i\}_{i \in I}$ be a family of relations over R. Define the disjoint union $r = \bigoplus_{i \in I} r_i$.

By definition, a join dependency holds in a relation r if:

$$r = \Pi_{X_1}(r) \bowtie \Pi_{X_2}(r) \bowtie \ldots \bowtie \Pi_{X_n}(r).$$

Define the following:

$$\begin{aligned} E &= \Pi_{X_1}(r) \bowtie \Pi_{X_2}(r) \bowtie \ldots \bowtie \Pi_{X_n}(r), \\ e_i &= \Pi_{X_1}(r_i) \bowtie \Pi_{X_2}(r_i) \bowtie \ldots \bowtie \Pi_{X_n}(r_i), \quad \text{for each } i \in I, \\ \tilde{E} &= \bigoplus_{i \in I} e_i. \end{aligned}$$

We will prove that $E = \tilde{E}$, which establishes the equivalence between computing the join over the disjoint union r and computing the join over each component r_i individually, then recombining the results.

Claim 1: $E \subseteq \tilde{E}$.

Let $t \in E$ be an arbitrary tuple. By construction of the join, for each $j \in \{1, \ldots, n\}$, we have $t[X_j] \in \Pi_{X_j}(r)$. Since $r = \bigoplus_{i \in I} r_i$, for each j there exists an index i_j such that $t[X_j] \in \Pi_{X_j}(r_{i_j})$.

Since σ is standard, the hypergraph formed by the sets $\{X_1, \ldots, X_n\}$ is connected. Now, consider any two sets X_j and $X_{j'}$ such that $X_j \cap X_{j'} \neq \emptyset$. Because the union used to construct the tuple t is disjoint across the relations r_i, the overlapping values in $t[X_j]$ and $t[X_{j'}]$ must originate from the same relation r_i. This implies that both $t[X_j]$ and $t[X_{j'}]$ are entirely contained in some common r_i.

By the connectedness of the hypergraph, this reasoning extends to all the sets $X_1, \ldots, X_n$. Therefore, all projections $t[X_j]$ must lie in the same relation r_i, for some $i \in I$.

Therefore, $t \in e_i \subseteq \tilde{E}$, and we conclude that $E \subseteq \tilde{E}$.

Claim 2: $\tilde{E} \subseteq E$.

Let $t \in \tilde{E}$. Then $t \in e_i$ for some $i \in I$, and thus:

$$t \in \Pi_{X_1}(r_i) \bowtie \Pi_{X_2}(r_i) \bowtie \ldots \bowtie \Pi_{X_n}(r_i).$$

Since $r_i \subseteq r$, we have $\Pi_{X_j}(r_i) \subseteq \Pi_{X_j}(r)$ for each j, and so $t \in E$.

Hence, $\tilde{E} \subseteq E$, and therefore $E = \tilde{E}$.

Upward Faithfulness.

Suppose σ holds for each r_i, i.e., $e_i \subseteq r_i$ for all $i \in I$. Then:

$$E = \tilde{E} = \bigoplus_{i \in I} e_i \subseteq \bigoplus_{i \in I} r_i = r.$$

Thus, σ holds for r, and we conclude that σ is upward faithful.

Downward Faithfulness.

Suppose σ holds for r, i.e., $E \subseteq r$. Since $E = \tilde{E}$, we have:

$$\bigoplus_{i \in I} e_i = \tilde{E} = E \subseteq r = \bigoplus_{i \in I} r_i.$$

Because the union is disjoint, it follows that $e_i \subseteq r_i$ for all $i \in I$. Hence, σ holds for each r_i, and σ is downward faithful.

Since σ is both upward and downward faithful, it follows that standard join dependencies are faithful. □

Remark 3.2.4. *Non-standard* join dependencies may fail to be upward faithful.

Consider the non-standard join dependency $\sigma\colon AB \bowtie BC \bowtie DE$ over schema $R = \{A, B, C, D, E\}$. The hypergraph formed by $\{AB, BC, DE\}$ is disconnected, since $\{A, B, C\}$ and $\{D, E\}$ form two separate components.

Define two relations r_1 and r_2 as follows:

r_1:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_2	d_1	e_1

r_2:

A	B	C	D	E
a_2	b_2	c_3	d_2	e_2
a_3	b_2	c_3	d_2	e_2

Each of r_1 and r_2 satisfies σ individually. However, consider their disjoint union:

$$r = \bigoplus\{r_1, r_2\}.$$

In this union, the join dependency σ no longer holds.

$r = \oplus\{r_1, r_2\}$:

A	B	C	D	E
a_1	b_1	c_1	d_1	e_1
a_1	b_1	c_2	d_1	e_1
a_2	b_2	c_3	d_2	e_2
a_3	b_2	c_3	d_2	e_2

The join of projections $\Pi_{AB}(r)$, $\Pi_{BC}(r)$, and $\Pi_{DE}(r)$ can produce combinations that are not in r, thereby violating the lossless join condition.

Thus, non-standard join dependencies may not be upward faithful.

Definition 3.2.7 (Standard and Non-Standard Inclusion Dependency)**.** Let $M[X] \subseteq N[Y]$ be an *inclusion dependency* between two relation schemata M and N, where $X \subseteq \text{attr}(M)$ and $Y \subseteq \text{attr}(N)$ are sequences of attributes of the same length. This dependency expresses that the projection of relation schema M onto attribute list X is a subset of the projection of relation schema N onto attribute list Y:

$$\Pi_X(M) \subseteq \Pi_Y(N).$$

We classify inclusion dependencies as follows:

- The inclusion dependency $M[X] \subseteq N[Y]$ is called *standard* if both $X \neq \emptyset$ and $Y \neq \emptyset$, i.e., the inclusion constraint involves at least one attribute from each relation.
- It is called *non-standard* if $X = \emptyset$ and $Y = \emptyset$, meaning it trivially compares the empty projections of M and N. Since $\Pi_\emptyset(M)$ and $\Pi_\emptyset(N)$ are always either empty or contain a single empty tuple (depending on whether the relations are empty), such dependencies tend to hold trivially or convey no meaningful constraint. △

Theorem 3.2.4. *Standard inclusion dependencies are faithful with respect to disjoint union.*

Proof. Let $\sigma\colon M[X] \subseteq N[Y]$ be a standard inclusion dependency over a database schema $\mathbf{R}$, where M and N are relation symbols, and X, Y are non-empty sequences of attributes with $|X| = |Y|$. Let $\{r_i\}_{i\in I}$ be a family of database instances over $\mathbf{R}$, and let $r = \bigoplus_{i\in I} r_i$ denote their disjoint union. We denote the interpretation of relation schema M in instance r_i as m_i, and of N as n_i, and similarly m and n in r.

Upward faithfulness. Assume that σ holds in every instance r_i, i.e., for each $i \in I$, we have $\Pi_X(m_i) \subseteq \Pi_Y(n_i)$. We must show that $\Pi_X(m) \subseteq \Pi_Y(n)$.

Let $x \in \Pi_X(m)$. Then $x = t[X]$ for some tuple $t \in m$. Since r is a disjoint union of the r_i, this tuple t must originate from some $m_j \in r_j$ for some $j \in I$, and thus $x \in \Pi_X(m_j)$. By the assumption that σ holds in r_j, we have

$\Pi_X(m_j) \subseteq \Pi_Y(n_j)$, so there exists $t' \in n_j$ such that $t'[Y] = x$. Since $n_j \subseteq n$, it follows that $x \in \Pi_Y(n)$.

Hence, every $x \in \Pi_X(m)$ also belongs to $\Pi_Y(n)$, so σ holds in r. Therefore, σ is *upward faithful*.

Downward faithfulness. Assume that σ holds in the disjoint union r, i.e., $\Pi_X(m) \subseteq \Pi_Y(n)$. We must show that for every $j \in I$, the inclusion $\Pi_X(m_j) \subseteq \Pi_Y(n_j)$ also holds.

Let $x \in \Pi_X(m_j)$ for some $j \in I$. Since $m_j \subseteq m$, we have $x \in \Pi_X(m)$. By assumption, $\Pi_X(m) \subseteq \Pi_Y(n)$, so there exists a tuple $t' \in n$ such that $t'[Y] = x$. Because r is a disjoint union, and x originated from r_j, and X, Y are non-empty (i.e., contain identifying information), t' must belong to n_j, meaning $x \in \Pi_Y(n_j)$.

Thus, $\Pi_X(m_j) \subseteq \Pi_Y(n_j)$ for each $j \in I$, and so σ holds in each r_j. Therefore, σ is *downward faithful*.

Since σ is both upward and downward faithful, it is *faithful*. □

Remark 3.2.5. There exist *non-standard* inclusion dependencies that are not downward faithful. Consider a database schema $\mathbf{R}$ consisting of two unary relations $M[X]$ and $N[Y]$. Let $\sigma\colon M[\,] \subseteq N[\,]$ be a non-standard inclusion dependency, involving empty attribute sequences.

Define two database instances r_1 and r_2 over $\mathbf{R}$ as follows:

$$r_1.m_1\colon \begin{array}{|c|}\hline \text{X} \\ \hline x_1 \\ \hline \end{array} \qquad r_1.n_1\colon \begin{array}{|c|}\hline \text{Y} \\ \hline \\ \hline \end{array} \qquad r_2.m_2\colon \begin{array}{|c|}\hline \text{X} \\ \hline \\ \hline \end{array} \qquad r_2.n_2\colon \begin{array}{|c|}\hline \text{Y} \\ \hline y_2 \\ \hline \end{array}$$

The disjoint union r gives:

$$r.m = \oplus\{m_1, m_2\}\colon \begin{array}{|c|}\hline \text{X} \\ \hline x_1 \\ \hline \end{array} \qquad r.n = \oplus\{n_1, n_2\}\colon \begin{array}{|c|}\hline \text{Y} \\ \hline y_2 \\ \hline \end{array}$$

Since both $\Pi_\emptyset(r.m)$ and $\Pi_\emptyset(r.n)$ yield the singleton set containing the empty tuple, the non-standard inclusion dependency σ holds in r. However, σ does not hold for relations r_1.

Thus, σ is not downward faithful. This highlights how non-standard inclusion dependencies behave differently from standard ones and typically lack meaningful semantic guarantees.

From the results above, we conclude that all *standard dependencies* - including functional dependencies (FDs), multivalued dependencies (MVDs), join dependencies (JDs), and inclusion dependencies (INDs) - are *faithful* with respect to disjoint union. In contrast, *non-standard dependencies* do not generally exhibit this property.

This distinction has significant implications for the construction of Armstrong databases. Specifically, if the constraint set Σ contains any non-standard dependencies, the disjoint union becomes unsuitable for use in generating Armstrong relations for such sets Σ.

Fortunately, non-standard dependencies are rarely encountered in practical database design. For example, consider a non-standard key dependency $\sigma = \{\emptyset\}$, which asserts that the empty set of attributes forms a key. Any relation r that satisfies this constraint can contain at most one tuple, since all tuples must agree on the empty set (i.e., be indistinguishable). Such constraints might occasionally arise in system configurations or special-purpose metadata, but they offer little value in typical applications involving structured, multi-tuple data.

Because of this, there is limited interest in constructing Armstrong tables that account for non-standard dependencies - especially in contexts where the primary goal is to minimize the size of such tables. Focusing on standard dependencies not only aligns with real-world database design but also ensures that disjoint union remains a useful tool for Armstrong database construction.

Chapter 4

Small Armstrong tables for keys and functional dependencies

Different strategies for constructing Armstrong tables are employed depending on the specific constraints of the database schema, particularly keys and functional dependencies (FDs). The complexity and size of an Armstrong table are often closely tied to the number of anti-sets, such as anti-keys and anti-FDs (see Definitions 2.2.3 and 2.2.7), which are derived from the given constraint sets. In practice, the usefulness of an Armstrong table can diminish as its size increases, particularly when it contains a large number of tuples.

Prior work has proposed basic methods for constructing Armstrong tables, often without regard for minimizing size. In this work, we investigate algorithmic strategies designed to construct Armstrong tables more efficiently, with a focus on minimizing size where possible, or at least ensuring that table size remains sub-linear with respect to the number of anti-sets.

4.1 Build Armstrong table

In this section, we present foundational methods for constructing Armstrong tables, with a focus on approaches that typically yield tables of linear size. Since the introduction of Armstrong databases by Fagin in his seminal work[4], extensive research has explored construction techniques tailored to various classes of constraints. These methods aim to generate representative instances that satisfy all intended dependencies while violating none of the unintended ones.

Our focus here is on the construction of Armstrong tables specifically for keys and functional dependencies (FDs) - two of the most fundamental and widely used types of integrity constraints in database theory and practice. These constraints not only underpin the design and normalization of relational schemas but also play a critical role in ensuring data consistency and integrity.

4.1.1 Armstrong Table for Keys

We begin by presenting the construction of Armstrong tables for keys. An Armstrong table is a relation that satisfies a given set of constraints while simultaneously violating all constraints that are not logically implied by that set. In the case of keys, this requires constructing a table that satisfies the declared keys and violates all non-keys - referred to as *anti-keys*.

The construction process involves two main steps: computing the anti-keys and then generating tuples that violate them. The anti-keys are derived from the declared keys using the following procedure:

- Compute the *transversal* (as defined in Definition 2.2.4) of the key set. The transversal contains all minimal sets of attributes that intersect every key in the set.
- Take the *complements* of each transversal set with respect to the full attribute set R; these complements constitute the set of maximal anti-keys.

Example 1. Let the relation schema be $R = \{A, B, C\}$, and let the constraint set be $\Sigma = \{ABC\}$ as a key.

1. **Compute the transversal of the keys:**

$$T(\{ABC\}) = \{A, B, C\}$$

2. **Compute the complements of the transversal to get the anti-keys:**

$$MAX_\Sigma(R) = \{R \setminus A, R \setminus B, R \setminus C\} = \{BC, AC, AB\}$$

The next step is to construct tuples that *agree* on each of the anti-keys, i.e., tuples that share the same values on the attributes of a given anti-key. This ensures that these attribute sets do not form keys in the relation, as they fail to uniquely identify tuples. In practice, such tuple pairs can be generated systematically by assigning values that enforce agreement on each anti-key, often by deriving them from adjacent tuple pairs[11].

A	B	C	
a_1	b_1	c_1	AB
a_1	b_1	c_2	
a_2	b_1	c_2	BC
a_2	b_2	c_2	AC

$R = ABC$
$MAX_\Sigma(R) = \{AB, BC, AC\}$

Figure 4.1: Armstrong table for Example 1

4.1.2 Armstrong Tables for Functional Dependencies (FDs)

Constructing an Armstrong table for a set of functional dependencies (FDs) involves identifying all anti-FDs - those dependencies not implied by the given set Σ. As with keys, the first step in the construction process is to compute the anti-FDs, with a particular focus on identifying their maximal forms. Below, we review several foundational concepts essential to this construction:

- **Closed Sets:** For ease of reference, we recall the notion of a *closed set* (see Definition 2.2.6): a set of attributes that functionally determines only attributes within itself under the dependency set Σ. The collection of all such closed subsets of R is denoted by $CL_\Sigma(R)$. These sets form the boundary within which Armstrong tables must operate. Among the closed sets are both *generated sets* and a subset known as *generator sets* (see Definition 2.2.9), which play a crucial role in identifying anti-FDs.

- **Maximal Sets of Anti-FDs:** For a given attribute $A \in R$, the maximal anti-FD sets are defined as:

$$MAX_\Sigma(A) = \{Y \subseteq R \,|\, Y \neq \emptyset,\, Y \nrightarrow A, \text{ and no } Y' \supset Y \text{ satisfies } Y' \nrightarrow A\}\,.$$

  Intuitively, each such set Y is a maximal subset of attributes that fails to determine A. The union of all such sets over attributes in R, denoted by $MAX_\Sigma(R)$, represents the complete set of anti-FDs. Notably, it has been shown that $MAX_\Sigma(R) = GEN_\Sigma(R)$[2].

- **Closure Computation:** The *closure* of a set of attributes (see Definition 1.1.8) is computed as the intersection of all maximal anti-FD sets that contain the given set. Closure computation is used to identify which attributes are implied by a candidate determinant under Σ.

After computing the maximal anti-FD sets, we proceed to construct the Armstrong table. This involves generating a set of tuple pairs hat agree on specific attribute combinations. Let $agr(r)$ denote the set of attribute sets on which some pair of tuples in relation r agree.

A relation r is an Armstrong table for the FD set Σ if and only if the following condition holds:

$$MAX_\Sigma(R) \subseteq agr(r) \subseteq CL_\Sigma(R).$$

This condition ensures that:

- All anti-FDs (i.e., maximal sets not implied by Σ) are violated by ensuring at least one pair of tuples agree on those attributes but differ elsewhere.

- All FDs implied by Σ are respected.

4.2 Build Armstrong Table in Sublinear Size

To reduce the size of Armstrong tables, we propose representing them using graphs where tuples correspond to vertices and agree sets are represented as labeled edges between tuples. This abstraction removes the dependence on specific data values and focuses solely on the equality relationships between attribute values across tuples.

The Armstrong table construction from Section 4.1 corresponds to a linear graph, shown in Figure 4.2. Each edge is labeled with the agree set shared between two adjacent tuples, and no two non-adjacent tuples share values on any attributes.

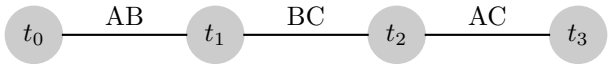


Figure 4.2: Graph representation of linear-size Armstrong table

Although this representation satisfies all the requirements of an Armstrong table, the number of tuples will grow linearly with the number of maximal sets. To better support domain experts in validation and analysis tasks, a more compact representation is often preferable.

Example 2. Let $R = ABCDEF$, and let $\Sigma = \{XY \mid X, Y \in R, X \neq Y\}$. Then:

$$MAX_\Sigma(R) = \{A, B, C, D, E, F\}, \quad m = |MAX_\Sigma(R)| = 6$$

Unimproved Armstrong Table

A	B	C	D	E	F
a_1	b_1	c_1	d_1	e_1	f_1
a_1	b_2	c_2	d_2	e_2	f_2
a_2	b_2	c_3	d_3	e_3	f_3
a_3	b_3	c_3	d_4	e_4	f_4
a_4	b_4	c_4	d_4	e_5	f_5
a_5	b_5	c_5	d_5	e_5	f_6
a_6	b_6	c_6	d_6	e_6	f_6

Size $= m + 1$

Sublinear Armstrong Table

A	B	C	D	E	F
a_1	b_1	c_1	d_1	e_1	f_1
a_1	b_2	c_2	d_2	e_2	f_2
a_2	b_2	c_1	d_3	e_3	f_3
a_3	b_3	c_3	d_1	e_2	f_3

$O(\sqrt{m}) <$ Size $\leq m$

Table 4.1: Reducing the size of Armstrong tables

In the corresponding sublinear graph (Figure 4.3), we maximize the number of agree sets per tuple pair. In the best case, n tuples can represent up to $\frac{n(n-1)}{2}$ agree sets.

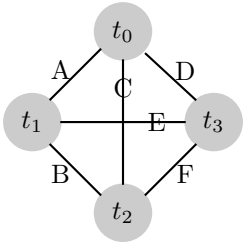


Figure 4.3: Graph representation of sublinear-size Armstrong table

However, sub-linear Armstrong tables are not guaranteed to exist for all sets of constraints. Whether a sub-linear size is achievable depends heavily on the interplay between the agree sets. The objective remains to construct Armstrong tables that are as small as possible - ideally with size growing sub-linearly relative to the number of agree sets.

To better understand how agree sets interact and constrain table construction, two key concepts are essential: *Near Cycles* and *Forced Sets*.

Definition 4.2.1 (Near Cycle)**.** Let G be an undirected graph representing an Armstrong table, where each vertex corresponds to a tuple, and edges are labeled with the agree set. A *Near Cycle* occurs in G when there exists a simple cycle (a closed path with no repeated vertices except the start and end) such that for some attribute A, the label on every edge in the cycle contains A except for one edge. In other words, attribute A is present on all but one edge of the cycle. △

In the graph below (Figure 4.4), there are two distinct Near Cycles:

- Attribute B appears on edges (t_0, t_1) and (t_1, t_2), but is absent from edge (t_0, t_2).
- Similarly, attribute C appears on edges (t_1, t_2) and (t_0, t_2), but is missing from edge (t_0, t_1).

Each forms a Near Cycle for the corresponding attribute.

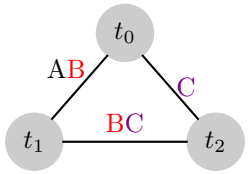


Figure 4.4: Example of Near Cycles

Agreement is transitive. For instance, if tuples t_0 and t_1 agree on attribute B, and t_1 and t_2 also agree on B, then by transitivity, t_0 and t_2 must agree on B as well. If this transitive edge (t_0, t_2) lacks B in its label, the consistency of the table is violated. Consequently, Armstrong table constructions must ensure that Near Cycles are detected and resolved, as their presence undermines the correctness of the agreement representation.

Definition 4.2.2 (Forced Set)**.** Let G be a graph representing an Armstrong table, where vertices correspond to tuples and each edge is labeled with the set of attributes on which the tuples agree (the *agree set*).

Suppose a *Near Cycle* is detected in G with respect to an attribute A; that is, in a simple cycle, attribute A appears on all but one edge. To preserve transitivity of agreement, A must also appear on the remaining edge. In this case, we say that A is *forced* on that edge. More generally, if multiple attributes are missing due to multiple Near Cycles, the set of all such attributes is called the *Forced Attribute Set* for the edge.

However, not all attribute subsets can serve as valid agree sets in Armstrong tables. To preserve the semantics of functional dependencies (FDs), agree sets must be *closed* under the set of FDs Σ. That is, if F is the Forced Attribute Set for an edge, we define the corresponding *Forced Set* as the closure F^* of F under Σ.

Thus, the Forced Set is the agree set that must be assigned to the edge, guaranteeing both transitivity of agreement and satisfaction of all functional dependencies in Σ. △

Forced Sets are central to the construction of Armstrong tables, ensuring that the interaction between tuples correctly captures the distinction between implied and non-implied dependencies. An Armstrong table must simultaneously satisfy the following two conditions:

- Violation of Non-Implied Keys and FDs (Anti-keys/FDs):
  Every functional dependency and key that is *not* implied by Σ must be explicitly violated in the table. To achieve this, the graph includes an edge for each agree set (also known as an anti-FD or anti-key). The agree set assigned to that edge contains exactly the attributes in the non-implied FD, ensuring that the two corresponding tuples agree on those attributes, but differ elsewhere - thus violating the dependency.
- Satisfaction of Implied Keys and FDs:
  It is essential to ensure that all keys and functional dependencies (FDs), along with their logical implications, are satisfied. Closed sets play a critical role in capturing these implications and ensuring they are properly enforced.

Here is a detailed example illustrating the step-by-step progression from the detection of Near Cycles and the enforcement of attribute sets, to the generation of a *Forced Set*.

Example 3. Let $R = ABCDE$, and let the set of functional dependencies be

$$\Sigma = \{A \to B,\ C \to D,\ BC \to A,\ BD \to A,\ AD \to C,\ BD \to C,\ E \to ABCD\}.$$

Then the maximal sets under Σ over R are:

$$MAX_\Sigma(R) = \{AB,\ B,\ CD,\ D,\ ABCD\}.$$

Figures 4.5 through 4.7 visualize how Forced Sets emerge during the construction of the Armstrong table for Example 3. These diagrams show how attributes propagate through the graph and how certain combinations can trigger new Near Cycles.

- **Step 1: Initial Edge Assignments and First Forced Attribute**
  In Figure 4.5, we:
  - begin by assigning attribute set AB to the edge (t_0, t_1),
  - followed by B on the edge (t_1, t_2).

  Since t_0 connects to t_1, and t_1 to t_2, this results in a transitive enforcement of B on the edge (t_0, t_2). The closure of B under Σ is just B, so the Forced Set on (t_0, t_2) is initially B.

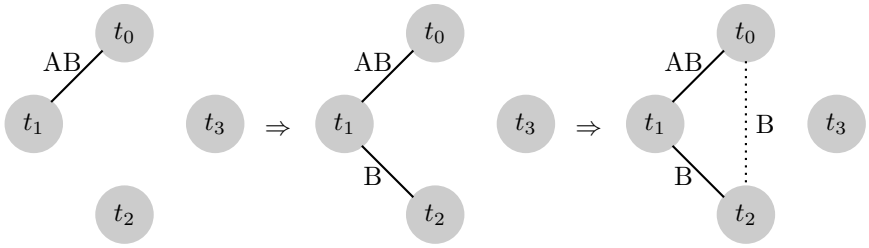


Figure 4.5: Example of Forced Set 1

- **Step 2: Additional Edge Assignments and Closure Expansion**
  In Figure 4.6, we assign attributes sets:
  - CD to edge (t_0, t_3),
  - D to edge (t_2, t_3).

  This results in D being added as another forced attribute on edge (t_0, t_2). At this point, the attribute set on edge (t_0, t_2) becomes BD. However, according to the functional dependencies $BD \to A$ and $BD \to C$, attributes A and C must now also be present. The transitive closure leads to the following derivation:

$$BD \to A, \quad BD \to C \quad \Rightarrow (BD)^* = ABCD.$$

  Hence, the full Forced Set on edge (t_0, t_2) becomes $ABCD$.

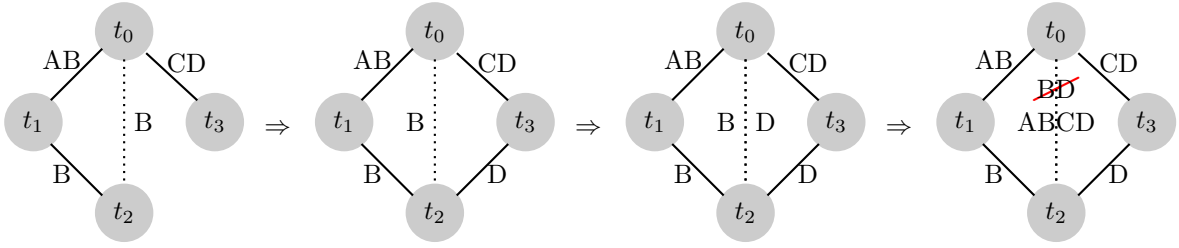


Figure 4.6: Example of Forced Set 2

- **Step 3: Emergence of New Near Cycles**

  As shown in Figure 4.7, the introduction of $ABCD$ as a Forced Set on edge (t_0, t_2) causes new Near Cycles involving attributes A and C. However, since edges (t_1, t_2) and (t_2, t_3) already have agree-sets assigned, simply adding the Forced Set $ABCD$ on edge (t_0, t_2) is not an option. To resolve this, attribute D must be removed from the edge (t_2, t_3) and reassigned to another edge.

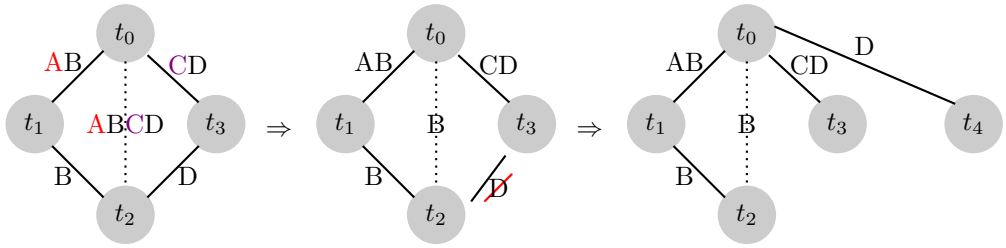


Figure 4.7: Example of Forced Set 3

Summary This example illustrates several critical mechanisms in the construction of Armstrong tables:

- Attribute values assigned to one edge can lead to the derivation of *forced attributes* on other edges through transitive agreement.
- The computation of attribute *closures* plays a central role in determining the final Forced Set associated with an edge.
- Newly generated Forced Sets can themselves introduce additional *Near Cycles*, which may invalidate previous assignments and require reevaluation or backtracking.

The construction process rigorously enforces the detection of Near Cycles and the computation of Forced Sets at each step. This often involves iterative adjustments: a newly identified Near Cycle may force the reassignment of attributes, which in turn affects the closure and potentially triggers new Near Cycles. The process continues until a stable configuration is reached - one in which all Near Cycles have been eliminated and all Forced Sets are correctly assigned. For instance, in Figure 4.7, although t_1 and t_3 are not directly connected, the structure ensures that no unintended attribute agreement is introduced through transitive relationships.

Once a Forced Set is finalized for an edge, only those agree sets in $MAX_\Sigma(R)$ that are equal to or supersets of the Forced Set may be assigned to that edge. After all agree sets have been assigned, the final graph can be used to construct the Armstrong table.

This disciplined approach provides a reliable framework for constructing Armstrong tables that satisfy all required dependencies while violating exactly those that are not implied. The next challenge lies in minimizing the overall size of such Armstrong table.

Chapter 5

Implementation for Sublinear Armstrong Table

In this chapter, we explore a range of strategies aimed at constructing the smallest possible Armstrong table corresponding to a given collection of agree sets. Our primary objective is to generate a table whose size grows sublinearly - ideally, much slower than linearly - with respect to the number of agree sets provided. This pursuit involves identifying techniques that not only guarantee correctness, i.e., the table reflects precisely the specified agree sets, but also optimize for compactness and efficiency.

We examine both theoretical and practical methods for minimizing table size, including algorithmic heuristics and structural optimizations. Special attention is given to the inherent trade-offs between the size of the resulting table and the computational complexity. By systematically comparing different approaches, we aim to build a deeper understanding of the limitations and possibilities in generating minimal Armstrong tables[1].

5.1 Basic Algorithm

The algorithm accepts as input a collection of agree sets, which may be derived either from existing table instances or directly from schema-level specifications. Its primary goal is to construct an Armstrong table of sublinear size. If such a compact representation is achievable, the algorithm outputs the corresponding Armstrong table. Otherwise, it returns a failure signal, indicating that no sublinear solution exists under the given conditions.

An overview of the algorithm's workflow and key components is illustrated in Figure 5.1. The subsequent sections provide a detailed explanation of the algorithm's core logic, underlying principles, and decision-making processes that guide the construction of a sublinear Armstrong table.

1. **Initialize with a minimally sized graph, expanding only as necessary**

   The algorithm begins by constructing a graph with the smallest number of vertices sufficient to potentially represent all the provided agree sets. In an undirected graph with n vertices, the maximum number of possible edges is given by $\frac{n(n-1)}{2}$. Since each agree set corresponds to a required agreement relation, we need at least as many edges as there are agree sets. This gives the inequality:

   $$m \leq \frac{n(n-1)}{2}$$

   Solving for n yields a lower bound on the number of vertices required:

   $$n_{\min} = \left\lceil \frac{1 + \sqrt{1 + 8m}}{2} \right\rceil \quad \text{where} \quad m = |MAX_{\Sigma}(R)|$$

[1]The code for all experiments is available in the GitHub repository: https://github.com/WendyMYALE/SublinearArmstrongTable.git.

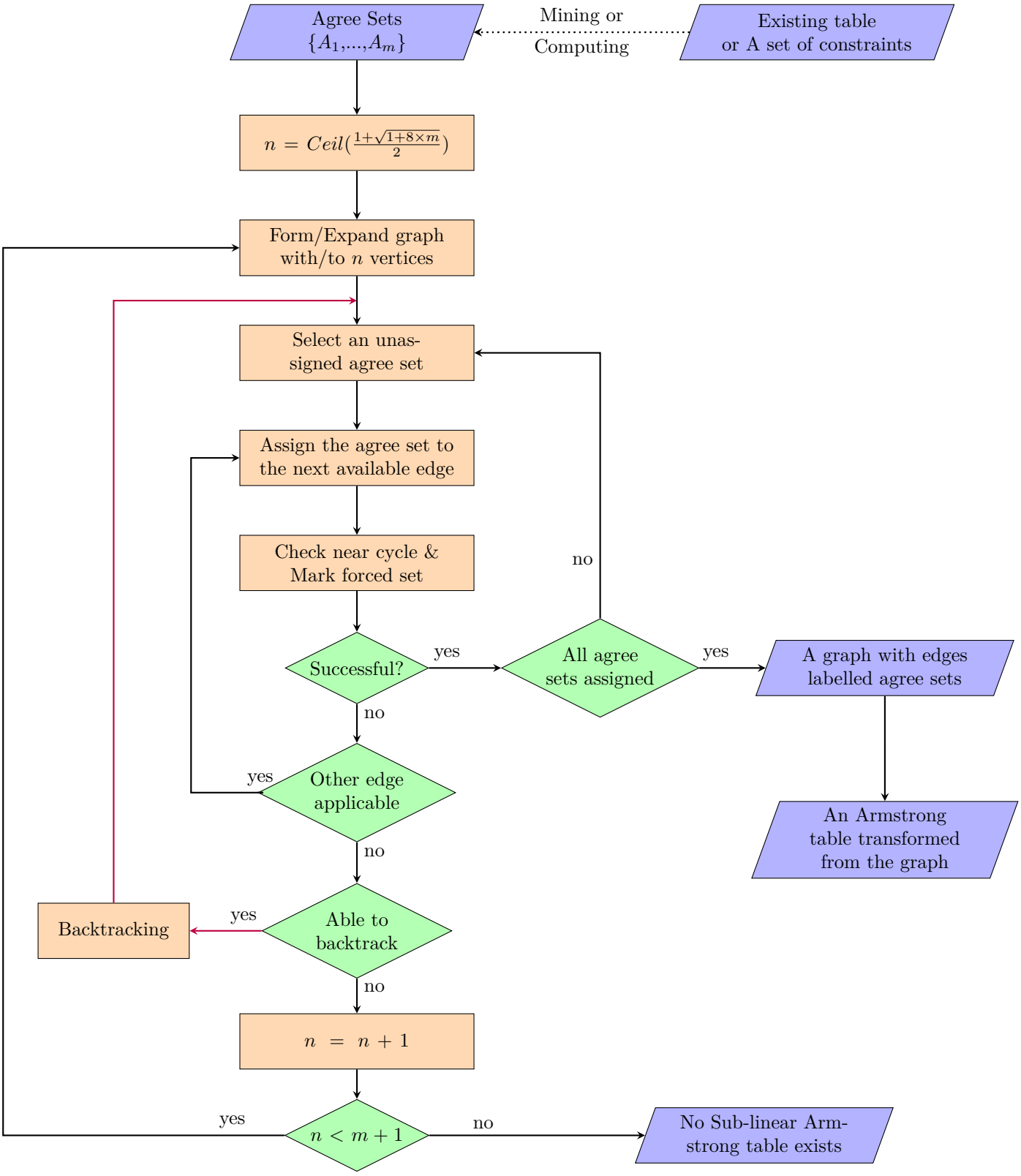


Figure 5.1: Algorithm overview

This value provides the theoretical minimum graph size needed to attempt constructing a valid Armstrong table. If the current graph fails to satisfy all agree sets, the algorithm incrementally increases the graph size by adding a vertex and repeating the construction process.

This expansion continues until either a valid Armstrong table is found or the graph reaches a size of $n = m+1$. At this linear size, it is guaranteed that an Armstrong table can be constructed by enforcing agree sets on all adjacent tuple pairs (see Figure 4.2).

2. **Construct a graph with n vertices and assign edge numbers in a specific order**

The next step involves constructing a graph with n vertices and assigning a unique index to each edge based on a chosen numbering scheme. Two numbering strategies were explored: *lexicographical* and *clique-wise* ordering.

Definition 5.1.1 (Lexicographical numbering order)**.** Edges are numbered based on the alphanumeric order of their endpoints, starting with the vertex associated with the earliest tuple (typically the smallest index). This total order is primarily determined by the smaller-indexed vertex, and secondarily by the larger-indexed vertex. △

For example, in a graph representing a table with four tuples (t_0, t_1, t_2, t_3), the lexicographical edge ordering is:

0 - (t_0, t_1), 1 - (t_0, t_2), 2 - (t_0, t_3), 3 - (t_1, t_2), 4 - (t_1, t_3), 5 - (t_2, t_3)

Definition 5.1.2 (Clique-wise numbering order)**.** Edges are numbered according to their participation in growing cliques. The numbering starts from a small complete subgraph (clique), and edges are assigned numbers in the order they are added as new vertices are incorporated. This ensures that edge numbers remain stable as the graph grows. The total order is determined primarily by the larger-indexed vertex and secondarily by the smaller-indexed one. △

Using the same four-tuple example (t_0, t_1, t_2, t_3), the clique-wise numbering is:

0 - (t_0, t_1), 1 - (t_0, t_2), 2 - (t_1, t_2), 3 - (t_0, t_3), 4 - (t_1, t_3), 5 - (t_2, t_3)

Figure 5.2 below compares the edge numbering in a five-vertex graph using the two strategies: lexicographical (left) and clique-wise (right).

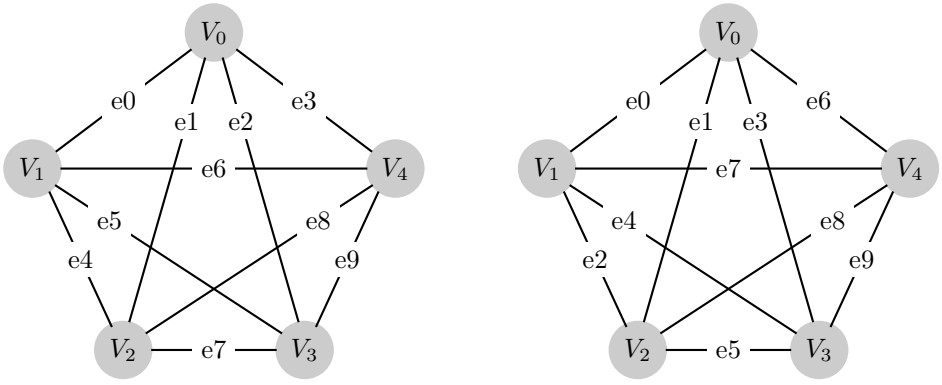


Figure 5.2: Comparison of edge numbering: Lexicographical (left) vs. Clique-wise (right)

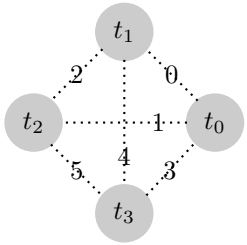
The numbering order significantly influences how agree sets are assigned to edges. As the graph grows, lexicographical numbering can cause previously assigned edge indices to shift. In contrast, clique-wise numbering preserves edge indices, ensuring that earlier edges retain their sequence numbers regardless of graph

expansion. This stability is critical for heuristic methods that rely on consistent edge references across multiple iterations.

Moreover, clique-wise ordering tends to group smaller edge indices within smaller cliques. When agree sets are prioritized by edge number, this has a desirable side effect: edges associated with larger forced sets (and thus more difficult to assign) tend to be processed earlier. This improves efficiency and reduces the likelihood of backtracking during assignment.

To clarify the upcoming steps, we provide a concrete example that will be used throughout the next sections to illustrate key concepts and guide the construction process.

Example 4. Let $R = ABCD$, and$MAX_\Sigma(R) = \{AB, BC, CD, AD, E\}$



$$m = |MAX_\Sigma(R)| = 5$$

$$n = \left\lceil \frac{1+\sqrt{1+8\times m}}{2} \right\rceil = 4$$

Figure 5.3: Graph with $n = 4$ vertices and edges numbered in clique-wise order

3. **Assign the current agree set to the next available edge.**

Begin by assigning the initial agree set, AB, to the first available edge, denoted as e_0. This initial assignment is guaranteed to succeed, as the inclusion of a single agree set cannot result in the formation of a Near Cycle or a Forced Set.

Once AB has been successfully assigned to e_0, proceed by assigning the next agree set, BC, to the subsequent available edge, e_1. Refer to Figure 5.4 for a visual representation of this progression.

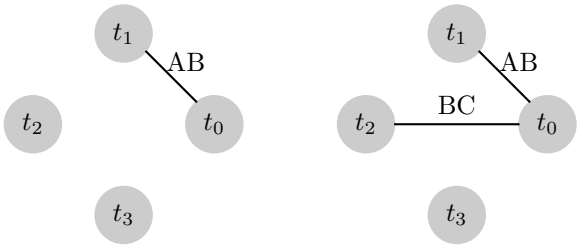


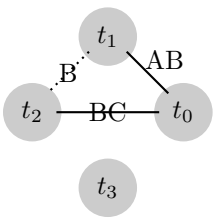
Figure 5.4: Assign AB to edge e_0 and BC to edge e_1

4. **Check for Near Cycle and mark Forced Set**

- Assign the agree set AB to edge e_0 (t_0, t_1) - *succeeds.*

  Since this is the first assignment, no Near Cycle is detected at this stage.
- Assign the agree set BC to edge e_1 (t_0, t_2) - *succeeds.*

  During the Near Cycle check following this assignment, a **Forced Set** is generated. Specifically, attribute B is inferred as forced on edge e_2 (t_1, t_2). No additional Near Cycles are detected as a result of this assignment.



  - A Forced Attribute B is generated on edge e_2.
  - The closure of B is simply B, resulting in Forced Set B on e_2.
  - No Near Cycle is detected after B is forced on e_2.
  - From this point onward, only supersets of B can be validly assigned to e_2.

Figure 5.5: Marking the Forced Set B on edge e_2 due to Near Cycle detection

- Attempt to assign CD to edge e_2 (t_0, t_2) - *skipped*.

  Skipped because CD does not contain the **Forced Set** B on e_2.

- Assign CD to edge e_3 (t_0, t_3) - *succeeds*.

  After assigning agree set CD to edge e_3, a Near Cycle check is triggered. As a result, a **Forced Set** is identified: attribute C is forced on edge e_5 (t_2, t_3). No further Near Cycles are detected in this step.

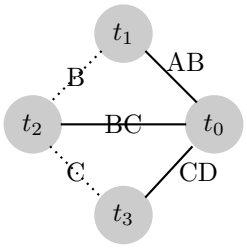


Figure 5.6: Forced Set C is marked on edge e_5 (t_2, t_3) after Near Cycle detection

- Attempt to assign AD to edge e_2 (t_0, t_2) - *skipped*.

  Skipped for the same reason: AD does not contain the **Forced Set** B on e_2.

- Assign AD to edge e_4 (t_1, t_3) - *fails*

  Attempting to assign agree set AD to edge e_4 results in failure due to the emergence of **Near Cycles** that conflict with previously assigned agree sets on other edges.

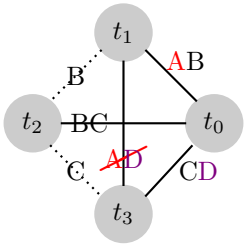


Figure 5.7: Assignment of AD fails due to the emergence of Near Cycles

When attempting to assign AD to edge e_4 (t_1, t_3), two conflicting Near Cycles are identified:

(a) Attribute A appears on e_0 (t_0, t_1) and e_4 (t_1, t_3), but is missing from e_3 (t_0, t_3).

(b) Attribute D appears on e_3 (t_0, t_3) and e_4 (t_1, t_3), but is absent from e_0 (t_0, t_1).

5. **Backtrack if no edge is applicable and unassigned agree sets remain.**

   If all currently available edges have been evaluated and none can accommodate the next agree set, initiate **backtracking**. This step attempts to reverse previous assignments to explore alternative valid configurations.

6. **If backtracking fails and the graph has not yet reached the linear size $m+1$, add a new vertex.**

   Should backtracking be unsuccessful (i.e., no previous configuration allows for progress), and the number of vertices in the graph is still less than the linear size threshold of $m+1$, a new vertex is introduced. This expansion provides additional flexibility to accommodate remaining agree sets.

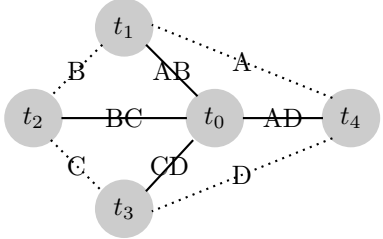


Figure 5.8: The graph grows by adding a new vertex (t_4)

7. **Iterate through the remaining agree sets until all are successfully assigned.**

   With the extended graph structure, continue the assignment process. Attempt to place each remaining agree set on available edges. This iterative process continues until every agree set has been assigned successfully.

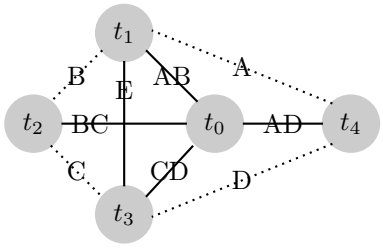


Figure 5.9: Final graph configuration for Example 4

5.2 Alternative paradigm

In the algorithm described above, the standard approach involves iterating through a predefined list of agree sets and identifying suitable edges for their assignment. This agree-set-first strategy prioritizes the sequential processing of agree sets while dynamically evaluating edge eligibility based on structural constraints such as Near Cycles and Forced Sets.

An alternative strategy is to reverse this process by iterating over the list of edges instead. For each edge, we select and assign an appropriate agree set from the pool of unassigned candidates. This edge-first approach leverages the clique-wise ordering of edges, where adjacent or nearby edges are likely to belong to the same clique. As a result, the agree sets assigned to these edges often share common attributes.

This structural alignment can significantly streamline the assignment process. The underlying assumption is that it becomes easier - and often faster - to identify supersets of previously generated Forced Sets, thereby improving the chances of successful assignments. By exploiting the topological and attribute-based relationships between edges and agree sets, the edge-first strategy can effectively reduce the size of the search space during assignment.

Empirical results from our experiments using real-world datasets indicate that this alternative approach can, in some cases, outperform the agree-set-first method in terms of computational efficiency. However, this is not universally the case. There are scenarios where the agree-set-first strategy remains more effective.

5.3 Optimization efforts

We explore several optimization strategies to improve both the efficiency and the quality of the assignment process. These methods are designed either to speed up the algorithm's execution or to improve the likelihood of achieving better results.

- **Sort Agree Sets Prior to Assignment**

  Before the assignment process begins, we can sort the list of agree sets based on attribute similarity - that is, grouping together agree sets that share a larger number of common attributes. When this sorted list is processed sequentially, agree sets with similar attribute compositions are more likely to be assigned to adjacent or closely connected edges in the graph.

  This ordering has two major advantages:

  – It increases the likelihood of attribute continuity across neighboring edges, which often corresponds to the structural tendencies of clique-based graphs.

– It facilitates earlier detection of Near Cycles and the generation of Forced Sets, as structurally related agree sets are introduced earlier in the process. This can help the algorithm make more informed decisions sooner, potentially avoiding deeper backtracking.

Overall, sorting agree sets based on attribute overlap can serve as a lightweight yet effective heuristic for improving both the efficiency and robustness of the assignment phase.

- **Near Cycle Checking Strategies**

  During the agree set assignment process, a critical step is the detection and handling of *Near Cycles*. We explore two main strategies for performing near cycle checks: *Path Seeking* and *Clique Checking*. Each strategy has distinct characteristics in terms of granularity and efficiency.

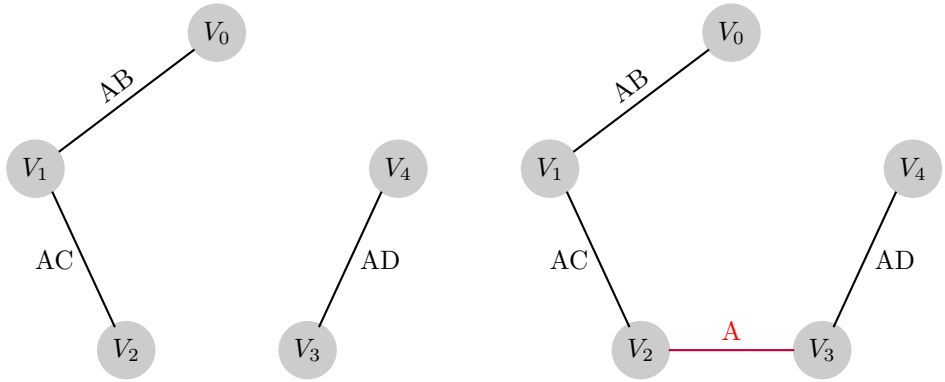
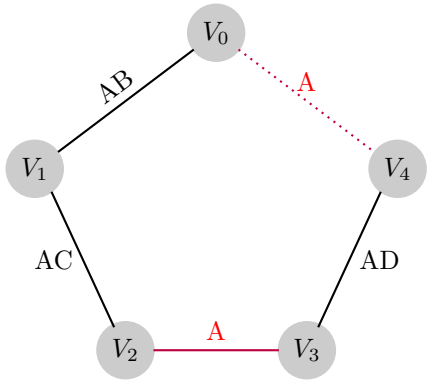


Figure 5.10: Example of Near Cycle Checking after an Assignment

– **Path Seeking**

  In this method, the algorithm searches for existing paths where all edges share a common attribute. If such a path exists and the attribute is missing on the edge directly connecting the two vertices, a near cycle is detected. The missing attribute is then *forced* onto that edge to maintain consistency.

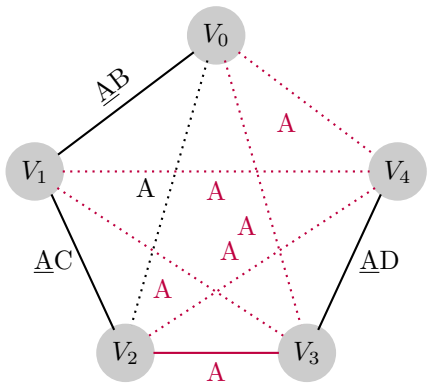


In this example, attribute A has a path from V_0 to V_4, while attribute B does not. Consequently, a near cycle is detected for A, and the attribute A is forced onto the connecting edge between V_0 and V_4.

Figure 5.11: Path Seeking

– **Clique Check**

  This method takes a more global view. Before assigning a new agree set to an edge, the algorithm checks whether the attribute being assigned exists in multiple disjoint cliques. If the current edge connects two such cliques, it effectively merges them into a larger clique. As a consequence, the attribute must then be present on all edges of this new merged clique - forcing the attribute where it is missing.



Before assigning the agree set containing A to the edge between V_2 and V_3, there existed two disjoint cliques labeled with attribute A. When the edge is added, it connects these two cliques into a single larger one, and consequently, A is forced on all edges within the new combined clique.

Figure 5.12: Clique Check

Comparison.
The two strategies emphasize different strengths and limitations:

- **Path Seeking** examines one edge at a time and is well-suited for localized checking, but may require repeated traversals, especially in dense graphs.
- **Clique Check**, on the other hand, operates at a higher level by identifying entire cliques and applying forced assignments in bulk. This approach significantly reduces redundant checks and leads to more efficient runtime performance.

5.4 Complexity Analysis and Heuristic approaches

5.4.1 Theoretical Complexity and Algorithm Analysis

In theory, if a sublinear-size Armstrong table exists for a given relational schema, a corresponding graph can always be constructed. However, in practice, the computational cost of this construction becomes rapidly intractable. The algorithm exhibits super-exponential complexity, specifically $\mathcal{O}(m^{2m})$, where $m = |MAX_\Sigma(R)|$ represents the number of agree sets.

Figure 5.13 presents the high-level pseudocode for the graph construction algorithm:

```
1: n_0 ← ⌈(1+√(1+8m))/2⌉
2: for n = n_0 to m do
3:     Attempt to assign m agree sets to the n(n−1)/2 edges
4:     if successful then
5:         Return graph with n vertices and agree sets assigned
6:     end if
7: end for
8: Return False (no valid assignment found)
```

Figure 5.13: Algorithm in Pseudocode

For a given number of vertices n, let $f(n)$ denote the number of attempts to assign m agree sets to the $\frac{n(n-1)}{2}$ edges of a graph. At the initial value $n = n_0$, the number of permutations is approximately:

$$f(n_0) \approx m!$$

As n increases, the number of possible assignments grows rapidly. For general n, we estimate:

$$f(n) = \frac{\left(\frac{n(n-1)}{2}\right)!}{\left(\frac{n(n-1)}{2} - m\right)!} \approx \left(\frac{n(n-1)}{2}\right)^m$$

In the worst-case scenario, the algorithm continues until $n = m$, in which case the total number of attempts becomes:

$$f(n = m) = \left(\frac{m(m-1)}{2}\right)^m \approx m^{2m}$$

Thus, the total running time across all iterations is:

$$\mathcal{F}(m) = m! + \sum_{n=n_0+1}^{m-1} \left(\frac{n(n-1)}{2}\right)^m + m^{2m}$$

Since the final term dominates the expression, we conclude that the algorithm's overall time complexity is:

$$\mathcal{F}(m) = \mathcal{O}(m^{2m})$$

This exponential behavior poses a significant computational challenge. Although the algorithm is guaranteed to find a valid Armstrong graph if one exists, the runtime can be impractical even for moderately large values of m. This reality motivates the exploration of heuristic and optimization techniques (as discussed in previous sections) to make the problem more tractable in practice.

5.4.2 Heuristic approaches

In an effort to achieve better results within a reasonable computation time, we experimented with a range of heuristic strategies, which can be broadly classified into two categories: **without backtracking** and **with backtracking**.

- **Without backtracking**

  The core algorithm systematically explores all possible assignments of agree sets to the edges of a candidate minimal graph through a backtracking procedure. This exhaustive search ensures correctness but is computationally expensive, with the backtracking step contributing to the majority of the runtime complexity. In contrast, alternative approaches that omit backtracking - illustrated by the absence of the purple-highlighted loop in Figure 5.1 (Algorithm Overview) - circumvent this cost. These approaches instead rely on initial heuristics or deterministic assignments, which can greatly reduce computational overhead, especially when the resulting Armstrong table remains sublinear in size. Although less thorough, such strategies can be effective in practice when performance constraints outweigh the need for completeness.

  – **Randomization**

    We observed that the order in which agree sets are processed can significantly influence the algorithm's output, particularly in the absence of backtracking. Since backtracking constitutes a major portion of the overall computational cost, eliminating it can substantially speed up execution - but may also lead to suboptimal results due to the fixed order of agree set assignments. To address this, we introduce randomization: by executing the algorithm multiple times with different random permutations of the agree sets, we increase the likelihood of discovering a more compact Armstrong table. This probabilistic strategy often yields smaller tables more efficiently, striking a balance between computational cost and solution quality.

- **With backtracking**

  – **Branch and Bound**

    This approach begins by constructing a reasonably sized Armstrong table *without* using backtracking, enabling faster initial computation. The goal is then to iteratively reduce the table size by pruning the graph, using a combination of vertex removal and selective backtracking.

    Once the graph has been built with all agree sets assigned (as illustrated in Figure 5.14, left), we attempt to reduce its size by removing the vertex with the highest index (e.g., the last added vertex) and reassigning agree sets that depended on it. This reassignment is performed using backtracking in the reduced graph.

In the example shown in Figure 5.14 (right), vertex V_5 is removed, which leaves agree set $Ag6$ unassigned, as it previously depended on an edge involving V_5. At this point, it becomes clear that the current configuration of agree sets $Ag1$ through $Ag5$ cannot support $Ag6$ in a graph with only five vertices. If they could, then V_5 would not have been necessary in the original construction. To resolve this, we backtrack a previous assignment - in this case, reassigning $Ag5$ from (V_0, V_4) to (V_3, V_4) - which in turn makes it possible to assign $Ag6$ to edge (V_2, V_4).

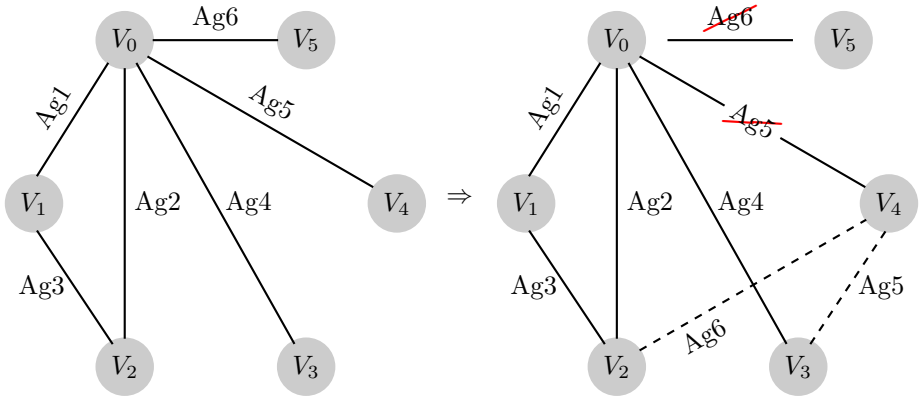


Figure 5.14: Branch and Bound - initial shrink step

This process continues iteratively: we keep attempting to remove additional vertices and reassign any agree sets that become unassigned. The algorithm stops when no further vertex can be removed. As shown in Figure 5.15, if the reassignment of $Ag5$ and $Ag6$ succeeds after removing vertex V_4, the algorithm returns a four-vertex graph. Otherwise, it reverts to the last successful configuration that satisfied all agree sets (see Figure 5.15, left).

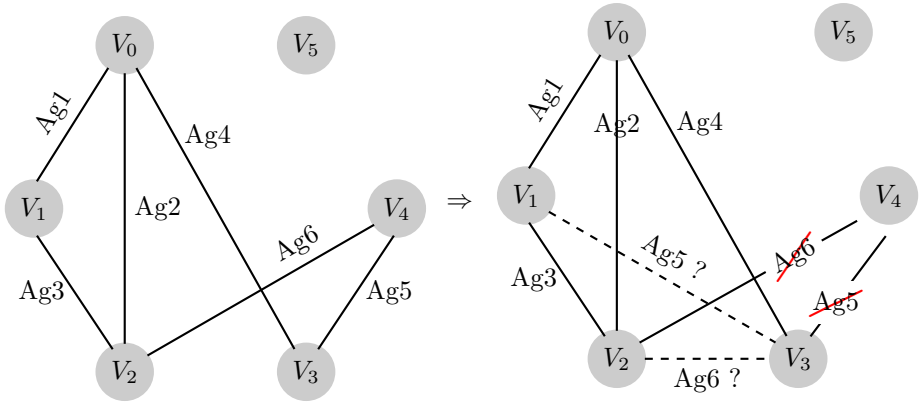


Figure 5.15: Branch and Bound - further reduction and potential failure case

To maintain efficiency, the pruning procedure is bounded by a timeout or maximum number of iterations. If the process takes too long - due to extensive backtracking or repeated reassignment failures - it is aborted, and the last valid, successfully reduced graph is returned as the final output. This allows the algorithm to strike a practical balance between solution quality and computational cost.

– **Early Abortion**

The basic algorithm begins its search from the smallest possible graph size that could represent an Armstrong table. However, in some cases, no such table exists at this minimal size. Rather than exhaustively exploring all possible agree set assignments and engaging in time-consuming backtracking, we implement an early termination strategy.

Specifically, we define a threshold for the number of failed assignment attempts at each size. If this threshold is exceeded - indicating that the current graph size is likely too small to support a valid Armstrong table -

we abort further exploration at that size and increment the graph size before retrying.

While this heuristic may cause the algorithm to skip over a potentially minimal solution, it significantly improves overall efficiency by prioritizing feasibility over optimality. In practice, this trade-off helps the algorithm quickly reach a valid Armstrong table.

– **Staged Backtracking**

Like the previous heuristic strategies, staged backtracking is designed to balance completeness with computational efficiency. The process begins with a graph consisting of three vertices - the minimum number needed to form a cycle - which serves as the initial stage. From this starting point, the graph is progressively expanded in stages, adding one vertex at each step.

At every stage, the algorithm attempts to assign as many agree sets as possible to the edges of the current graph. To enhance flexibility while controlling complexity, backtracking is allowed - but only to a limited depth of one level. This means the algorithm can undo recent agree set assignments made at the current stage, but it does not revisit decisions from earlier stages. This shallow backtracking strikes a compromise between greedy and exhaustive approaches: it enables local corrections without incurring the high cost of full recursive backtracking.

By breaking the construction into stages and incrementally adding vertices, the algorithm reduces the search space at each step, making the process more manageable for larger instances. Furthermore, this staged approach naturally prioritizes simpler subgraphs, which often accommodate many agree sets early on, leading to more efficient convergence.

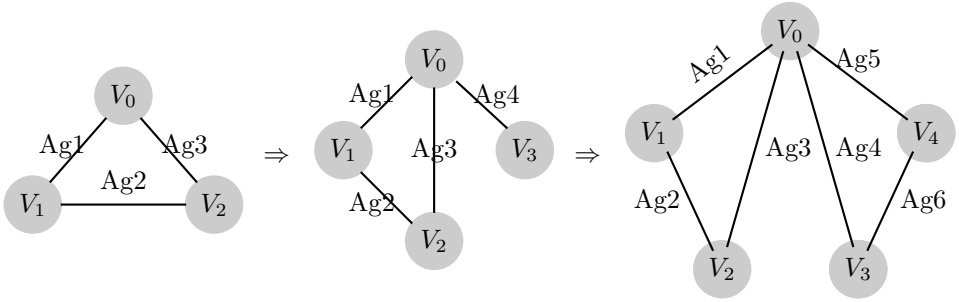


Figure 5.16: Staged Backtracking: expanding the graph step-by-step

5.5 Experiments

We conducted a comprehensive evaluation of our proposed approaches using real-world datasets[2]. The experimental workflow was structured into three distinct phases:

- **Phase 1: Single Dataset Analysis.** We began by selecting a representative real-world dataset to analyze the impact of varying the number of agree sets. Specifically, we compared results using both a partial subset of agree sets and the complete set, denoted as $MAX_{\Sigma}(R)$. This phase allowed us to observe how the density and distribution of agree sets influence the resulting Armstrong table sizes.
- **Phase 2: Cross-Dataset Evaluation.** In the second phase, we expanded our evaluation to include multiple datasets of varying sizes and characteristics, ranging from small tables with a few attributes to large datasets with high dimensionality. This scaling analysis helped assess the generalizability and performance of each approach across different data profiles.
- **Phase 3: Evaluation on Randomly Generated Tables.** To further test robustness and performance, we generated a set of synthetic (random) tables with controlled sizes. For each such table, multiple experiments

[2]Available at https://hpi.de/en/naumann/projects/repeatability/data-profiling/fds.html

were conducted to observe consistency in the resulting Armstrong table sizes. In particular, for experiments involving randomization heuristics, each configuration was executed ten times, and the smallest resulting Armstrong table size was recorded to represent the best-case outcome.

The results of these experiments - including the computed Armstrong table sizes under different heuristics - are presented for comparison and analysis.

5.5.1 Experiment 1: Single Dataset Analysis

We recognize that the number of agree sets plays a significant role in determining the size of the resulting Armstrong table. To better understand this relationship and to evaluate the relative performance of our proposed heuristic approaches, we selected a medium-sized real-world dataset, `bridges`, for detailed experimentation. This dataset yields a total of 63 mined agree sets.

To systematically assess how the number of input agree sets affects the final Armstrong table size, we randomly sampled subsets of varying sizes from the full set of 63 agree sets. For each subset size, we applied four different algorithmic strategies: *Randomization*, *Branch and Bound*, *Early Abortion*, and *Staged Backtracking*. The resulting Armstrong table sizes were recorded for each approach.

The table below presents the results. The first column indicates the number of agree sets used as input, while the subsequent columns show the corresponding sizes of the generated Armstrong tables under each method:

# Agree Sets	Randomization		Branch and Bound		Early Abortion		Staged BT	
	Size	Time	Size	Time	Size	Time	Size	Time
8	8	0.106s	7	0.475s	7	1.856s	8	2.321s
10	10	0.272s	9	0.334s	9	2.162s	10	10.079s
14	11	1.133s	11	1.171s	11	2.199s	14	40.644s
21	17	7.864s	17	10.703s	17	21.348s	21	3m15s
32	26	57.189s	25	23.176s	22	38.165s	30	22m44s
63	38	20m40s	41	2m04s	37	8m03s	41	77m53s

Table 5.1: Effect of Agree Set Size on Armstrong Table Size and Running Time (Dataset: `bridges`)

As shown in Table 5.1, the size of Armstrong tables increases across all methods as the number of agree sets grows. However, the final table size varies depending on the strategy employed. Notably, the *Early Abortion* strategy often produces smaller tables than the other methods, particularly for moderate agree set counts. This strategy terminates early after 10 consecutive failed assignment attempts at each candidate table size.

In contrast, the *Branch and Bound* method applies more conservative pruning but is constrained by a time and iteration limit: it aborts the search if no improvement is found within 1 minute or after 10 iterations. While generally effective, these constraints may limit its ability to find compact solutions under tighter conditions.

Staged Backtracking, though more exhaustive, tends to yield larger tables. Finally, the *Randomization* strategy executes 10 independent runs, each using a different random ordering of the agree sets. The smallest resulting table size among these runs is recorded as the output. While this approach introduces variability, it occasionally finds competitive solutions with minimal overhead.

5.5.2 Experiment 2: Cross-Dataset Evaluation

In this experiment, we evaluated the performance of our proposed approaches across a diverse range of real-world datasets. These datasets vary significantly in structure, with attribute counts ranging from a few to over a dozen, and record counts spanning from hundreds to tens of thousands. The goal was to examine how each heuristic scales under different data conditions and to compare the resulting Armstrong table sizes.

For each dataset, we mined its full set of agree sets and applied four strategies: *Randomization*, *Branch and Bound*, *Early Abortion*, and *Staged Backtracking*. The final sizes of the generated Armstrong tables are summarized in Table 5.2, sorted by the number of mined agree sets.

Dataset	Size (Cols×Rows)	# Agree Sets	Random	B & B	E. Abortion	Staged BT[3]
balance_scale	5×625	8	8	7	7	8
iris	5×150	10	8	8	8	8
chess	7×28056	12	12	11	11	13
nursery	9×12960	16	16	15	15	17
breast_cancer_wisconsin	11×699	43	33	32	31	32
adult	15×32561	55	45	49	47	49
abalone	9×4177	64	27	32	32	32
letter	17×20000	81	60	65	65	65
echocardiogram	13×132	84	30	31	31	31

Table 5.2: Armstrong Table Sizes Across Multiple Datasets

As shown in Table 5.2, the size of the resulting Armstrong table generally increases with the number of agree sets. While all methods perform similarly on smaller datasets, performance differences become more pronounced as dataset size increases. *Early Abortion* and *Branch and Bound* frequently generate smaller tables than other approaches, particularly on medium to large datasets.

Staged Backtracking, despite its exhaustive nature, delivers consistently modest results which is contrary to expectations and consistent with its Phase 1 performance. In some cases (highlighted in red), it even fails to produce sublinear-sized tables.

Interestingly, the *Randomization* strategy performs quite well across a broad range of datasets. By executing 10 runs with randomized agree set orders and selecting the smallest resulting table, it often achieves results that are competitive with or even better than more structured methods. Its strong performance, particularly on large and high-dimensional datasets like `abalone`, `letter`, and `echocardiogram`, demonstrates the potential benefit of leveraging randomness to escape local optima with minimal computational overhead.

5.5.3 Experiment 3: Evaluation on Randomly Generated Tables

In this experiment, we evaluated the performance of our approaches using synthetically generated tables with predefined sizes. Unlike real-world datasets, these synthetic tables allow us to tightly control the size of the minimal Armstrong table. This setup is particularly valuable because, in practical scenarios, the true minimal size is typically unknown. By establishing a known upper bound, we can systematically observe algorithmic behavior under controlled conditions.

For each configuration (i.e., fixed number of columns and rows), we randomly generated a binary table, where the number of rows corresponds to the theoretical upper bound, or in some cases, the minimal size of the Armstrong table for that instance. This upper bound is especially tight in configurations with a large number of columns and relatively few rows.

An example of a 5×5 random table is shown below:

Col 1	Col 2	Col 3	Col 4	Col 5
0	0	1	0	1
1	1	1	1	1
0	0	1	1	0
0	1	1	0	0
0	1	1	1	1

[3]Red values indicate that no sublinear-size Armstrong table was achieved using Staged Backtracking.

From each generated table, we mined the set of agree sets and then applied all four heuristic approaches: *Randomization*, *Branch and Bound*, *Early Abortion*, and *Staged Backtracking*. The goal was to evaluate how close each method could get to the theoretical minimum table size derived from the original random table.

Size (Cols×Rows)	#Agree Sets	Randomization	Branch and Bound	Early Abortion	Staged BT
10×10	26	15	18	18	21
10×15	49	27	28	26	28
10×20	48	31	30	25	30
15×10	37	20	22	22	22
15×15	86	35	43	39	43
15×20	161	54	58	51	58
20×10	42	21	26	23	26
20×15	101	43	48	48	50

Table 5.3: Armstrong Table Sizes for Randomly Generated Tables

As shown in Table 5.3, the number of agree sets increases with the size of the original table. Across different configurations, the *Early Abortion* approach often achieved the smallest Armstrong tables, particularly when the number of agree sets was moderate to high. In several configurations (e.g., 10×20 and 15×20), it produces the smallest Armstrong tables among all methods. *Randomization* and *Branch and Bound* also performed competitively but tended to produce slightly larger results. As in previous experiments, *Staged Backtracking* generally lagged behind the other methods.

5.5.4 Summary

The experimental results indicate that *Branch and Bound* is among the most consistently effective approaches, frequently yielding smaller Armstrong tables across a variety of datasets. However, *Early Abortion* often outperforms it in terms of table size, especially as the number of agree sets increases. This suggests that *Early Abortion* can achieve higher-quality results when allowed sufficient runtime. Nevertheless, its execution time is generally longer than that of *Branch and Bound*, which may impact its practicality in time-sensitive scenarios. The *Randomization* approach demonstrates mixed performance: in some cases, it produces optimal or near-optimal results, but in others, it fails to find competitive solutions. On the other hand, *Staged Backtracking* generally underperforms. It often produces larger Armstrong tables and, in some cases, fails to produce sublinear solutions altogether.

In terms of computational efficiency, methods involving backtracking such as *Branch and Bound* and *Staged Backtracking* tend to incur higher runtime costs, especially as the number of agree sets increases. *Randomization* is consistently the fastest method for small to moderately sized agree set configurations. However, its runtime increases steeply with larger agree sets, eventually surpassing other methods due to repeated randomized evaluations. In contrast, *Staged Backtracking* is typically the slowest approach. While *Branch and Bound* and *Early Abortion* demonstrate more moderate scaling behavior, they too experience runtime growth as the complexity of the input increases. Notably, datasets with a larger number of attributes (i.e., columns) significantly amplify runtime across all methods. This overhead arises from the increased cost of Near Cycle detection and the more intricate interactions between agree sets in higher-dimensional spaces.

It is important to note that our construction methods do not guarantee optimality in terms of table size. Achieving sublinear-sized Armstrong tables within practical runtime constraints necessitates trade-offs. For example, the last entry in Table 5.3 shows that 101 agree sets were derived from a base of only 15 tuples, leading to a linear-sized Armstrong table with up to 102 rows. This highlights a general trend: even the most compact sublinear Armstrong tables found to be 2-3 times larger than the original datasets, particularly in high-density scenarios.

Chapter 6

Implementation for Informative Sublinear Armstrong Table

The Informative Armstrong table is derived from an existing Armstrong table, which adheres to the same constraints but is presented in a more compact form. It builds upon the traditional Armstrong table framework by offering a more intuitive representation of dependencies [31]. By semantically sampling real-world databases, the Informative Armstrong table provides domain experts with a clearer and more comprehensible understanding of the dependencies. Its reduced size allows for a more focused view, making it easier to identify missing constraints and preventing the inadvertent overlooking of undesired ones.

While the reduced size of the data is advantageous, a substantial volume of information can still be overwhelming, complicating the process of constraint identification. In the case of non-informative Armstrong tables, the maximum size is $m + 1$, where m represents the number of agree sets, such a table can always be constructed. In contrast, informative Armstrong tables generally have a linear size of $2m$ because they rely on guaranteed tuple pairs derived from the existing table. This structure can be more efficient in capturing dependencies but might not always accommodate more compact form - such as configurations involving three tuples satisfying two agree sets - which may not be feasible in all cases.

When the number of agree sets in an existing table is large, an informative Armstrong table with a size of $2m$ may become impractical due to its size. In such scenarios, reducing the size to a sublinear scale could make the table much more manageable. Discovering an informative Armstrong table of sublinear size could significantly increase its practical utility, offering a more efficient and accessible way for domain experts to explore and understand dependencies.

6.1 Algorithm principle

The algorithm processes an existing real world table as input and generates an informative Armstrong table as its output. The primary goal of this algorithm is to extract and represent dependencies in a more efficient and comprehensible manner, making it easier for domain experts to understand the underlying relationships within the data. The general procedural steps involved in this process are outlined below:

1. The process begins with mining agree sets from an existing table, which forms the foundational step in constructing informative Armstrong tables. Agree sets capture combinations of attributes where pairs of tuples agree in their values, and these sets play a crucial role in identifying both keys and functional dependencies (FDs). A comprehensive visualization of how agree sets are mined specifically for the discovery of keys is provided in Figure 6.1. Similarly, the procedure for mining agree sets in the context of identifying functional dependencies is illustrated in Figure 6.2. These figures serve to clarify the nuanced steps involved in the mining process and highlight the structural differences in the approaches used for keys and FDs, respectively.

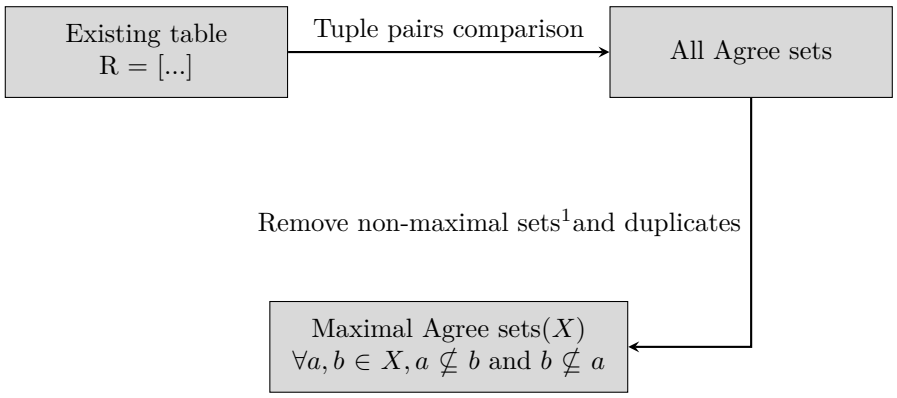


Figure 6.1: Mining Process for Keys

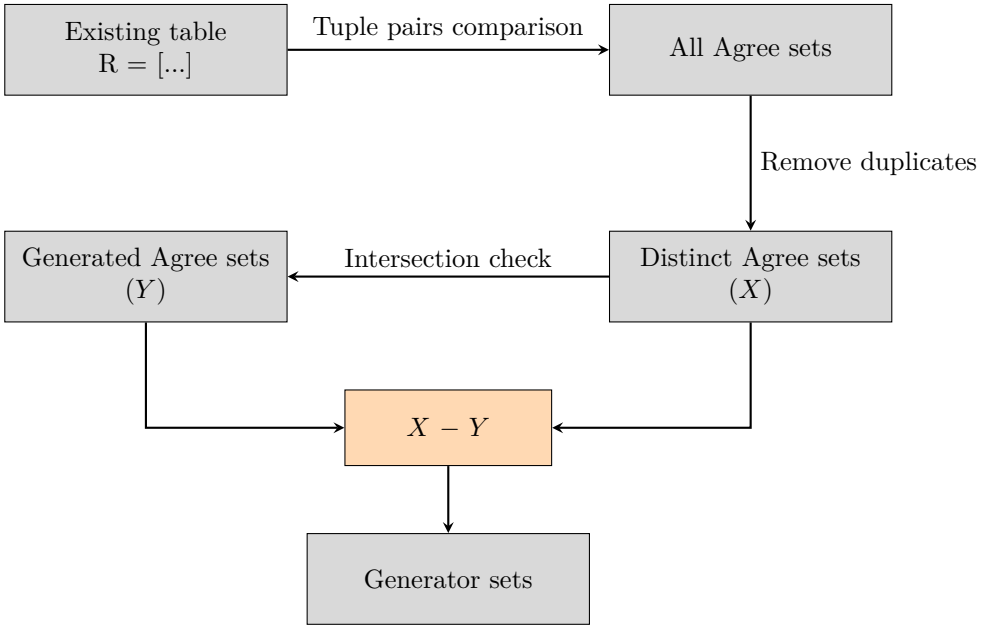


Figure 6.2: Mining Process for FDs

The mining process generates a collection of Generator Sets, which represent the desired agree sets as formally defined in Definition 2.2.9. These Generator Sets are crucial to the construction of the informative Armstrong table, as they encapsulate the essential agree sets from the data.

2. Convert the original table into a graph:

   - Numbering tuples as vertices: Each tuple in the table is treated as a vertex (node) in the graph. These vertices are uniquely numbered to represent the distinct tuples from the table.
   - Drawing edges between vertices (tuples) labelled with agree sets: If two tuples share common attribute values (i.e., they agree on certain attributes), an edge is drawn between the corresponding vertices. Each edge is labeled with the agree set, which represents the set of attributes on which the two tuples agree.
   - Remove non-maximal/generated sets from the graph: Non-maximal sets are subsets of maximal sets. In some cases, these sets can be derived from generators and are thus referred to as generated sets. These non-maximal sets are considered redundant or non-essential to the structure of the graph and, therefore, should be removed.

[1]Non-maximal sets are subsets of other sets

- Handling isolated vertices: If a tuple (vertex) does not share any agreeing attributes with other tuples, it is considered an isolated vertex. These isolated vertices do not have any edges connecting them to other vertices. Therefore they can be removed from the graph, simplifying the structure.

3. Eliminate duplicate agree sets from the graph

   At this stage, the graph contains only the essential agree sets. The next step is to further streamline the graph by identifying and eliminating as many duplicate agree sets as possible. While the goal is to reduce redundancy, it's important to note that the elimination process may not always result in entirely distinct agree sets. In some cases, certain duplicates might remain due to the inherent complexity of the data. Nevertheless, the focus remains on minimizing the graph's size by discarding unnecessary vertices, resulting in a more concise and manageable structure.

4. Convert the simplified graph back into an Armstrong table

   After processing, the final graph will contain only the minimal number of essential vertices, eliminating redundancies that were previously present. This reduced graph can then be translated back into an informative Armstrong table, now compacted into a significantly smaller size. Ideally, this transformation leads to a table that has a sublinear size, while still retaining all critical information. As a result, the data becomes more manageable, enabling more efficient and effective analysis.

6.2 Methods to eliminate duplicates

The size of the output Informative Armstrong table is primarily determined by how effectively duplicate agree sets are eliminated during step 3 of the algorithm. This step is crucial because it directly influences the compactness of the final graph, aiming to reduce the number of vertices while still ensuring that each distinct agree set is represented at least once in the final structure. The challenge is to minimize redundancy without losing critical information. To achieve this goal efficiently, various strategies can be employed, including:

- Type A: These methods are designed to ensure that the final table remains as small as possible. By prioritizing minimality at each step, they help optimize the overall size of the graph.
- Type B: These heuristic methods are designed to provide rapid solutions by prioritizing speed and practicality over guaranteed minimality. While they can offer quick approximations, their primary focus is on delivering results within a reasonable timeframe, often at the expense of achieving the most compact or optimal solution.

Most approaches combine elements from both Type A and Type B methods to ensure computational feasibility. This hybrid strategy allows for a more versatile solution. The challenge lies in striking the right balance between minimality and efficiency, as these two goals often conflict: while minimality focuses on reducing the size of the output, efficiency prioritizes speed and resource usage. Achieving an optimal balance is crucial to ensure that the solution is both effective and practical, especially when dealing with large datasets.

In situations where straightforward decisions or optimal choices are not immediately apparent, heuristics come into play to resolve ties. Heuristics provide a means of making quick, pragmatic decisions. These decision-making strategies are particularly useful when exact calculations would be computationally expensive or time-consuming. By utilizing heuristics, they offer solutions that are "good enough" while maintaining computational efficiency.

These methods can also be classified based on their operational focus: either edge-oriented or vertex-oriented.

- Edge-oriented approaches focus on evaluating and selecting the most suitable edge, particularly among those labeled with the same agree sets. The objective is to make optimal decisions at the edge level.
- Vertex-oriented approaches, by contrast, involve operations centered on vertices - such as selecting specific vertices for inclusion or removing them - to achieve a desired optimization.

Definition 6.2.1 (Key Vertex)**.** A Key Vertex (abbreviated as KV) is defined as follows:

1. It is a vertex that is adjacent to an edge labeled with a unique agree set, or

2. It is a vertex that is connected to all duplicates of edges labeled with the same agree set. △

For example, consider an agree set $Ag1$ present on edges $e0$ (between vertices V_0 and V_1) and $e1$ (between vertices V_0 and V_2), with no other edges involved. In this case, vertex V_0 qualifies as a Key Vertex because it is adjacent to both edges $e0$ and $e1$, which are labeled with the same agree set $Ag1$. Thus, V_0 is connected to all the edges labeled by the same agree set, satisfying the second condition of the definition.

Definition 6.2.2 (Critical Vertex)**.** A Critical Vertex (abbreviated as CV) is defined as a vertex that is either explicitly selected during the Selection Process or becomes a Key Vertex as a result of graph updates. Initially, the set of Critical Vertices consists exclusively of the Initial Key Vertices. As the Selection Process unfolds and the graph is dynamically updated, the list of Critical Vertices is expanded through a series of strategic selections, ensuring that the final set represents a minimal subset of vertices. The ultimate goal of the selection process is to form a smaller graph in which each agree set is represented at least once by one or more edges connecting the selected Critical Vertices. △

Definition 6.2.3 (Validated)**.** An agree set is marked as Validated if it satisfies one of the following conditions:

1. An edge labeled with the agree set has been selected. This selected edge remains in the graph, while all other edges labeled with the same agree set are removed. (Edge-oriented approaches).

2. At least one edge labeled with the agree set is adjacent to both Critical Vertices. Once validated, all edges labeled with this agree set are removed from the graph to prevent interference with subsequent vertex selection or removal (Vertex-oriented approaches). △

Refer to Figure 6.3 for an illustration of the principle behind duplicate elimination (vertex-oriented approach). The process of efficiently removing redundant edges and reducing the graph size is heavily reliant on strategic vertex selection. By carefully choosing which vertices to keep and which vertices to remove, we can significantly streamline the graph.

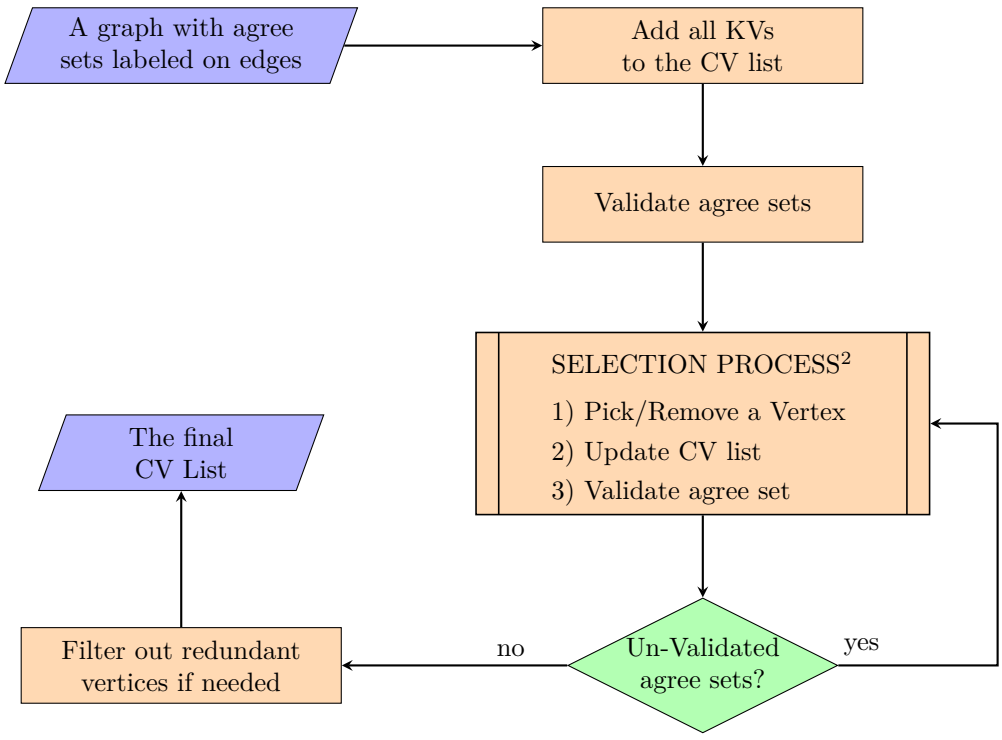


Figure 6.3: Vertex Selection Process

[2]Selection processs for edge-oriented approach: 1) Select a edge; 2) Validate agree set; 3) Update CV list.

Vertices with higher degrees, particularly those that are adjacent to a larger number of distinct edges, play a pivotal role in this process. These vertices are considered more valuable because they are more interconnected and often serve as key connectors in the graph. Selecting edges that are connected to these high-degree, high-value vertices is a strategic approach to achieving a more compact graph, as it ensures that the essential connections are preserved while eliminating redundant edges.

The effectiveness of the duplicate elimination process is directly influenced by the specific vertex selection methods employed. Different methods may prioritize various factors, such as vertex degree, edge connectivity, or the overall structure of the graph, resulting in different outcomes. Some methods might lead to a more compact graph by focusing on the most connected vertices, while others might achieve efficiency through different heuristics or criteria.

A deeper exploration of these vertex selection strategies will offer valuable insights into how each approach impacts the final graph's size.

6.2.1 Edge Number (EN)

The Edge Number serves as a heuristic baseline for edge selection, offering a straightforward yet effective method for determining which edges should be prioritized. In this approach, edges are assigned numbers based on a clique-wise order (refer to Definition 5.1.2). When an agree set is associated with multiple edges, the heuristic dictates that the edge with the smallest assigned number should be selected. This edge is retained, while all other edges labeled with the same agree set are removed from the graph. For further clarification, please refer to Figures 6.4 and 6.5, which provide the pseudocode and an example graph.

Due to the nature of the clique-wise numbering method, edges with smaller numbers tend to cluster within a more compact graph. This method is characteristic of a Type B, edge-oriented approach, where decisions are driven by heuristics rather than exhaustive searches or complex optimization techniques. Type B methods prioritize computational efficiency by employing simple, rule-based strategies to approximate solutions. Instead of exploring every possible edge configuration or seeking a globally optimal solution, the heuristic focuses on locally favorable conditions, such as the edge number, to expedite the graph shrinking process. While this approach may not yield the optimal solution, it establishes a useful baseline for comparison with other approaches.

Input: A graph with agree sets labeled on edges
1: `Add all KVs to the CV List`
2: **for each** un-Validated agree set A_i **do**
3: `Choose the edge labeled` A_i `with the smallest number`
4: `Add adjacent vertices to the CV List`
5: `Mark agree set` A_i `as Validated`
6: **end for**
Output: The final CV List

Figure 6.4: Pseudocode for Edge Number (EN)

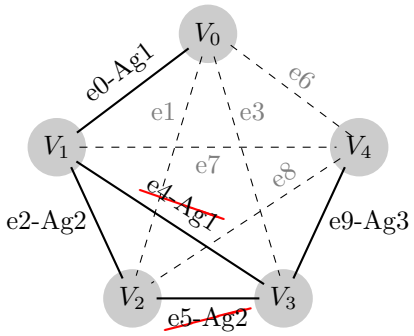


Figure 6.5: Example of Edge Number (EN)

In the example illustrated in Figure 6.5, the agree set $Ag1$ labels both $e0$ and $e4$. Since $e0$ is assigned the smaller number, it is selected over $e4$ according to the heuristic of choosing the edge with the smallest number. This selection ensures that $e0$ is retained while $e4$ is eliminated as a redundant edge for the agree set $Ag1$.

Similarly, for $Ag2$, edge $e2$ and $e5$ are labeled with the same agree set. Since $e2$ has the smaller number, it is chosen over $e5$, and $e5$ is subsequently removed from the graph. This process is applied consistently across all agree sets in the graph.

6.2.2 Progressive Vertex Expansion (PVE)

Progressive Vertex Expansion (PVE) is also a Type B, edge-oriented method. The process begins with an initial set of Key Vertices, which form the seed batch of Critical Vertices.

For each unvalidated agree set, PVE evaluates the connectivity of its edges with the existing Critical Vertices. If a labeled edge is found to be adjacent to one or two vertices currently in the CV list, that edge is selected. Any vertices adjacent to this selected edge are then promoted to Critical Vertices and added to the CV list, provided they are not already included.

By prioritizing edges that are adjacent to existing Critical Vertices, PVE ensures a localized and efficient expansion of the CV set. This strategy minimizes the number of new vertices introduced during the validation process, leveraging the structural influence of already identified CVs. The progressive nature of the method allows it to incrementally grow the CV set in a way that maximizes relevance and connectivity while avoiding unnecessary exploration of unrelated regions in the graph.

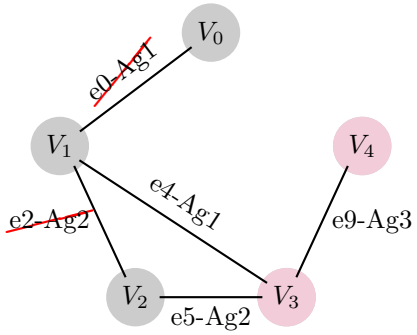


Figure 6.6: Example of Progressive Vertex Expansion(PVE)

In the example illustrated in Figure 6.6, vertices V_3 and V_4 are designated as Critical Vertices (CVs). When evaluating the edges labeled with the same agree set $Ag1$, two candidate edges $e0$ and $e4$ are considered. Although both belong to $Ag1$, edge $e4$ is selected over $e0$ because it is adjacent to one or more of the existing Critical Vertices. This selection aligns with the core principle of Progressive Vertex Expansion (PVE), which prioritizes edges that connect to the current CV list.

A similar decision process is applied for the agree set $Ag2$. Here, edges $e2$ and $e5$ are candidates. Edge $e5$ is chosen over $e2$ because it is adjacent to an existing Critical Vertex, whereas $e2$ is not. This consistent preference for edges connected to the current CV set ensures a more focused and efficient expansion of Critical Vertices, reinforcing the localized growth strategy that defines PVE.

Figure 6.7 presents the pseudocode implementation of the Progressive Vertex Expansion (PVE) algorithm. This pseudocode outlines the step-by-step procedure used to incrementally grow the set of Critical Vertices based on edge adjacency within labeled agree sets. It formalizes the selection criteria described earlier- specifically, the preference for edges that are adjacent to existing Critical Vertices - and details how new vertices are identified and added to the CV list. By encapsulating the logic of PVE in algorithmic form, Figure 6.7 serves as a reference for both conceptual understanding and practical implementation.

```
for each agree set A_i marked as un-Validated do
    for each edge labeled with A_i do
        if two adjacent vertices are CVs then
            Choose this edge
            Break
        else if one of adjacent vertex is a CV then
            Choose this edge
            Break
        end if
    end for
    if no edge has been chosen then
        Choose the first edge
    end if
    Add both adjacent vertices of the chosen edge to the CV list if not already included
    Mark agree set A_i as Validated
end for
```

Figure 6.7: Pseudocode for Progressive Vertex Expansion(PVE)

6.2.3 Edge Rank (ER)

To guide the selection of edges during the expansion process, a ranking mechanism is introduced - referred to as Edge Rank (ER). This ranking system enables more informed decision making by prioritizing edges based on the structural relevance of their adjacent vertices. Edges with higher ranks are preferred, as their inclusion is more likely to contribute meaningful vertices to the set of Critical Vertices (CVs). Conversely, edges with lower ranks are deprioritized or excluded from consideration.

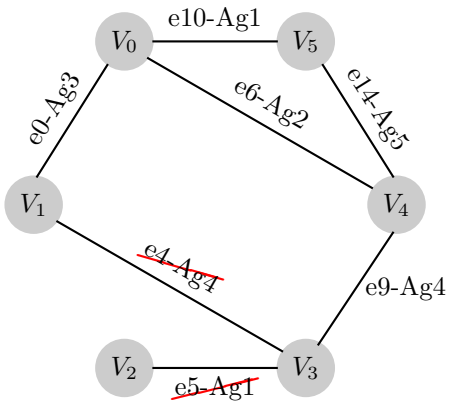
The ranking system is defined through two key components:

- Vertex Rank: For any given vertex, the rank is determined by the number of distinct adjacent agree sets it participates in. This is conceptually similar to the vertex's degree, but with a critical distinction - it excludes multiple occurrences of the same agree set label. Thus, if a vertex is connected to multiple edges labeled with the same agree set, they are only counted once in the rank calculation.
- Edge Rank: The rank of an edge is defined as the maximum vertex rank among its two endpoints. This ensures that edges connected to highly relevant (i.e., highly connected) vertices are prioritized during the selection process.

It is important to note that vertex ranks are dynamic. As agree sets are validated and removed from consideration, they no longer contribute to the rank computation of their associated vertices. This dynamic behavior ensures that the ranking system remains responsive to changes in the graph and accurately reflects the current state of edge and vertex relevance.

The Edge Rank method, by emphasizing connectivity through structured prioritization, is still a Type B, edge-oriented approach. As a heuristic method, it prioritizes speed and practicality - delivering timely and effective approximations of relevance, rather than guaranteeing the most optimal or minimal outcome.

Figure 6.8 illustrates how Edge Rank is used to prioritize which edges should be selected when expanding the set of Critical Vertices. Edges like $e10$ and $e9$, with higher adjacent vertex ranks, are favored over lower-ranked alternatives. The exclusion of $e5$ and $e4$, despite being part of the same agree sets as higher-ranked edges, highlights the filtering effect that Edge Rank introduces.



Rank(e10) = 3
(rank(V_0) = 3, rank(V_5) = 2)
Rank(e5) = 2
(rank(V_3) = 2, rank(V_2) = 1)

Rank(e9) = 3
(rank(V_4) = 3, rank(V_3) = 2)
Rank(e4) = 2
(rank(V_1) = 2, rank(V_3) = 2)

Figure 6.8: Example of Edge Rank (ER)

6.2.4 Weighted Edge Rank (WER)

To more effectively prioritize edges that are structurally significant and closely connected to the current set of Critical Vertices (CVs), the Weighted Edge Rank (WER) method introduces a dynamic weighting factor. This factor amplifies the importance of edges adjacent to Critical Vertices, thereby influencing edge selection in a way that promotes meaningful graph expansion.

Unlike basic Edge Rank (ER), which only considers the static ranks of adjacent vertices, WER combines this information with the proximity to Critical Vertices. Specifically, edges connected to one or more CVs are given a higher priority by adding a weight that reflects the number of adjacent Critical Vertices. This adjustment allows the algorithm to more strongly favor edges that reinforce or extend existing critical regions in the graph.

The weighted rank of an edge is computed using the following principles:

- It builds upon the original Edge Rank, which is defined as the maximum rank among the edge's two endpoints (as illustrated in Figure 6.8).
- A weight boost is applied based on the number of adjacent vertices that are currently marked as Critical Vertices.
- The final weighted rank is recalculated dynamically at each iteration to account for:
  - The continuous addition of new Critical Vertices.
  - The removal of validated agree sets, which in turn affects vertex ranks.

This hybrid approach merges the targeted expansion behavior of Progressive Vertex Expansion (PVE) with the structural prioritization of Edge Rank (ER). By balancing rank-based relevance with CV adjacency, WER ensures that the most contextually important edges are selected. Overall, WER is categorized as a Type B, edge-oriented method too.

Figure 6.9 outlines the process for selecting edges based on a weighted ranking system that emphasizes connectivity to Critical Vertices. For each unvalidated agree set, all associated edges are evaluated. An edge's rank is calculated by combining its base vertex rank with a weighting factor that reflects how many of its adjacent vertices are already Critical Vertices. Edges adjacent to two CVs receive the highest boost, followed by those adjacent to one CV. The edge with the highest computed rank is selected, and its adjacent vertices are added to the Critical Vertex list if not already present. Finally, the agree set is marked as validated.

```
Weight = a large integer constant
for each agree set A_i marked as un-Validated do
    for each edge labeled by A_i do
        if two adjacent vertices are Critical Vertices then
            E_Rank = Weight * 2 + Max(V_Rank)
        else if one ajacent vertex is Critical Vertex then
            E_Rank = Weight + Max(V_Rank)
        else
            E_Rank = Max(V_Rank)
        end if
    end for
    Add adjacent vertices of the highest-ranked edge to the CV list if not included
    Mark agree set A_i as Validated
end for
```

Figure 6.9: Pseudocode for Weighted Edge Rank (WER)

6.2.5 Weighted Vertex Rank (WVR)

While previous methods have focused on edge-based strategies for expanding the set of Critical Vertices (CVs), an alternative approach is to prioritize vertex selection directly, using a weighted ranking system. This method ensures comprehensive coverage by aiming to include all agree sets through the strategic selection of high-priority vertices.

The Weighted Vertex Rank (WVR) of a vertex is determined by combining two components:

- Base Vertex Rank: Defined as the number of distinct agree sets connected to the vertex
- Critical Adjacency Weight: A dynamic value computed as the product of the number of neighboring Critical Vertices and a predefined constant weight.

$$WVR(v) = BaseRank(v) + (NumCVNeighbors(v) \times Weight)$$

This formula boosts the priority of vertices that are structurally well-connected to already identified Critical Vertices, effectively guiding the selection process toward regions of the graph that are more likely to contribute to efficient and complete agree set validation.

WVR leverages structural information and dynamically updates ranks as Critical Vertices and validated agree sets evolve over time. By integrating local vertex importance with proximity to Critical Vertices, it enables targeted and efficient graph expansion - minimizing redundancy while ensuring comprehensive coverage of relevant regions. As such, WVR aligns with Type B methods, prioritizing practical efficiency and timely approximations over guaranteed optimality. In contrast to earlier approaches, WVR is explicitly vertex-oriented in its design.

```
Add all vertices not in the CV list to the candidate pool and sort them by weighted rank
while un-Validated agree sets exist do
    Choose a vertex with the highest rank and add it to the CV list
    Remove the chosen vertex from the candidate pool
    if an edge is adjacent to two Critical Vertices then
        Mark labeled agree sets as Validated
    end if
    Remove vertices adjacent to Validated agree from the cadidate pool
    Update ranks of vertices in the cadidate pool
end while
```

Figure 6.10: Pseudocode for Weighted Vertex Rank (WVR)

This algorithm selects vertices based on their weighted rank to ensure all agree sets are validated. Initially, all non-Critical Vertices are added to a candidate pool and sorted by rank. The highest-ranked vertex is iteratively selected and added to the Critical Vertex (CV) list. If any edge becomes adjacent to two CVs as a result, its labeled agree set is marked as validated. Vertices connected to validated agree sets are removed from the candidate pool, and ranks are updated accordingly. This process continues until all agree sets are validated.

6.2.6 Domination

The Domination approach is a vertex-centered method that, unlike previous strategies which focus on selecting vertices or edges for inclusion, works by eliminating vertices that are dominated by others. After this pruning process, the remaining vertices - once refined - form the final set of Critical Vertices (CVs). This method ensures that all agree sets are preserved while minimizing redundancy, thereby producing a concise and informative Armstrong table. Notably, this is the only method classified as Type A, characterized by its strict adherence to minimality in the resulting CV set. It is also explicitly vertex-oriented in its operation.

The core idea behind this method is to remove only those vertices that can be confidently replaced by others without leading to worse outcomes. When no vertices meet the criteria for dominance, a heuristic selection mechanism is activated to guide further refinement - strategically removing vertices while preserving the overall coverage of agree sets.

To formalize the notion of dominance, two key definitions are introduced:

Definition 6.2.4 (Set of Confirmed Agree Sets)**.** The set of confirmed agree sets for a vertex V, denote as $CAS(V)$, is the set of agree set labels (i.e., edge labels) connecting V to one or more Critical Vertices. △

Definition 6.2.5 (Set of Potential Agree Sets)**.** The set of potential agree sets for a vertex V, denote as $PAS(V)$, consists of all agree set labels associated with edges adjacent to V, regardless of the CV status of neighboring vertices. △

In the context of decision theory, one option is said to dominate another if it consistently performs better and never worse. Translating this to the graph setting: a vertex V_1 dominates vertex V_2 if V_1 can fulfill all the roles V_2 does with respect to agree sets - meaning V_1 could replace V_2 in any valid solution without loss of information.

The following cases A and B illustrate specific instances where such dominance can be identified and exploited to eliminate superfluous vertices.

- Case A - Direct Domination Based on Agree Set Coverage:

  A vertex V_x is said to dominate vertex V_y if the following condition holds:

  $$PAS(V_y) \subseteq CAS(V_x)$$

  This means that every potential agree set associated with V_y is already confirmed through existing connections of V_x to Critical Vertices. In other words, V_y contributes no unique agree set information that is not already accounted for by V_x. Therefore, V_y can be safely removed without compromising the completeness of the agree set coverage.

  Figure 6.11 illustrates an example of Case A, where vertex domination is determined based on the inclusion relationship between potential and confirmed agree sets. Vertex V_3 has identical $CAS(V_3)$ and $PAS(V_3)$, covering agree sets $\{Ag1,\ Ag2,\ Ag3,\ Ag4\}$, while vertices V_8 and V_9 have smaller or redundant agree set coverage. Since both $PAS(V_8) \subseteq CAS(V_3)$ and $PAS(V_9) \subseteq CAS(V_3)$, vertex V_3 dominates both V_8 and V_9. This makes V_8 and V_9 candidates for removal in the domination-based pruning process.

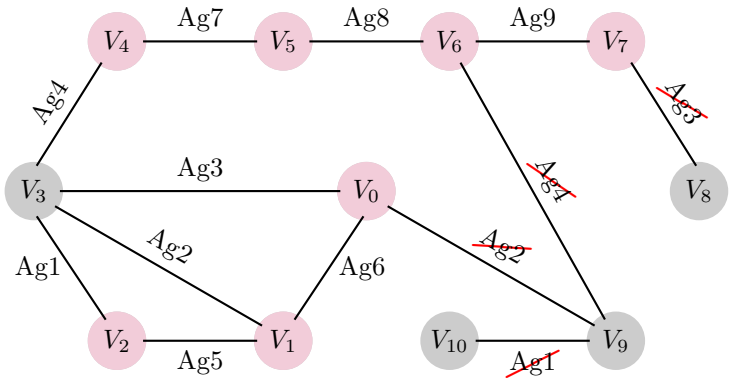


Figure 6.11: Example of Domination: Case A

- Case B - Adjusted Domination with Shared Non-Critical Neighbors:

  In scenarios where V_x and V_y both share a common neighbor that:

  – Is not currently marked as a Critical Vertex, and

  – Is connected to both V_x and V_y via edges labeled with the same agree set,

  that shared neighbor is disregarded in the computation of $PAS(V_x)$ and $PAS(V_y)$.

  After applying this exclusion, if the adjusted potential agree set of V_y is still a subset of the confirmed agree set of V_x, then we again say that:

  $$PAS(V_y) \subseteq CAS(V_x)$$

  and V_x dominates V_y. This refined condition helps prevent false negatives in dominance detection.

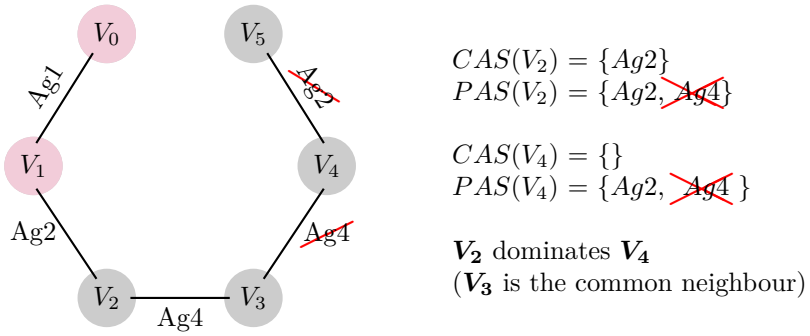


Figure 6.12: Example of Domination: Case B

Figure 6.12 illustrates an example of Case B, where domination is determined after discounting the influence of shared non-critical neighbors. Here, vertices V_2 and V_4 are both connected to vertex V_3 (not a Critical Vertex), and both edges from V_2 and V_4 to V_3 share the same agree set $Ag4$.

In this case, when computing the Potential Agree Sets (PAS) for V_2 and V_4, the shared neighbor V_3 is excluded. After this adjustment:

– V_2 retains $Ag2$ in its Confirmed Agree Set (CAS), meaning it already contributes to validation.

– V_4, on the other hand, has no confirmed agree sets and only potentially contributes to $Ag2$.

Since $PAS(V_4) \subseteq CAS(V_2)$, V_2 dominates V_4 under Case B. In practical terms, selecting V_2 ensures the same or better coverage of agree sets than selecting V_4, regardless of whether V_3 is included in the Critical Vertex list or not.

```
Add un-Validated agree sets to D_queue
while D_queue is not empty do
    for each agree set Ag_i in D_queue do
        Domination check for all vertices adjacent to edges labelled Ag_i
        Update the graph by removing dominated vertices and their adjacent edges
        Add newly identified Critical Vertices to the CV list
        if Ag_i is validated then
            Remove Ag_i from D_queue
        else
            Move Ag_i from D_queue to W_queue
        end if
    end for
    if W_queue is not empty then
        Select a vertex heuristically for removal
        Update the graph accordingly
        Move agree sets in W_queue back to D_queue
    end if
end while
```

Figure 6.13: Pseudocode for Domination

.

The domination-based algorithm aims to minimize the number of Critical Vertices by systematically removing vertices that are dominated by others, without compromising the completeness of agree set coverage. While it is theoretically possible to check domination relationships for all vertices simultaneously, this becomes computationally expensive for large graphs. Instead, the algorithm iterates over individual agree sets, checking only the vertices adjacent to edges labeled with each agree set. This localized checking significantly reduces computational overhead.

The process is managed using two queues:

- D_queue holds unvalidated agree sets that are actively being processed.
- W_queue temporarily stores agree sets that could not be validated through domination.

For each agree set in D_queue, the algorithm:

– Performs domination checks among the relevant vertices,

– Removes dominated vertices and updates the graph accordingly,

– Adds any newly identified Critical Vertices to the CV list,

– Removes the agree set from D_queue if it is validated.

If an agree set is validated during this process, it is removed from the D_queue; otherwise, it is moved to the W_queue. When all agree sets in the D_queue have been processed and no vertices are found to be dominated, a heuristic is applied - specifically, the vertex with the lowest Weighted Vertex Rank (WVR) is removed to enforce progress. The agree sets in the W_queue are then re-added to the D_queue, and the loop continues until all agree sets are successfully validated.

This approach balances efficiency and minimality, using structural properties of the graph to guide pruning, and resorts to ranking-based heuristics only when necessary.

6.3 Experiments

To evaluate the effectiveness of the proposed methods, we conducted experiments on a variety of real-world datasets. Table 6.1 presents a comparative summary of the results. The row entries denote the original dataset sizes, expressed as the number of columns multiplied by the number of rows. Before initiating the Critical Vertex (CV) selection process, isolated vertices were removed, and the resulting number of connected vertices is listed under the "IncidentV" column.

Source	Size(Col*Row)	AS#	IncidentV	EN	PVE	ER	WER	WVR	Dom
balance_scale	5*625	8	625	9	9	10	9	9	9
iris	5*150	10	71	18	16	16	14	15	14
chess	7*28056	12	28056	21	14	21	13	13	13
nursery	9*12960	16	12960	17	17	23	17	17	15
breast_cancer _wisconsin	11*699	43	417	74	62	68	64	63	61
adult	15*32561	55	8101	106	101	105	100	101	98
bridges	13*108	63	75	66	64	66	64	64	64
abalone	9*4177	64	434	123	113	114	111	112	110
letter	17*20000	81	4266	154	139	150	136	136	131
echocardiogram	13*132	84	99	77	70	75	70	70	70

Table 6.1: Size of Informative Armstrong Tables

.

Among the methods evaluated, the Domination approach consistently outperforms the others in terms of minimizing the size of the resulting Armstrong tables. This advantage stems from its Type A classification, which ensures minimality in solution construction. Although Type B methods such as PVE, ER, WER, and WVR tend to be more computationally efficient, they often do so at the expense of larger table sizes.

Notably, both Weighted Edge Rank (WER) and Weighted Vertex Rank (WVR) demonstrate better performance than other three methods. Their improvements are attributed to the inclusion of Critical Vertex weighting, which helps prioritize more informative nodes and edges during selection.

When compared to non-informative Armstrong tables (as shown in Tables 5.1 and 5.2), the informative Armstrong tables are often larger in size. However, the key advantage of informative Armstrong tables lies in their composition - they are subsets of real data, meaning the resulting tuples are semantically meaningful and interpretable. This makes them more suitable for practical applications where user-facing insights or real-world context is essential.

Interestingly, informative Armstrong tables are not always larger. As demonstrated in Experiment 3 (Chapter 5), randomly generated Armstrong tables - which can be viewed as informative Armstrong tables - are in some instances more compact than their non-informative counterparts. This occurs because synthetic, randomly generated tables can exhibit higher data density, allowing a greater number of agree sets to be discovered from fewer rows.

Chapter 7

Keys with NULLs

Keys play a foundational role in the relational model of databases, where they are essential for ensuring data integrity and enabling efficient data management. A key is defined as a minimal set of attributes that uniquely identifies each tuple (row) within a relation (table). In `SQL`, this concept is enforced through *primary keys*, which must satisfy two constraints: they must be `UNIQUE` (no two tuples share the same key value) and `NOT NULL` (every tuple must have a key value). These constraints collectively uphold the consistency and reliability of the data.

However, real-world datasets often contain incomplete or missing information, commonly represented as `NULL` values in relational databases. This poses challenges for key enforcement because `SQL` treats `NULL`s as unknown, making them non-comparable for equality. While `UNIQUE` constraints offer a form of possible key semantics by preventing definite duplicates, they fall short of enforcing true keys, as they cannot rule out potential duplicates involving `NULL`s. As a result, standard primary key constraints disallow `NULL`s, which can lead to data modeling challenges and the potential exclusion of otherwise valid records.

To address this issue, researchers have proposed more nuanced definitions of keys that account for the uncertainty introduced by `NULL`s. One influential approach introduces the concepts of *possible keys* and *certain keys*, as discussed by Köhler et al. [14]. In this framework, a *possible key* is a set of attributes that uniquely identifies tuples in at least one possible interpretation (or "possible world") of the data, considering different instantiations of `NULL`s. In contrast, a *certain key* guarantees uniqueness across all possible worlds. This distinction allows for more flexible and expressive modeling of incomplete data, enabling database systems to preserve the essence of uniqueness constraints even when some attribute values are unknown.

By extending the classical notion of keys to accommodate uncertainty, these approaches help bridge the gap between theoretical data integrity requirements and the practical realities of incomplete or evolving datasets. Such advancements contribute to more robust and semantically meaningful database designs in applications where missing information is unavoidable.

7.1 Possible keys and certain keys

A *possible key*, denoted as $p\langle X\rangle$, is said to be *satisfied* in a SQL relation r if there exists *at least one possible world* of r in which the attribute set X functions as a valid key. In other words, it is *possible* that X uniquely identifies tuples in some conceivable realization of the data represented by r.

In contrast, a *certain key*, denoted as $c\langle X\rangle$, is satisfied in a SQL relation r *only if* X acts as a key in *every possible scenario* consistent with the information present in r. That is, across all interpretations of the uncertain or incomplete data in r, the attribute set X must unambiguously identify tuples.

The symbol $\perp$ represents NULL in SQL, which signifies *unknown* or *missing* information. Although $\perp$ is not a conventional value within attribute domains, it is typically treated as a distinct marker to support the operational semantics of SQL.

These notions of *possible* and *certain* keys are central to understanding *data integrity* in databases that incorporate incomplete or uncertain information. Additional concepts - such as *strong/weak similarity*, *possible/certain worlds semantics*, and *strong/weak Agree Set* - provide further depth and formalism for analyzing key constraints in such contexts[14]:

Definition 7.1.1 (Strong and Weak Similarity)**.** Let r be a relation over a schema R, and let t, t' be two tuples in r. For an attribute set $X \subseteq R$, we define:

- **Strong similarity** on X: Tuples t and t' are said to be *strongly similar* on X, if

$$\forall A \in X, \quad t[A] = t'[A] \neq \perp.$$

  That is, for every attribute A in X, the values in t and t' are equal and non-null.

- **Weak similarity** on X: Tuples t and t' are said to be *weakly similar* on X, if

$$\forall A \in X, \quad t[A] = t'[A] \quad \vee \quad t[A] = \perp \quad \vee \quad t'[A] = \perp.$$

  That is, for every attribute A in X, either both tuples agree on the value, or at least one of them contains a null value ($\perp$) in that attribute.

△

Definition 7.1.2 (Possible and Certain Keys)**.** Let r be a relation over a schema R, where the special symbol $\perp$ denotes `NULL` values - representing unknown or missing data. A *possible world* of r is a fully instantiated version of the relation, obtained by replacing each occurrence of $\perp$ independently with a concrete value from the corresponding attribute's domain.

Let $X \subseteq R$ be a set of attributes. Then:

- X is a **possible key** for r, denoted $p\langle X \rangle$, if there exists at least one possible world of r in which X functions as a key - that is, no two distinct tuples agree on all values in X in that world.
- X is a **certain key** for r, denoted $c\langle X \rangle$, if X serves as a key in *every* possible world of r. In other words, X uniquely identifies tuples regardless of how the nulls in r are instantiated.

△

Definition 7.1.3 (Strong and Weak Agree Sets)**.** Let t and t' be two tuples over a relation schema R. An attribute set $X \subseteq R$ is called:

- a **strong agree set** for t and t' if t and t' are *strongly similar* on X.
- a **weak agree set** for t and t' if t and t' are *weakly similar* on X.

Now, consider a relation r over schema R. We define:

- $\mathcal{AG}^s(r)$: the set of all *maximal strong agree sets* among distinct tuples in r.
- $\mathcal{AG}^w(r)$: the set of all *maximal weak agree sets* among distinct tuples in r.

A set $X \subseteq R$ is *maximal* in this context if there is no proper superset $X' \supset X$ and $X' \subseteq R$ such that t and t' still satisfy the corresponding similarity condition on X'.

△

Definition 7.1.4 (Strong and Weak Anti-Keys)**.** Let r be a relation over a schema R, and let $X \subseteq R$ be a set of attributes. We define:

- X is a **strong anti-key** for r, denoted by $\neg p\langle X \rangle$, if there exist two distinct tuples $t, t' \in r$ such that t and t' are *strongly similar* on X; that is,

$$\forall A \in X, \quad t[A] = t'[A] \neq \perp.$$

  In this case, X cannot be a possible key, since it does not uniquely identify tuples.

- X is a **weak anti-key** for r, denoted by $\neg c\langle X\rangle$, if there exist two distinct tuples $t, t' \in r$ such that t and t' are *weakly similar* on X; that is,

$$\forall A \in X, \quad t[A] = t'[A] \ \vee \ t[A] = \bot \ \vee \ t'[A] = \bot.$$

  In this case, X cannot be a certain key, since it fails to distinguish between tuples under all possible interpretations.

An anti-key is said to be *maximal* if it is not properly contained in any larger anti-key of the same type:

- A strong anti-key $\neg p\langle X\rangle$ is **maximal** if there does not exist an attribute set $Y \supset X$ such that $\neg p\langle Y\rangle$ also holds.
- A weak anti-key $\neg c\langle X\rangle$ is **maximal** if there does not exist an attribute set $Y \supset X$ such that $\neg c\langle Y\rangle$ holds.

We denote:

- $\mathcal{A}^s_{\max}$: the set of all *maximal strong anti-keys* in relation r.
- $\mathcal{A}^w_{\max}$: the set of all *maximal weak anti-keys* in relation r. △

7.2 Building Armstrong tables for possible and certain keys

For ease of reference, an *Armstrong table* (see Definition 1.2.1) is a valuable construct for representing and analyzing data constraints through concrete examples. Given a set of constraints Σ, an Armstrong table for Σ is a relation that satisfies every constraint $\sigma \in \Sigma$, while deliberately violating all constraints that are *not* logically implied by Σ. In this way, the table serves as an exact characterization of the expressive power of Σ, capturing all and only those constraints that follow from it.

When working with *nullable schemas* - schemas in which attributes may contain `NULL` values (denoted by $\bot$) - the concept of an Armstrong table is extended to accommodate the semantics of incomplete information. An *Armstrong table over nullable schemas* is a systematically constructed relation that:

- Satisfies all constraints in Σ under the possible-world semantics associated with NULLs, ensuring that every implied constraint holds in all or some interpretations;
- Violates every constraint not entailed by Σ, while appropriately accounting for the presence of NULL values.

This extended form of the Armstrong table is particularly valuable in settings with incomplete data, as it enables rigorous validation of constraint implication in the presence of uncertainty introduced by NULL values.

7.2.1 Existence of Armstrong table

When considering only key constraints over *null-free* schemas, the existence of an Armstrong table is guaranteed. In such settings, where all attributes are assumed to be non-null, the semantics of keys align with classical definitions, and Armstrong tables can be constructed to represent the exact implication closure of a given constraint set.

However, when extending to *nullable schemas*, particularly those involving *possible* and *certain keys*, the existence of an Armstrong table becomes *conditional*. This is due to the additional complexity introduced by the presence of `NULL` values and the differing semantics of key satisfaction under incomplete information.

To formalize this scenario, let R be a relation schema, and let $R_s \subseteq R$ denote the subset of attributes declared to be *null-free*. The pair (R, R_s) defines a schema where attributes in $R \setminus R_s$ may contain `NULL` values (denoted $\bot$).

Consider the constraint set $\Sigma = \{c\langle B\rangle\}$ over the relation schema $(R, R_s) = (AB, A)$. Let I be an instance over this schema, and suppose it contains a tuple $t \in I$ such that $t[B] = \bot$. Since $c\langle B\rangle$ is a *certain key*, this constraint requires that no two distinct tuples can be *weakly similar* on attribute B. However, the presence of $\bot$ in $t[B]$ implies

that any additional tuple $t' \neq t$ in I would be weakly similar to t on B, thus violating $c\langle B\rangle$. Therefore, the instance I must contain only a single tuple t to satisfy Σ.

Yet, this singleton instance $I = \{t\}$ inadvertently satisfies another constraint: $c\langle A\rangle$, since there are no distinct tuples that could violate uniqueness on attribute A, which is in the null-free set R_s. However, $c\langle A\rangle \notin \Sigma$, nor is it logically implied by Σ, meaning the Armstrong table unintentionally satisfies an unintended constraint.

This example demonstrates that under certain conditions the construction of an Armstrong table may be infeasible. The core issue arises from the interplay between NULL semantics and constraint satisfaction, which may prevent the strict separation between implied and non-implied constraints required for an Armstrong table to exist.

Definition 7.2.1 (Armstrong Table and Pre-Armstrong Table over Nullable Schemas)**.** Let (R, R_s, Σ) be a database schema where:

- R is the set of all attributes,
- $R_s \subseteq R$ is the subset of attributes that are declared to be *null-free* ,
- Σ is a set of constraints, consisting of *possible keys* and *certain keys*.

An *Armstrong table* for Σ over the schema (R, R_s) is a finite relation instance that satisfies the following conditions:

(a) It satisfies every possible key and certain key that is logically implied by Σ. That is, for all $\sigma \in \mathrm{cl}(\Sigma)$, the table satisfies σ.

(b) It violates every possible key and certain key that is *not* implied by Σ. That is, for all $\sigma \notin \mathrm{cl}(\Sigma)$, the table does not satisfy σ.

(c) For every nullable attribute $A \in R \setminus R_s$, the table includes at least one tuple in which A takes the value $\perp$ (i.e., `NULL`). This ensures that the presence and effect of NULLs are explicitly captured.

A *pre-Armstrong table* for Σ over (R, R_s) is a relation instance that satisfies conditions **(a)** and **(b)** above, but not necessarily **(c)**. That is, it correctly reflects the implication structure of Σ, but may omit NULL values for some nullable attributes. △

It has been shown that Armstrong tables over nullable schemas do not always exist. Their existence depends on the structure of the schema and the nature of the constraints in the set Σ. The following identifies both necessary and sufficient conditions under which Armstrong tables can or cannot be constructed [14].

Let (R, R_s, Σ) be a schema, where R is the full set of attributes, $R_s \subseteq R$ is the set of attributes declared to be null-free, and Σ is a set of possible and certain key constraints. Suppose that $\Sigma \not\models c\langle\emptyset\rangle$, i.e., the empty set is not a certain key. Then:

i) If $\Sigma \not\models c\langle X\rangle$ for every $X \subseteq R \setminus R_s$, then an Armstrong table *does* exist.

ii) If there exists $X \subseteq R \setminus R_s$ such that $\Sigma \models c\langle X\rangle$ and $|X| \leq 2$, then an Armstrong table *does not* exist.

To elaborate on these results:

[1] **Case where the empty key is implied:** If $\Sigma \models c\langle\emptyset\rangle$, then any valid instance must contain *exactly one* tuple, since the empty key implies that all tuples must be identical. To satisfy condition **(c)** in Definition 7.2.1 (i.e., ensuring each nullable attribute contains a NULL), this single tuple must contain $\perp$ for every attribute $A \in R \setminus R_s$. In this case, this single-row relation is trivially an Armstrong table.

[2] **Case where no certain key is confined to nullable attributes:** If $\Sigma \not\models c\langle X\rangle$ for every $X \subseteq R \setminus R_s$, then every certain key involves at least one non-nullable attribute. Formally, for every Y such that $\Sigma \models c\langle Y\rangle$, it holds that $|Y \cap R_s| > 0$. This guarantees that at least one attribute in each key is never NULL, which helps avoid accidental creation of weak anti-keys. Thus, condition **(b)** in Definition 7.2.1 is satisfied, and an Armstrong table can be constructed.

[3] **Case where a small certain key uses only nullable attributes:** If $\Sigma \vDash c\langle X\rangle$ for some $X \subseteq R \setminus R_s$ with $|X| \leq 2$, then an Armstrong table does *not* exist. For instance, suppose $c\langle AB\rangle \in \Sigma$, where $AB \subseteq R \setminus R_s$. To satisfy condition **(c)** (presence of NULLs), the Armstrong table would include a tuple t_A with $t_A[A] = \perp$ and a tuple t_B with $t_B[B] = \perp$. This results in t_A and t_B being weakly similar on AB, thereby violating the certain key $c\langle AB\rangle$, contradicting condition **(a)**.

However, if all certain keys that are subsets of $R \setminus R_s$ have size greater than two (i.e., $|X| > 2$), then Armstrong tables are constructable. Including more attributes in the key provides additional flexibility in tuple design.

In this and the following chapters, we focus on constructing Armstrong tables for the favorable case i), where every certain key involves at least one non-nullable attribute. This scenario is particularly relevant for practical applications, as it ensures the existence of Armstrong tables and simplifies their construction, while still covering a wide range of real-world database schemas.

7.2.2 Constructing Armstrong table

In cases where every certain key in Σ includes at least one non-nullable attribute (i.e., involves at least one attribute from R_s), Armstrong tables can be effectively constructed. This section outlines the structured procedure for constructing such tables, based on the techniques and theoretical foundations provided in [14].

1. **Computation of Strong and Weak Agree Sets:**

   According to Lemma 3 in [14], given the set of possible keys Σ^P and certain keys Σ^C, the maximal strong and weak agree sets - denoted $\mathcal{A}^s_{\max}$ and $\mathcal{A}^w_{\max}$, respectively - can be computed as follows:

   $$\mathcal{A}^s_{\max} = \left\{ X \in \overline{\mathcal{T}_r(\Sigma^C \cup \Sigma^P)} \;\middle|\; \neg(X \subseteq R_s \wedge \exists Y \in \mathcal{A}^w_{\max},\ X \subset Y) \right\}$$

   Compute the transversals (as defined in Definition 2.2.4) of the possible and certain keys, where each item's complement serves as a strong anti-key. By collecting all maximal strong anti-keys and removing redundant entries, we form $\mathcal{A}^s_{max}$. These redundant entries can be excluded because enforcing weak anti-keys ensures that the corresponding strong anti-keys are also enforced, given the null free constraints.

   For example, consider:
   $$\mathcal{AG}^s = \{AB\}, \quad \mathcal{AG}^w = \{ABC\}, \quad R_s = \{A, B\}.$$
   In this case, enforcing weak similarity on ABC automatically implies strong similarity on AB, since both A and B are non-nullable.

   $$\mathcal{A}^w_{\max} = \overline{\mathcal{T}_r\left(\Sigma^C \cup \{X \in \Sigma^P \mid X \subseteq R_s\}\right)} \setminus \mathcal{A}^s_{\max}$$

   Compute the transversals of certain keys and possible keys that have null-free attributes (these are effectively also certain keys). Use the complements of these transversals as weak anti-keys. The set of all maximal weak anti-keys, excluding those found in $\mathcal{A}^s_{max}$ constitutes $\mathcal{A}^w_{max}$.

2. **Armstrong-Set $\mathcal{W}$:**

   As introduced in Theorem 3 and computed in Algorithm 1 of [14], the *Armstrong-set* $\mathcal{W}$ represents a sub-schema structure where null markers co-occur in tuple pairs. In the scenario under consideration - where all certain keys involve at least one attribute from R_s - it holds that:

   $$\mathcal{W} = R \setminus R_s$$

   That is, $\mathcal{W}$ consists of all nullable attributes and itself forms a weak anti-key. This set is used to develop a general construction for pre-Armstrong tables, allowing us to introduce $\perp$ in nullable attributes that do not yet contain a $\perp$.

3. **Tuple Construction for Strong Anti-Keys:**

   For every strong anti-key $\neg p\langle X\rangle \in \mathcal{A}^s_{\max}$, generate two tuples t and t' such that:

$$t[A] = t'[A] \quad \text{for all } A \in X, \qquad t[B] \neq t'[B] \quad \text{for all } B \in R \setminus X$$

   These tuples strongly agree only on X, thereby enforcing that X is not a possible key.

4. **Tuple Construction for Weak Anti-Keys:**

   For every weak anti-key $\neg c\langle X\rangle \in \mathcal{A}^w_{\max}$, generate tuple pairs t, t' with the following properties:

   (a) They *agree* on the intersection with the null-free attributes: $t[A] = t'[A]$ for all $A \in X \cap R_s$.

   (b) Let $Y = X \cap (R \setminus R_s)$, representing the nullable portion of X. Let $Y = Y_1 \cup Y_2$, where $Y_1, Y_2 \in \mathcal{W}$. Assign NULLs as:

$$t[A] = \bot \text{ for } A \in X \cap Y_1, \qquad t'[A] = \bot \text{ for } A \in X \cap Y_2$$

   This ensures t and t' are weakly similar on X, thus violating $c\langle X\rangle$.

   (c) Fill all remaining attributes of t and t' with distinct, arbitrary domain values to avoid accidental agreements on other attributes.

5. **Finalization: Ensuring NULL Presence:**

   To convert the resulting pre-Armstrong table into a full Armstrong table, ensure that every nullable attribute $A \in R \setminus R_s$ appears with a NULL value in at least one tuple t in the relation. This step guarantees satisfaction of condition (c) in Definition 7.2.1, thereby completing the Armstrong table construction.

Let's walk through an illustrative example that demonstrates each step of constructing an Armstrong table over nullable schemas:

Example 5. Let the schema be $R = \{N, D, G, S\}$, the null-free subset be $R_s = \{N\}$, and the constraint set $\Sigma = \{c\langle ND\rangle,\ p\langle S\rangle\}$, where $c\langle ND\rangle$ is a certain key and $p\langle S\rangle$ is a possible key.

Following the Armstrong table construction steps:

1. **Compute Maximal Strong and Weak Anti-Keys:**

   - For strong anti-keys::

$$\mathcal{T}_r\big(\{ND, S\}\big) = \{NS,\ DS\} \quad \Rightarrow \quad \mathcal{A}^s_{\max} = \{\overline{NS},\ \overline{DS}\} = \{DG,\ NG\}$$

   - For weak anti-keys:

$$\mathcal{T}_r\big(\{ND\} \cup \emptyset\big) = \{N,\ D\} \quad \Rightarrow \quad \overline{\{N\}} = DGS,\ \overline{\{D\}} = NGS$$

$$\mathcal{A}^w_{\max} = \{DGS,\ NGS\} \setminus \{DG,\ NG\} = \{DGS,\ NGS\}$$

2. **Armstrong-Set:**

   We compute the Armstrong-set $\mathcal{W}$ as:

$$\mathcal{W} = R \setminus R_s = DGS$$

3. **Add Tuple Pairs for Strong Anti-Keys:**

   For each $X \in \mathcal{A}^s_{\max}$, we add tuple pairs that agree only on X and disagree on all other attributes. The following tuples are constructed:

N	D	G	S	
n_1	d_1	g_1	s_1	DG
n_2	d_1	g_1	s_2	
n_3	d_3	g_3	s_3	NG
n_3	d_4	g_3	s_4	

Figure 7.1: Add Tuple Pairs for Strong Anti-Keys

4. **Add Tuple Pairs for Weak Anti-Keys:**

   We now add tuple pairs that are weakly similar on each $X \in \mathcal{A}^w_{\max}$. The following logic is applied:

(a) For $X = DGS$, observe $X \cap R_s = \emptyset$; For $X = NGS$, observe $X \cap R_s = \{N\}$.

(b) $Y_1 = Y_2 = DGS$,

- $X = NGS$: $X \cap Y_1 = GS$, $X \cap Y_2 = GS$;
- $X = DGS$: $X \cap Y_1 = DGS$, $X \cap Y_2 = DGS$.

(c) For $X = DGS$, the remaining attribute is D; For $X = NGS$, the remaining attribute is N.

The results obtained at each step are as follows:

N	D	G	S
n_1	d_1	g_1	s_1
n_2	d_1	g_1	s_2
n_3	d_3	g_3	s_3
n_3	d_4	g_3	s_4
n_5			
n_5			

$\Rightarrow$

N	D	G	S
n_1	d_1	g_1	s_1
n_2	d_1	g_1	s_2
n_3	d_3	g_3	s_3
n_3	d_4	g_3	s_4
n_5		$\perp$	$\perp$
n_5		$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$
	$\perp$	$\perp$	$\perp$

$\Rightarrow$

N	D	G	S
n_1	d_1	g_1	s_1
n_2	d_1	g_1	s_2
n_3	d_3	g_3	s_3
n_3	d_4	g_3	s_4
n_5	d_5	$\perp$	$\perp$
n_5	d_6	$\perp$	$\perp$
n_7	$\perp$	$\perp$	$\perp$
n_8	$\perp$	$\perp$	$\perp$

Figure 7.2: Add Tuple Pairs for Weak Anti-Keys

5. **Finalization: Ensuring NULL Coverage:**

   To meet the full Armstrong table requirements, ensure that each nullable attribute $A \in DGS$ appears with a NULL in at least one tuple. As seen in the tuples above:

$$\exists t_i : t_i[D] = \perp, \quad \exists t_j : t_j[G] = \perp, \quad \exists t_k : t_k[S] = \perp$$

   This confirms that all nullable attributes have been covered with NULLs.

The final table on the right shown above (Figure 7.2) constitutes a valid Armstrong table. It:

- Satisfies all constraints in Σ, namely $c\langle ND \rangle$ and $p\langle S \rangle$
- Violates all possible/certain keys not implied by Σ
- Includes NULLs for each nullable attribute D, G, S

Thus, it meets all criteria of Definition 7.2.1, and is a complete Armstrong table for the given schema and the constraint set.

Chapter 8

Sublinear Armstrong table for possible and certain keys

Using the approach introduced in the previous chapter, the Armstrong tables constructed for sets of possible and certain keys have a minimum size of $2m + 2n$, where $m = |\mathcal{A}^s_{\max}|$ is the number of maximal strong anti-keys, and $n = |\mathcal{A}^w_{\max}|$ is the number of maximal weak anti-keys. This assumes that no additional tuples are required in the final step of construction, transitioning from the pre-Armstrong table to the completed Armstrong table. However, this size can be further optimized under certain conditions.

To reduce the size of the Armstrong table, adjacent tuple pairs can be constructed for strong agree sets. This optimization allows the linear contribution of $\mathcal{A}^s_{\max}$ to be reduced from $2m$ to $m + 1$, since each strong agree set can be realized by the agreement between two consecutive tuples.

However, this technique does not extend to weak agree sets. Due to the presence of NULL values, tuples may weakly agree with multiple - or even all - other tuples, which can lead to the unintended formation of additional weak agree sets.

Consider Example 5:

$$R = \{N, D, G, S\}, R_s = \{N\}, \mathcal{A}^s_{\max} = \{DG,\ NG\}, \mathcal{A}^w_{\max} = \{DGS,\ NGS\}$$

If we construct the Armstrong table for weak agree sets in the same manner as for strong agree sets, we might obtain the following:

N	D	G	S	
n_1	d_1	g_1	s_1	DG
n_2	d_1	g_1	s_2	NG
n_2	d_2	g_1	s_3	DGS
n_3	$\perp$	g_1	s_3	NGS
n_3	d_3	g_1	$\perp$	

Here, the last two tuples weakly agree on $NDGS$, even though that agree set was not intended. This illustrates that each weak agree set must be explicitly constructed using carefully chosen tuple pairs to prevent unintended overlap.

In summary, with chaining applied to strong anti-keys, the total size of the Armstrong table can be reduced from the naive bound of $2m + 2n$ to an optimized size of $m + 1 + 2n$, which is linear in the number of anti-keys. However, in scenarios where the constraint set Σ gives rise to a large number of strong and weak anti-keys, even this optimized size may be prohibitively large for practical use.

However, by employing targeted construction techniques and incorporating additional consistency checks, it is still possible to generate Armstrong tables of sublinear size. These optimized tables, while smaller in the number of

tuples, retain their effectiveness in validating constraint implication and detecting violations. As a result, they offer a substantial advantage in real-world scenarios by enabling domain experts to identify and analyze design issues using a significantly reduced set of representative records.

8.1 Algorithm Principle

This section introduces the principle behind the algorithm designed to construct a *sublinear-size Armstrong table* for a given schema. The input to the algorithm is either:

- A schema (R, R_s, Σ), where $\Sigma = \Sigma^P \cup \Sigma^C$ contains possible and certain key constraints, or
- An existing relation instance that may contain NULL values.

The goal is to generate a sublinear-size Armstrong table that satisfies the following properties:

- All key constraints implied by Σ are satisfied.
- All key constraints not implied by Σ are violated.
- NULL values are present for every attribute in $R \setminus R_s$.

If no such sublinear Armstrong table can be constructed, the algorithm returns `false`, indicating that such a construction is not feasible under the given constraints.

Graph-Based Representation

The algorithm utilizes a graph-based model to represent the structure of the Armstrong table:

- **Vertices** correspond to tuples.
- **Edges** are labeled with strong and/or weak *agree sets*, indicating attribute sets on which the connected tuples must strongly and/or weakly agree.
- **Vertex labels** indicate the presence of NULLs in particular attributes.

To prevent accidental agreement on certain keys (when assigning anti-key values containing NULL), a key rule is enforced: if a NULL value is assigned to attribute A at a vertex, then *all adjacent edges* must include A in their weak agree set labels. This restriction ensures that weak agreement is intentional and preserves constraint semantics.

Assignment Procedure

Strong and weak agree sets are assigned in two distinct phases:

1. **Strong Agree Set Assignment:**

   The graph is initially constructed by assigning all maximal strong agree sets $\mathcal{A}^s_{\max}$ to specific edges. This assignment process is similar to the one described in detail in Chapter 5 and illustrated in Figure 5.1. The key difference in this case is that if a sublinear assignment - i.e., one using no more than m tuples relative to the number of strong agree sets - cannot be achieved, we do not immediately return false. Instead, we proceed with a linear number of tuples $(m + 1)$ and continue to the next step to investigate whether further tuple reduction is possible for the weak agree sets.

2. **Assignment of Weak Agree Sets:**

   Once the graph has been constructed with all strong agree sets $\mathcal{A}^s_{\max}$ assigned to edges, the next step is to integrate the weak agree sets $\mathcal{A}^w_{\max}$. This involves identifying suitable edges - including those already assigned strong agree sets - while appropriately assigning NULL values to the vertices.

   Formally, given a strong agree set $X \in \mathcal{A}^s_{\max}$ and a weak agree set $Y \in \mathcal{A}^w_{\max}$, the weak agree set Y can potentially be assigned to the same edge as X if the following conditions hold:

- $X \subseteq Y$, and
- $Y \setminus X \subseteq R \setminus R_s$.

To assign a weak agree set $\mathcal{AG}_w$ to an edge $e_i = (V_j, V_k)$, the following steps are taken:

- **Near-Cycle Check for Non-Nullable Attributes:** Compute $\mathcal{AG}_w \cap R_s$, the portion of the agree set over non-nullable attributes. If a strong agree set $\mathcal{AG}_s \cap R_s = \mathcal{AG}_w \cap R_s$ is already assigned to e_i, skip this check since it has been previously validated. Otherwise, perform a near-cycle check to ensure assigning $\mathcal{AG}_w \cap R_s$ does not transitively imply a constraint not in Σ.
- **NULL Compatibility Check for Nullable Attributes:** To ensure that weak agreement is enforced only where permitted, we examine the *nullable portion* of the weak agree set, defined as

  $$\mathcal{AG}_w^{\text{null}} := \mathcal{AG}_w \cap (R \setminus R_s),$$

  which captures the subset of weak agreement attributes that are nullable. Our goal is to assign `NULL` values to attributes at only one of the two incident vertices, either V_j or V_k.

  We begin by examining the lower-indexed vertex V_j. Let $\mathcal{AG}_{s(1)}, \ldots, \mathcal{AG}_{s(n)}$ denote the strong agree sets associated with the edges adjacent to V_j. If the union of these strong agree sets fully overlaps with the nullable portion of the weak agree set, formally, if

  $$\left(\bigcup_{i=1}^{n} \mathcal{AG}_{s(i)} \right) \cap \mathcal{AG}_w^{\text{null}} = \mathcal{AG}_w^{\text{null}},$$

  then assigning `NULL` to any attribute in $\mathcal{AG}_w^{\text{null}}$ at V_j would violate existing strong agree sets. In this case, V_j is marked as unsuitable for NULL assignment. The same procedure is then applied to V_k.

  If instead there exists at least one attribute in $\mathcal{AG}_w^{\text{null}}$ that is not covered by any adjacent strong agree set, that is,

  $$\mathcal{AG}_w^{\text{res}} := \mathcal{AG}_w^{\text{null}} \setminus \left(\bigcup_{i=1}^{n} \mathcal{AG}_{s(i)} \right) \neq \emptyset,$$

  then V_j is eligible to be assigned `NULL` values on those uncovered attributes, denoted $\mathcal{AG}_w^{\text{res}}$.
- **Enforce NULL Propagation on Adjacent Edges:** Once a suitable vertex V_x is found, assign NULLs to all attributes in $\mathcal{AG}_w^{\text{res}}$ at V_x. To prevent unintended violation of certain keys, every edge adjacent to V_x must be forced to include these attributes in their weak agree sets. These weak agree sets are then checked to ensure that they do not violate any certain keys. This guarantees the integrity of the constraint representation when NULLs are introduced.

If no applicable edges are found and some agree sets remain unassigned, the graph is incrementally expanded by adding one vertex at a time, until it reaches its predefined linear size. As noted at the beginning of this chapter, the linear size of an Armstrong table for possible and certain keys, denoted by S, depends on the number of maximal strong and weak maximal agree sets. It is computed using the following formula:

$$S = (|\mathcal{A}^s_{\max}| + 1) + 2 \cdot |\mathcal{A}^w_{\max}|\,,$$

where $|\mathcal{A}^s_{\max}|$ is the number of maximal strong agree sets, and $|\mathcal{A}^w_{\max}|$ is the number of maximal weak agree sets.

If even after reaching size S the assignment is not possible, the algorithm concludes that no sublinear Armstrong table can be generated and terminates with a negative result. The corresponding flow chart is shown in Figure 8.1.

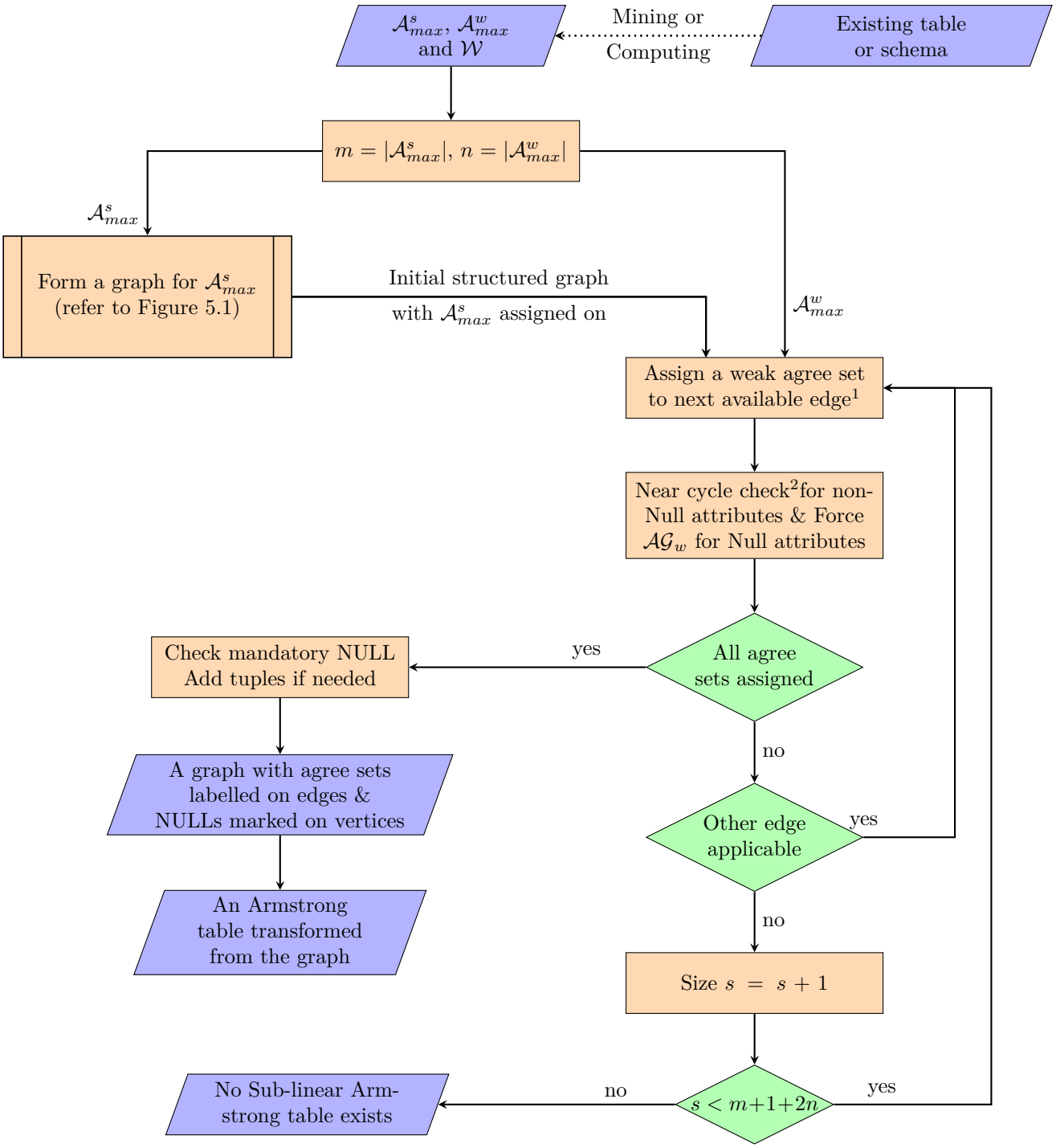


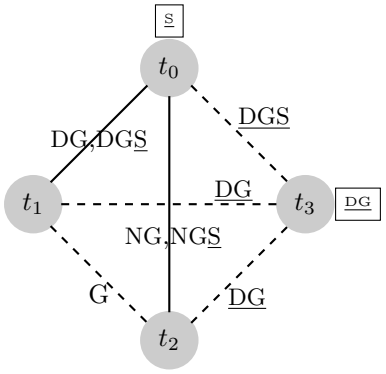
Figure 8.1: Algorithm summary

[1]Combining strong and weak agree sets on the same edges wherever possible.
[2]Closure computations are also performed after near cycle checks, though it has no impact here, as we are only considering keys.

Once more, we illustrate the algorithm in action using Example 5. See Figure 8.2 for a visual representation of the assignment process.

Let the schema be defined as: $R = \{N, D, G, S\}$, with non-nullable attributes $R_s = \{N\}$ and constraints $\Sigma = \{p\langle S\rangle, c\langle ND\rangle\}$. From this, we derive:

- Maximal strong agree sets: $\mathcal{A}^s_{\max} = \{DG, NG\}$
- Maximal weak agree sets: $\mathcal{A}^w_{\max} = \{NGS, DGS\}$
- Armstrong-set: $\mathcal{W} = \{DGS\}$



Assignment Sequence for Figure 8.2

```
Assign strong agree sets in A^s_max:
  DG on (t0, t1) -- success
  NG on (t0, t2) -- success
Assign weak agree sets in A^w_max:
  NGS on (t0, t1) -- fails
  NGS on (t0, t2) -- success
  Mark S as NULL on t0
  Force S to weakly agree on (t0, t1) and (t0, t3)
  DGS on (t0, t1) -- already enforced
Mandatory NULL check:  D and G still unsatisfied
  Add extra tuple t3 and mark DG as NULL
  Force DG to weakly agree on:
    (t0, t3), (t1, t3), (t2, t3)
```

Figure 8.2: Example of the Algorithm Assignment Process

In Example 5, the computed linear size bound for a pre-Armstrong table is $S = 7$. However, the algorithm successfully constructs a table of size 3. A fourth vertex is added in the final step during the Mandatory NULL check, resulting in an output graph with 4 vertices, each corresponding to a tuple. This reduction illustrates that, in practice, sublinear Armstrong tables can be achieved when weak agree sets structurally overlap with strong ones, or when `NULL` values can be effectively shared across tuples.

8.2 Approaches for Finding Sublinear Armstrong Tables for Possible and Certain Keys

Building on the foundation established in the previous sections, we present algorithmic strategies aimed at constructing sublinear Armstrong tables for possible and certain keys. These approaches are grounded in the structured use of strong and weak agree sets, with the goal of minimizing the number of tuples required.

The core idea is to leverage structural properties of agree sets - particularly their overlap and interaction through NULL values - to optimize the size of the Armstrong table. This is especially important when the number of constraints becomes large, where naive methods would otherwise produce impractically large tables.

- **Sequential Assignment (Baseline)**
  This is a simple and direct approach that depicted in Figure 8.1. It involves two main stages:

  1. Strong Agree Set Assignment: Construct an initial structured graph by assigning all strong agree sets from $\mathcal{A}^s_{\max}$ to edges, as outlined in Chapter 5. If a sublinear assignment fails at this stage, the algorithm continues with a linear-sized configuration to accommodate further steps.
  2. Weak Agree Set Assignment: Once the graph has been initialized with strong agree sets, the algorithm proceeds to assign each weak agree set from $\mathcal{A}^w_{\max}$ to available edges. During this phase, NULL values are introduced and care is taken to avoid unintentional agreements that could violate the semantics of anti-keys. No backtracking is performed during this stage.

  This sequential strategy serves as a baseline for evaluating more advanced optimization techniques. While it is not guaranteed to produce a minimal Armstrong table, it often succeeds in yielding a sublinear-sized table when there is sufficient structural overlap between agree sets or effective reuse of NULLs.

Matching Strategy for Size Reduction

To further reduce the size of Armstrong tables, we introduce a *matching strategy* that enables simultaneous assignment of compatible strong and weak agree sets to the same edge. This approach leverages structural containment between agree sets to reduce the total number of required tuples.

As demonstrated in Example 5, it is possible for a strong agree set and a weak agree set to coexist on the same edge. This observation motivates a proactive pairing strategy that aims to maximize such overlaps.

Recall that such coexistence is permitted when the weak agree set extends the strong agree set only over nullable attributes. Formally, given $X \in \mathcal{A}^s_{\max}$ and $Y \in \mathcal{A}^w_{\max}$, they can be paired on the same edge if:

$$X \subseteq Y \quad \text{and} \quad Y \setminus X \subseteq R \setminus R_s.$$

In this case, the additional attributes in $Y \setminus X$ must be nullable, allowing one of the two incident tuples to be assigned `NULL` values for those attributes. This ensures that strong agreement does not inadvertently extend to a superset of X, which could violate some possible key.

As illustrated in Figure 8.2, the strong agree set DG can be paired with the weak agree set DGS, since $DG \subset DGS$ and $S \in R \setminus R_s$. Both agree sets are simultaneously assigned to the edge (t_0, t_1), and S is marked as NULL on either t_0 or t_1. A similar pairing applies for NG and NGS.

Based on this principle, we propose the following matching strategies to guide the construction of Armstrong tables:

- **Matched Pair**
  This basic strategy matches the first compatible strong and weak agree set pairs that satisfy the above conditions. Once matched, the paired agree sets are assigned together. Any remaining agree sets that cannot be matched are handled using the baseline *Sequential Assignment* approach.

- **Most Matched**
  This strategy prioritizes matching based on the degree of overlap between agree sets. Given multiple possible matches for a weak agree set Y, we prefer the strong agree set X that shares the largest number of attributes with Y. For example, if $AB, BCD \in \mathcal{A}^s_{\max}$ and $ABCD \in \mathcal{A}^w_{\max}$, with $R = ABCD$ and $R_s = B$, then the pair $\{BCD, ABCD\}$ is preferred over $\{AB, ABCD\}$ due to the higher attribute overlap.

- **Maximum Matching**
  To optimize tuple reuse and reduce table size, we pair strong agree sets with weak agree sets using the Hopcroft–Karp algorithm [32] to compute the maximum matching before assignment begins.

  The Hopcroft–Karp algorithm is an efficient method for finding a maximum matching in a bipartite graph. A bipartite graph consists of two disjoint sets of vertices, typically denoted as U and V, with edges existing only between vertices from different sets. In this context, we construct a bipartite graph where the set of maximal strong agree sets $\mathcal{A}^s_{\max}$ corresponds to U, and the set of maximal weak agree sets $\mathcal{A}^w_{\max}$ corresponds to V. An edge is drawn between a strong and a weak agree set if they are compatible for shared assignment in the construction process.

  The objective is to identify the largest possible set of disjoint pairs - i.e., a maximum matching - such that no agree set participates in more than one pair. This pairing enables more efficient tuple construction by allowing each matched pair to share tuples, thereby reducing the overall size of the Armstrong table.

8.3 Experiments

To evaluate the proposed methods, we conducted a series of experiments using other real-world datasets from https://relational-data.org/dataset/Hockey. Unlike previously used datasets, these contain `NULL` values, allowing us to test the algorithm's behavior under more realistic and complex scenarios. Each dataset was analyzed to

extract maximal strong and weak agree sets, and various strategies were applied to construct sublinear Armstrong tables.

The table below compares the performance of different strategies:

No.	Dataset	$\lvert\mathcal{A}^s_{\max}\rvert$	$\lvert\mathcal{A}^w_{\max}\rvert$	Linear Size	Sequence	Matched	MostM	MaxM
I	Hockey.AwardsMisc	1	1	4	3	3	3	3
II	Hockey.AwardsCoaches	2	2	7	3	3	3	3
III	Hockey.ScoringSup	2	2	7	4	4	4	4
IV	Hockey.TeamVsTeam	3	3	10	7	7	7	7
V	Hockey.AwardsPlayers	4	3	11	6	6	6	6
VI	Hockey.TeamsSC	6	5	17	6	6	6	6
VII	Hockey.SeriesPost	21	16	54	24	22	22	22
VIII	Hockey.TeamsPost	149	15	180	44	37	37	35

As seen in the results, all strategies perform similarly on datasets with small numbers of agree sets. However, for more complex datasets (e.g., Dataset VII and VIII), matching-based strategies begin to demonstrate improvements. In particular, *Maximum Matching* yields the smallest Armstrong table in the largest dataset, indicating its effectiveness in optimizing complex dependency graphs.

To further evaluate scalability and performance, we conducted an additional set of experiments in which only keys of size three or fewer were considered. This restriction helps limit the number of resulting agree sets and keeps the computational complexity manageable. Larger keys are generally considered less meaningful, as they often introduce noise or excessive complexity without contributing significantly to the analysis. By restricting our focus to smaller keys, we ensure that the results remain both relevant and interpretable, while also maintaining computational efficiency.

The following table summarizes the outcomes under this restriction:

No.	Dataset	$\lvert\mathcal{A}^s_{\max}\rvert$	$\lvert\mathcal{A}^w_{\max}\rvert$	Linear Size	Sequence	Matched	MostM	MaxM
IX	Hockey.Scoring_M	3	3	10	6	5	5	5
X	Hockey.TeamsPost_M	6	2	11	8	7	7	7
XI	Hockey.SeriesPost_M	4	4	13	6	6	6	6
XII	Hockey.Goalies_M	30	3	37	29	28	28	28
XIII	Hockey.Teams_M	145	3	152	59	57	57	57

These results confirm that the benefit of matching strategies becomes more evident as the number of agree sets grows. When only a few agree sets are present, all strategies yield similar outcomes. However, as dependency complexity increases, matching strategies show better performance in reducing the final Armstrong table size.

Among all techniques evaluated, the *Maximum Matching* strategy tends to produce the smallest table sizes in the most complex scenarios. However, the overall improvements are generally modest, and the computational overhead of computing the matchings should be considered in practice, as matching strategies tend to increase running times by approximately 50%. For lightweight datasets with relatively few dependencies, the *Sequence* approach remains a viable and efficient baseline.

Chapter 9

Informative Sublinear Armstrong Tables for Possible and Certain Keys

As a practical extension of our work, we also explored the construction of *Informative Armstrong tables* for possible and certain keys in the presence of `NULL` values. In real-world relational datasets, the sheer volume and structural complexity often make it challenging for domain experts to validate key constraints effectively. In such scenarios, sublinear, reduced-size tables are especially valuable, as they highlight the necessary constraint semantics without overwhelming users with redundant data.

Informative Armstrong tables are derived from full Armstrong tables that satisfy the same sets of key constraints while violate those not implied. However, instead of preserving every detail of the original table, the informative version strategically retains only the minimal number of tuples needed to reflect the presence or absence of certain and possible keys. This results in a more interpretable and compact form - one that is easier to inspect, analyze, and reason about.

9.1 Algorithm for Constructing Smaller Informative Armstrong Tables with `NULL` Values

The construction of compact Informative Armstrong tables that include `NULL` values follows a process similar to that used for NULL-free datasets, as introduced in Chapter 6, with necessary adaptations to account for both strong and weak agree sets. The algorithm proceeds as follows:

1. **Mine Agree Sets:**
   Mine the set of maximal strong agree sets $\mathcal{A}^s_{\max}$ and maximal weak agree sets $\mathcal{A}^w_{\max}$ from the original dataset.
2. **Construct the Graph Representation:**
   - Each tuple from the original dataset is represented as a vertex in the graph.
   - Edges between vertices are labeled with agree sets that capture shared attribute values:
     - Strong agree sets are directly labeled on edges where two tuples agree on all attributes in the set.
     - Weak agree sets are also assigned to edges, with `NULL` values marked on the corresponding vertices where applicable.
3. **Filter Non-Maximal Agree Sets:**
   Remove all agree sets from the graph that are not elements of $\mathcal{A}^s_{\max}$ or $\mathcal{A}^w_{\max}$. Since these sets are not maximal, they do not contribute essential information and can be excluded from the informative representation.
4. **Prune Isolated Tuples:**
   Remove isolated vertices from the graph - i.e., tuples that do not participate in any maximal strong or weak agree sets. These tuples have no adjacent edges and thus do not contribute to the representation of constraints.

5. **Deduplicate Agree Sets:**
   To further reduce the size of the resulting table, retain only one instance of each maximal agree set - ensuring that every element in $\mathcal{A}^s_{\max} \cup \mathcal{A}^w_{\max}$ is represented at least once in the graph. While doing so, preserve any mandatory `NULL` annotations required by weak agree sets.

6. **Generate the Informative Table:**
   Transform the final, pruned graph back into a relational table. Each vertex becomes a tuple, with attribute values drawn from the corresponding real-world records. The result is a compact, informative Armstrong table.

By retaining only critical tuples and visualizing dependencies through agree sets, the resulting table serves as a concise yet expressive representation of the key-related semantics in real-world relational data - including the complexities introduced by partial information and `NULL` values.

9.2 Strategies for removing duplicates

The strategies introduced in Chapter 6 for generating NULL-free Informative Armstrong tables can be broadly classified into two categories: *edge-oriented approaches* and *vertex-oriented approaches*. Edge-oriented methods - such as *Edge Number*, *Progressive Vertex Expansion*, and *(Weighted) Edge Rank* - prioritize edges based on structural or constraint-related metrics to determine which agree sets should be retained. In contrast, vertex-oriented methods - such as *Weighted Vertex Rank* and *Domination* - evaluate tuples (vertices) to maximize coverage of agree sets.

These strategies are also applicable to Informative Armstrong tables that include `NULL` values. However, our empirical evaluation indicates that the differences in table size among the different strategies are only marginal.

Consequently, rather than exhaustively applying all previously defined strategies, we integrate one method from each category to form two representative approaches, referred to as the *Edge Selection* and *Vertex-Centric* strategies. Building on this foundation, we introduce new techniques that adopt alternative perspectives to more effectively eliminate redundant agree sets and enhance the overall compactness and utility of the tables. These new approaches are called *Agree Set Occurrence Frequency* and *Vertex Metric*.

9.2.1 Edge Selection

As a baseline method, this strategy integrates principles from both the *Edge Number (EN)* and *Edge Rank* approaches introduced in Chapter 6. It aims to minimize redundancy while ensuring full coverage of required agree sets, by prioritizing structurally significant edges. The method proceeds as follows:

1. **Initialize Critical Structures:**
   Begin by identifying and including all *Key Vertices* (see Definition 6.2.1) into the set of *Critical Vertices* (CV) (see Definition 6.2.2). At the same time, mark all agree sets that are already *Validated* (Definition 6.2.3) - that is, those for which at least one corresponding edge exists between vertices in the CV set.

2. **Enumerate Remaining Agree Sets:**
   Construct a list of all agree sets in $\mathcal{A}^s_{\max} \cup \mathcal{A}^w_{\max}$ that remain un-validated. These are the sets that have not yet been represented in the graph and must be addressed to complete the constraint representation.

3. **Iterative Edge Selection:**

   - For each un-validated agree set, identify all graph edges labeled with that set.
   - From this subset, select the edge with the smallest index (according to a fixed ordering) to serve as its representative.
   - Add the two vertices incident to the selected edge to the CV list, provided they are not already present.
   - After processing all agree sets, eliminate any duplicate edges or redundant vertices to ensure minimal graph size.

4. **Finalization:**
   The resulting graph, composed solely of vertices in the CV list and the selected representative edges, serves as the basis for generating the informative Armstrong table.

Similar to the *Progressive Vertex Expansion* (PVE) strategy, this approach ensures that each required agree set is represented at least once. To reduce the total number of vertices, it prioritizes edges with smaller indices, which are more likely to involve low-numbered and thus earlier-processed vertices. This increases the chances of reusing critical vertices. Although closely related to the *Edge Number* (EN) strategy, this version extends the method to handle both strong and weak agree sets in the presence of `NULL` values. The added complexity introduced by `NULL`s is accommodated by marking them appropriately on the relevant vertices during the edge selection process, ensuring correctness with respect to the weak agree set semantics.

9.2.2 Vertex-Centric Strategy

This approach centers on vertex-driven selection, primarily drawing from the *Weighted Vertex Rank* method, with conceptual influences from *Progressive Vertex Expansion*. The key idea is to prioritize edges that are adjacent to *Critical Vertices* (CVs), incrementally expanding the graph by selecting edges that are most likely to connect or extend validated regions of the data graph.

The method leverages vertex connectivity to efficiently validate all agree sets in $\mathcal{A}^s_{\max} \cup \mathcal{A}^w_{\max}$, while aiming to minimize the number of vertices and maintain adherence to both strong and weak agree set semantics, including handling of `NULL` values where applicable.

The procedure is as follows:

```
Initialize the CV list by adding all Key Vertices (Definition 6.2.1)
Mark all agree sets that are already Validated (Definition 6.2.3)
for each agree set A in A^s_max ∪ A^w_max that is not yet Validated do
    if there exists an edge labeled by A that connects two vertices already in CV then
        Mark A as Validated
    else if there exists an edge labeled by A that connects a CV vertex and a non-CV vertex then
        Add the non-CV vertex to the CV list
        Mark A as Validated
    else
        Select edge labeled A adjacent to highest-degree vertex; randomly break ties if any
        Add both vertices of that edge to the CV list
        Mark A as Validated
    end if
end for
```

This strategy incrementally builds a minimal informative graph by extending from already validated subgraphs. By favoring vertices with higher degrees, the algorithm increases the chance of reusing vertices across multiple agree sets, thus minimizing the total number of tuples in the final table.

9.2.3 Agree Set Occurrence Frequency

This strategy is based on the intuition that, when eliminating duplicates, prioritizing agree sets with lower occurrence frequencies simplifies the decision-making process by reducing the number of available edge options. This narrowing of choices can influence subsequent selections and more effectively guide the construction toward a compact final result.

The method proceeds by iterating over agree sets in ascending order of their frequency - that is, the number of times they appear as labels on edges in the graph. Agree sets that appear less frequently are processed earlier, as they offer fewer placement options and are more likely to constrain the overall structure.

Additionally, this strategy introduces and prioritizes *double-labelled edges* - edges that simultaneously satisfy both a strong and a weak agree set. Specifically, an edge is considered double-labelled if it is annotated with a pair of agree sets $X \in \mathcal{A}^s_{\max}$ and $Y \in \mathcal{A}^w_{\max}$ such that $X \subset Y$ and $Y \setminus X \subseteq R \setminus R_s$. These edges are particularly valuable because they enable the validation of two agree sets with a single edge, thereby decreasing the number of vertices required in the final table.

The algorithm follows these steps:

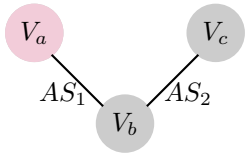
```
Add all Key Vertices to the CV (Critical Vertex) list
Mark all already-Validated agree sets
for each double-labelled edge do
    if either agree set is not yet Validated then
        Add both adjacent vertices to the CV list
        Mark both agree sets as Validated
    end if
end for
Sort the remaining un-Validated agree sets by increasing edge occurrence frequency
for each un-Validated agree set A in sorted order do
    if there exists a labeled edge for A between two vertices in CV then
        Mark A as Validated
    else if there exists a labeled edge for A between one CV vertex and one non-CV vertex then
        Add the non-CV vertex to the CV list
        Mark A as Validated
    else
        Randomly select one labeled edge for A
        Add both adjacent vertices to the CV list
        Mark A as Validated
    end if
end for
```

By validating double-labeled edges first, this method ensures early coverage of valuable agree set pairs, thereby simplifying and streamlining subsequent selection steps. When multiple double-labeled edges correspond to the same agree set pair, duplicates are eliminated to maintain minimality - ensuring that each pair is represented only once in the final graph.

9.2.4 Vertex Metric

As a newly introduced vertex selection strategy, the *Vertex Metric* approach ranks candidate vertices using a composite score, prioritizing those with higher values. The algorithm iteratively includes lower-ranked vertices until all strong and weak agree sets have been validated. This method is particularly effective for minimizing duplication while strategically expanding the set of Critical Vertices (CV).

Definition 9.2.1 (Confirmed and Tentative Agree Sets)**.** An agree set is considered *Confirmed* for a vertex V if it is labeled on an edge connecting V to a Critical Vertex. Conversely, an agree set is considered *Tentative* for V if it is labeled on an edge connecting V to a non-Critical Vertex. △



In the illustration above:

- V_a is a Critical Vertex.
- AS_1 is a *Confirmed* agree set for V_b, as it connects V_b to the Critical Vertex V_a.
- AS_2 is a *Tentative* agree set for V_b, as it connects V_b to the non-Critical Vertex V_c.

The algorithm proceeds as follows:

```
Add Key Vertices to the CV list
Mark Validated agree sets
while un-Validated agree sets remain do
    Compute metric scores for all vertices not in the CV list
    Select vertex with the highest metric value; break ties randomly
    Add selected vertex to the CV list
    Mark all newly validated agree sets
    Update graph structure
end while
```

Compared to the *Vertex-Centric* strategy, which emphasizes local adjacency to Key Vertices, the *Vertex Metric* approach provides a more global and weighted evaluation of candidate vertices. Rather than relying solely on proximity, it incorporates multiple factors to guide the selection process more strategically. Specifically, it considers:

- Coverage potential: The number and type of agree sets a vertex can help validate, with preference given to those that cover distinct (non-duplicate) agree sets.
- Frequency-based prioritization: The occurrence frequency of each agree set within the graph, giving higher priority to those that appear less frequently.
- Double-labelled edge presence: Vertices involved in double-labelled edges - those satisfying both strong and weak agree sets - are favored for their ability to reduce redundancy.
- Component connectivity: The size of the connected component to which the vertex belongs, with larger components preferred to encourage better coverage and minimize graph fragmentation.

Metric Function

The metric value for a candidate vertex is calculated as follows:

$$\text{Metric} = V_1 \cdot \text{Weight}^3 + V_{2a} \cdot 2 \cdot \text{Weight}^2 + V_{2b} \cdot \text{Weight}^2 + V_3 \cdot \text{Weight} + V_4$$

Where:

- O_{AS_i}: Number of occurrences of agree set AS_i in the graph.
- $V_1 = \sum_{i=1}^{n_{1a}} \frac{1}{O_{AS_i}}, n_{1a}$: number of *Confirmed distinct* agree sets
- $V_{2a} = \sum_{i=1}^{n_{2a}} \frac{1}{O_{AS_i}}, n_{2a}$: number of *Tentative distinct* agree sets
- $V_{2b} = \sum_{i=1}^{n_{2b}} \frac{1}{O_{AS_i}}, n_{2b}$: number of *Tentative duplicate* agree sets
- V_3: Number of adjacent double-labelled edges
- V_4: Size of the connected component containing the vertex
- *Weight*: A large constant that emphasizes more important terms

The design prioritizes:

- Confirmed distinct agree sets (most valuable for validation)
- Tentative distinct sets over duplicates, as the former contribute more unique information
- Vertices with strong integration into the graph via double-labelled edges or large connected components

By combining these factors, the scoring mechanism effectively balances coverage and uniqueness, ensuring broad validation across diverse agree sets while prioritizing vertices that contribute unique information. This balance guides the selection process toward compact informative Armstrong tables that reflect real-world relational semantics, including incomplete information represented by `NULL` values.

9.3 Experiments

We conducted a series of experiments using real-world datasets from the Hockey dataset group[1] to evaluate the effectiveness of different strategies for constructing smaller Informative Armstrong Tables with `NULL` values.

The table below summarizes the characteristics of each dataset and the performance of four different reduction strategies:

No.	Source	Size (Cols×Rows)	IncidentV	$\lvert\mathcal{A}^s_{\max}\rvert$	$\lvert\mathcal{A}^w_{\max}\rvert$	EdgeSel	Occur	V-Centric	V-Metric
I	AwardsMisc	6×125	117	1	1	5	3	3	3
II	ScoringSup	4×138	43	2	1	5	5	5	5
III	AwardsPlayers	6×2092	1536	4	2	9	8	7	7
IV	TeamsSC	10×31	13	6	5	11	11	11	11
V	Coaches	14×1813	1783	2	29	23	15	15	15
VI	SeriesPost	13×833	464	21	16	49	43	42	42
VII	TeamsPost	17×928	409	149	15	225	204	204	202
VIII	Goalies	23×4279	4254	7	1679	526	502	502	502

Each column under the method names (*EdgeSel*, *Occur*, *V-Centric*, *V-Metric*) reports the final number of tuples (vertices) in the resulting Informative Armstrong Table for that strategy. The column *IncidentV* denotes the number of vertices (tuples) from the original dataset that are involved in at least one maximal strong or weak agree set - serving as the initial unprocessed graph size.

The results indicate that the *Edge Selection* strategy consistently yields the largest table sizes across datasets. Although it maintains full coverage of all agree sets, it does so with less regard for compactness. On the other hand, while *Vertex Metric* was expected to provide the best reductions due to its composite prioritization strategy, its performance is only marginally better than the simpler *Vertex Centric* and *Occurrence Frequency* methods.

Despite this, all four strategies result in substantial compression when compared to the raw datasets - especially evident when contrasting final table sizes with the number of rows in the original tables. For example, the *AwardsPlayers* dataset, which contains over 2,000 rows, is reduced to just 7-9 tuples in the Armstrong table, depending on the strategy used.

Notably, datasets with high row counts but relatively few attributes (columns), such as *AwardsPlayers*, benefit the most from these reductions. This is because such datasets are more likely to contain redundant or overlapping patterns, which can be captured and represented compactly in Armstrong tables through well-selected agree sets.

In contrast, datasets with very large numbers of unique agree sets - such as *Goalies* with 1679 weak agree sets - pose a greater challenge. Still, even here, the reduction is significant: from over 4000 raw tuples to just over 500 in the final table, demonstrating the scalability and effectiveness of these strategies even under dense dependency conditions.

Overall, while the differences between the strategies may be small in absolute numbers for some datasets, the consistent reduction in size illustrates their utility in generating interpretable, semantically rich, and practically useful Armstrong tables that support constraint verification in real-world relational settings.

[1] Available at https://relational-data.org/dataset/Hockey

Chapter 10

Conclusions and Future Directions

Armstrong tables serve as a vital interpretative and practical tool in relational database design. They bridge the gap between database specialists and domain experts by offering a concrete, example-driven means of specifying, validating, and communicating data constraints. At their core, Armstrong tables represent a minimal yet complete instance of a database that satisfies exactly the constraints specified - adhering to all logical consequences while deliberately violating any constraints not implied by the input set. This makes them particularly useful for semantic verification.

While prior research has largely focused on the theoretical foundations of Armstrong databases - including their existence and structural properties - our work advances the practical dimension of their construction. Specifically, we have investigated how to minimize the size of Armstrong tables to make them more usable in real-world settings. Our focus has been primarily on Armstrong tables for *keys* and *functional dependencies (FDs)*, with additional exploration into general-purpose methods for combining multiple types of constraints.

A significant contribution of this study is the construction of *sublinear-sized* Armstrong tables for keys, especially in practical scenarios where interpretability and computational efficiency are critical. We introduced techniques for reducing redundancy, and systematically pruning unnecessary tuples, all while preserving semantic correctness. Moreover, we extended this approach to generate *Informative Armstrong Tables* - a compact, readable format tailored to highlight key-relevant structure. These were successfully derived from real-world datasets, demonstrating both scalability and clarity.

Our investigation also included Armstrong tables under incomplete information - specifically, schemas with `NULL` values. We incorporated the semantics of *possible keys* and *certain keys*, represented respectively by *strong* and *weak agree sets*. We adapted the Armstrong table construction framework to accommodate these extended semantics, introducing modified graph-based construction algorithms capable of reasoning over both standard and `NULL`-affected data.

Among the optimization strategies evaluated, we proved that the *Disjoint Union* method outperforms the *Direct Product* in reducing table size when combining multiple constraint classes, provided that Armstrong tables exist for those combined classes. Although this method is not applicable to certain rare or non-standard constraint types, it is highly effective in most practical settings.

From a theoretical standpoint, it is worth reiterating that Armstrong tables of *linear size* are always guaranteed to exist for keys, while *sublinear* Armstrong tables may be possible under favorable conditions, though their existence is not guaranteed. When the number of maximal agree sets grows excessively large, the complexity of construction algorithms can become intractable, even exceeding exponential time in worst-case scenarios. In such cases, heuristic and approximation techniques are essential to obtain a solution within reasonable computational bounds, even at the cost of slight sub-optimality.

When handling data with `NULL`s, the search space expands to include both strong and weak agree sets. Strong agree sets can be handled analogously to those in the standard, `NULL`-free setting. However, weak agree sets introduce additional complexity, requiring integration with explicit `NULL` annotations and specialized validation logic. To maintain compactness in such enriched settings, we evaluated a range of heuristics - such as graph-based

matching, vertex scoring, and frequency-driven selection.

In summary, this work contributes both theoretical insights and practical methodologies for constructing small, expressive, and constraint-faithful Armstrong tables. By extending the framework to handle mixed constraints and incomplete data, and by optimizing for real-world usability, we take a step closer to making Armstrong tables a viable tool for everyday use in data quality assurance, schema design, and automated reasoning in relational databases.

10.1 Discussion and Reflection on Research Aims

The primary aim of this thesis was to investigate practical approaches for constructing compact and semantically meaningful Armstrong tables for relational database constraints. This objective has been substantially achieved through the development and evaluation of construction, optimization, and interpretation techniques for Armstrong tables involving keys, functional dependencies, and incomplete information.

The first objective was to examine the theoretical foundations of Armstrong databases and their role in representing relational constraints. This was addressed through the study of Armstrong relations, various classes of dependencies, together with their significance in semantic verification and schema analysis[2, 4, 5, 6, 7, 9, 10]. The thesis established the theoretical basis required for later construction and optimization methods.

The second objective involved investigating existing Armstrong table construction techniques and identifying limitations related to scalability and practical usability[2, 7, 10, 12, 13]. Through this analysis, it became evident that many traditional approaches generate excessively large Armstrong tables that are difficult to interpret. These limitations motivated the focus on compactness and semantic clarity throughout the research.

The third objective focused on optimization strategies for combining multiple classes of constraints while minimizing the resulting table size. This objective was achieved by demonstrating that the *Disjoint Union* can accomplish the same purpose as the *Direct Product*[13] while producing substantially smaller Armstrong tables in most practical scenarios involving mixed constraint classes.

The fourth objective concentrated on the construction of smaller Armstrong tables for single classes of constraints, particularly for keys and functional dependencies[2, 4, 11]. To address this, the thesis developed graph-based construction methods and heuristic techniques capable of generating sublinear-sized Armstrong tables .

The fifth objective was to construct sublinear informative Armstrong tables that emphasize readability and semantic clarity while preserving correctness[31]. This objective was achieved through experimental constructions based on real-world datasets, showing that compact Armstrong tables can be generated while maintaining the required constraint representations and reducing the size of the resulting examples.

The final objective was to extend Armstrong table construction to databases containing incomplete information represented by `NULL` values[14, 26, 27, 28]. This objective was achieved through the optimization of assigning strong and weak agree sets, together with the implementation of matching strategies[32] to generate sublinear-sized Armstrong tables. These results demonstrate that Armstrong-style reasoning can be generalized beyond classical relational settings.

Nevertheless, several limitations remain. First, the existence of sublinear Armstrong tables cannot always be guaranteed. Moreover, for highly compact synthetic datasets such as those shown in Table 5.3, heuristic-guided approaches may produce Armstrong tables that are substantially larger than the optimum. A further limitation lies in the reliance on heuristic optimization techniques. While the proposed heuristics were effective in most experimental scenarios, they do not guarantee globally optimal solutions.

In summary, the research objectives defined in Chapter 1 were successfully achieved. The thesis demonstrates that compact and semantically meaningful Armstrong tables are both feasible and practically valuable for representing and analyzing relational constraints.

10.2 Future Research Directions

Several promising research directions emerge from this work. First, it may be worthwhile to investigate the theoretical conditions under which sublinear-sized Armstrong tables can be constructed. A deeper understanding of the structural characteristics of agree sets that permit compact representations may lead to tighter complexity bounds and more efficient construction algorithms.

Second, more sophisticated heuristic and approximation methods could be explored for scenarios in which exact construction becomes computationally infeasible. In particular, machine learning techniques and adaptive optimization strategies may offer practical mechanisms for dynamically selecting tuple-generation and pruning procedures.

Third, future work could extend compact Armstrong table construction beyond keys and functional dependencies to richer classes of constraints. Such extensions would significantly broaden the applicability of Armstrong tables in modern database design, data integration, and data quality management.

Finally, integrating Armstrong table construction into practical database engineering and data quality tools represents an important applied direction. Interactive systems capable of generating compact, semantically informative Armstrong tables automatically could substantially improve dependency validation, schema debugging, and explainable data governance in real-world environments.

Appendix A: Glossary

Symbol	Meaning	Definition (Page)
Σ^*	Closure of constraint set Σ	Def. 1.1.8 (p. 3)
$ag(r)$	Agree sets of relation r	Def. 2.2.1 (p. 12)
$\mathcal{T}_r(\mathcal{F})$	Transversal for a family of sets $\mathcal{F}$	Def. 2.2.4 (p. 13)
$\mathrm{cl}(A)$	Closure of attribute set A	Def. 2.2.5 (p. 13)
$\mathrm{cl}_\Sigma(X)$	Closure of attribute set $X \subseteq R$ w.r.t. constraint set Σ	Def. 2.2.5 (p. 13)
$\mathrm{CL}_\Sigma(R)$	Set of all closed subsets of schema R w.r.t. constraint set Σ	Def. 2.2.6 (p. 14)
$MAX_\Sigma(R)$	Set of all maximal sets over attributes in R under Σ	Def. 2.2.8 (p. 14)
$GEN_\Sigma(R)$	Minimal set of closed sets whose intersections generate all of $\mathrm{CL}_\Sigma(R)$	Def. 2.2.9 (p. 15)
$p\langle X\rangle$	Possible key	Def. 7.1.2 (p. 63)
$c\langle X\rangle$	Certain key	Def. 7.1.2 (p. 63)
$\mathcal{AG}^s(r)$	Strong agree sets of relation r	Def. 7.1.3 (p. 63)
$\mathcal{AG}^w(r)$	Weak agree sets of relation r	Def. 7.1.3 (p. 63)
$\neg p\langle X\rangle$	Strong anti-key	Def. 7.1.4 (p. 63)
$\neg c\langle X\rangle$	Weak anti-key	Def. 7.1.4 (p. 63)
$\mathcal{A}^s_{\max}$	Set of all maximal strong anti-keys	Def. 7.1.4 (p. 63)
$\mathcal{A}^w_{\max}$	Set of all maximal weak anti-keys	Def. 7.1.4 (p. 63)

Bibliography

[1] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[2] Heikki Mannila and Kari-Jouko Räihä. Design by example: An application of armstrong relations. *J. Comput. Syst. Sci.*, 33(2):126–141, 1986.

[3] William Ward Armstrong. Dependency structures of data base relationships. In Jack L. Rosenfeld, editor, *Information Processing, Proceedings of the 6th IFIP Congress 1974, Stockholm, Sweden, August 5-10, 1974*, pages 580–583. North-Holland, 1974.

[4] Ronald Fagin. Armstrong databases. *7th IBM Symp. on Mathematical Foundations of Computer Science*, 1982.

[5] Heikki Mannila and Kari-Jouko Räihä. *Design of Relational Databases*. Addison-Wesley, 1992.

[6] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[7] Ronald Fagin and Moshe Y. Vardi. Armstrong databases for functional and inclusion dependencies. *Inf. Process. Lett.*, 16(1):13–19, 1983.

[8] Xiaolei Qian and Douglas R. Smith. Integrity constraint reformulation for efficient validation. In Peter M. Stocker, William Kent, and Peter Hammersley, editors, *VLDB'87, Proceedings of 13th International Conference on Very Large Data Bases, September 1-4, 1987, Brighton, England*, pages 417–425. Morgan Kaufmann, 1987.

[9] Henning Köhler and Sebastian Link. Inclusion dependencies and their interaction with functional dependencies in SQL. *J. Comput. Syst. Sci.*, 85:104–131, 2017.

[10] Stavros S. Cosmadakis and Paris C. Kanellakis. Functional and inclusion dependencies: A graph theoretic approach. In Daniel J. Rosenkrantz and Ronald Fagin, editors, *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, April 2-4, 1984, Waterloo, Ontario, Canada*, pages 29–37. ACM, 1984.

[11] Van Bao Tran Le, Sebastian Link, and Mozhgan Memari. Discovery of keys from SQL tables. In Sang-goo Lee, Zhiyong Peng, Xiaofang Zhou, Yang-Sae Moon, Rainer Unland, and Jaesoo Yoo, editors, *Database Systems for Advanced Applications - 17th International Conference, DASFAA 2012, Busan, South Korea, April 15-19, 2012, Proceedings, Part I*, volume 7238 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2012.

[12] Catriel Beeri, Martin Dowd, Ronald Fagin, and Richard Statman. On the structure of armstrong relations for functional dependencies. *J. ACM*, 31(1):30–46, 1984.

[13] Ronald Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

[14] Henning Köhler, Uwe Leck, Sebastian Link, and Xiaofang Zhou. Possible and certain keys for SQL. *VLDB J.*, 25(4):571–596, 2016.

[15] Ziheng Wei and Sebastian Link. Dataprof: Semantic profiling for iterative data cleansing and business rule acquisition. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1793–1796. ACM, 2018.

[16] Johann Birnick, Thomas Bläsius, Tobias Friedrich, Felix Naumann, Thorsten Papenbrock, and Martin Schirneck. Hitting set enumeration with partial information for unique column combination discovery. *Proc. VLDB Endow.*, 13(11):2270–2283, 2020.

[17] Thorsten Papenbrock and Felix Naumann. A hybrid approach to functional dependency discovery. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 821–833. ACM, 2016.

[18] Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius, Martin Schirneck, and Felix Naumann. Discovering functional dependencies through hitting set enumeration. *Proc. ACM Manag. Data*, 2(1):43:1–43:24, 2024.

[19] Henning Koehler and Sebastian Link. Orthogonal keys high precision and recall for mining database keys from inconsistent and incomplete relations. *IEEE Trans. Knowl. Data Eng.*, 37(11):6550–6561, 2025.

[20] Hemant Saxena, Lukasz Golab, and Ihab F. Ilyas. Distributed dependency discovery. *CoRR*, abs/1903.05228, 2019.

[21] E. F. Codd. Further normalization of the data base relational model. *Research Report / RJ / IBM / San Jose, California*, RJ909, 1971.

[22] Philip A. Bernstein. Synthesizing third normal form relations from functional dependencies. *ACM Trans. Database Syst.*, 1(4):277–298, 1976.

[23] Henning Köhler and Sebastian Link. SQL schema design: foundations, normal forms, and normalization. *Inf. Syst.*, 76:88–113, 2018.

[24] Erhard Rahm and Hong Hai Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[25] Xu Chu, Ihab F. Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In Fatma Özcan, Georgia Koutrika, and Sam Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 2201–2206. ACM, 2016.

[26] Ron van der Meyden. Logical approaches to incomplete information: A survey. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems (the book grow out of the Dagstuhl Seminar 9529: Role of Logics in Information Systems, 1995)*, pages 307–356. Kluwer, 1998.

[27] Bernhard Thalheim. On semantic issues connected with keys in relational databases permitting null values. *J. Inf. Process. Cybern.*, 25(1/2):11–20, 1989.

[28] Mark Levene and George Loizou. Axiomatisation of functional dependencies in incomplete relations. *Theor. Comput. Sci.*, 206(1-2):283–300, 1998.

[29] Y. Edmund Lien. On the equivalence of database models. *J. ACM*, 29(2):333–362, 1982.

[30] Ashok K. Chandra and Moshe Y. Vardi. The implication problem for functional and inclusion dependencies is undecidable. *SIAM J. Comput.*, 14(3):671–677, 1985.

[31] Fabien De Marchi and Jean-Marc Petit. Semantic sampling of existing databases through informative armstrong databases. *Inf. Syst.*, 32(3):446–457, 2007.

[32] John E. Hopcroft and Richard M. Karp. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs. *SIAM J. Comput.*, 2(4):225–231, 1973.