Application of Graph Traversal Algorithms on Functional Dependency Hypergraphs for Automated Candidate Key Identification and Database Normalization

Billy Ontoseno Irawan
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
13524121@std.stei.itb.ac.id, billy.ontoseno@gmail.com

***Abstract*—Candidate key identification and database normalization are critical processes in relational database design that rely heavily on the analysis of functional dependencies (FDs). Traditional graph-based representations often struggle to capture the inherent many-to-many relationships present in functional dependencies where multiple attributes collectively determine one or more dependent attributes. This paper proposes a hypergraph-based approach for modeling functional dependencies and automating candidate key identification and database normalization. In the proposed model, attributes are represented as vertices while functional dependencies are represented as directed hyperedges that connect sets of determinant attributes to their corresponding dependent attributes**

***Keywords*—graph traversal, functional dependency, hypergraph, candidate key, database normalization**

I. Introduction

Modern relational database design relies heavily on the principles of normalization and functional dependency analysis to ensure data consistency, minimize redundancy, and prevent update anomalies. Since the introduction of the relational model[1], the identification of candidate keys and the decomposition of relations into higher normal forms have remained fundamental tasks in database theory. However, as schema complexity increases in real-world applications, deriving candidate keys and performing normalization becomes both error-prone and computationally intensive, motivating the need for automated and graph-theoretic approaches.

Functional dependencies (FDs) naturally induce structural relationships among attributes in a relation schema which can be represented using graph-based or hypergraph-based models. In particular, hypergraphs provide a more expressive representation than traditional directed graphs because a single functional dependency can relate multiple attributes on the left-hand side to a set of attributes on the right-hand side. This representation aligns well with the theoretical foundations of relational database decomposition and dependency preservation[2]. By modeling schemas as functional dependency hypergraphs, it becomes possible to utilize graph traversal algorithms and closure computation strategies to systematically explore attribute reachability and infer candidate keys.

Candidate key identification is fundamentally equivalent to finding minimal attribute sets whose closure under a given set of functional dependencies includes all attributes in the relation. Traditional algorithms for computing attribute closures are iterative and can be reframed as traversal problems over dependency structures. By treating attributes as nodes and functional dependencies as hyperedges, graph traversal techniques can be adapted to efficiently explore dependency propagation paths. This perspective enhances computational efficiency and also provides a unified framework for normalization processes where decomposition decisions can be guided by structural properties of the dependency hypergraph.

Recent advances in database theory and graph algorithms have further supported the integration of formal graph traversal techniques into schema analysis. Graph-based approaches enable scalable analysis of large and complex schemas, particularly in automated database design systems and schema optimization tools. Consequently, the application of graph traversal algorithms to functional dependency hypergraphs presents a promising direction for automating candidate key discovery and supporting systematic database normalization.

II. Theoretical Framework

A. Hypergraph

A hypergraph H is a generalization of a graph in which each edge (hyperedge) may connect any number of vertices. Formally,

$$H = (V, E) \tag{1}$$

where V is a finite set of vertices and E is a family of nonempty subsets of V called hyperedges. The order of H is $|V|$, and the size of H can be defined as $\sum_{e \in E} |e|$ (the sum of cardinalities of its edges). An undirected hypergraph has no direction on edges; each $e \in E$ is simply a subset of vertices. A hypergraph is k-uniform if every edge has size k; in particular,

a 2-uniform hypergraph is an ordinary graph. The rank of H, denoted $r(H)$, is the maximum $|e|$ among its hyperedges (and the co-rank is the minimum $|e|$). The incidence matrix of H is an $|V| \times |E|$ 0–1 matrix A where $A_{v,e} = 1$ if vertex v is in hyperedge e and 0 otherwise [3].

Directed hypergraphs extend this by orienting each hyperedge: a directed hypergraph is a pair

$$H = (V, \overrightarrow{E}) \tag{2}$$

where each directed hyperarc $e \in \overrightarrow{E}$ is an ordered pair (T, H) with $T \subseteq V$ the tail (source set) and $H \subseteq V$ the head (target set). Thus a directed hyperarc e connects all vertices in T jointly to those in H. Every ordinary directed graph (edges of size 1 tail and 1 head) is a special case. Many-to-many relationships (like FDs) are naturally modeled as such hyperarcs [4].

One can form the dual hypergraph H^* of H by swapping vertices and edges: the dual H^* has vertex set = original edge set E, and for each original vertex $v \in V$ there is a hyperedge in H^* consisting of all edges in E that contain v. Equivalently,

$$H^* = (E, \{e_v : v \in V\}) \tag{3}$$

where $e_v = \{f \in E : v \in f\}$. The notions of connectedness, path, and cut generalize to hypergraphs, though connectivity tests and traversal require care since reaching all vertices in an edge is more complex than single-edge adjacency.

For example, consider a relation schema $R(A, B, C, D)$ with FDs $\{A \to BC, D \to A\}$. We model the FD set as a directed hypergraph: vertices $= \{A, B, C, D\}$ and hyperedges $= \{(\{A\}, \{B, C\}), (\{D\}, \{A\})\}$. So there is a hyperarc with tail A and head $\{B, C\}$, and one with tail D and head A. The dual hypergraph has vertices = these two hyperedges, etc. The rank of this hypergraph is $\max(|\{A\}|, |\{D\}|)$ for tails or $(|\{B, C\}|, |\{A\}|)$ for heads; here rank $= 2$. The hypergraph's size is $|\{A\}| + |\{B, C\}| + |\{D\}| + |\{A\}| = 1 + 2 + 1 + 1 = 5$. This structure captures the FD constraints; for any instance of R, knowledge of A determines B, C and knowledge of D determines A, B, C.

When modeling FDs one typically assumes no empty edges or self-loops. A simple hypergraph disallows repeated identical edges. Subhypergraphs (induced by removing vertices/edges) and merging/splitting edges are analogous to graph operations, though many problems (like vertex cover) become NP-hard in hypergraphs for general edge sizes.

B. Graph Traversal Algorithms

Graph traversal algorithms are fundamental techniques used to systematically explore and search the vertices and edges of a graph. They play a critical role in artificial intelligence, network routing, pathfinding, social network analysis, and optimization problems. Traversal methods can be broadly classified into uninformed and informed search algorithms, as well as static and dynamic graph traversal approaches [5].

Two of the most widely used graph traversal algorithms are Depth-First Search (DFS) and Breadth-First Search (BFS). DFS explores a graph by moving as far as possible along a branch before backtracking. It is commonly implemented using recursion or a stack data structure. DFS is particularly useful for tasks such as cycle detection, topological sorting, and solving maze-like problems. The time complexity of DFS is $O(V + E)$, where V represents the number of vertices and E represents the number of edges in the graph.

In contrast, BFS traverses a graph level by level, visiting all neighboring vertices before moving to the next depth level. BFS is typically implemented using a queue data structure. One of its key advantages is its ability to find the shortest path in an unweighted graph. Similar to DFS, BFS operates with a time complexity of $O(V + E)$, making it efficient for large-scale graph exploration.

The choice between DFS and BFS depends on the specific application requirements. DFS generally requires less memory because it stores only the current path, while BFS may consume more memory as it stores all vertices at the current level. However, BFS is often preferred when the shortest path or minimum number of edges between two vertices is required [6].

C. Functional Dependencies and Attribute Closure

Let $R(U)$ be a relation schema, and let $X, Y \subseteq U$. A functional dependency $X \to Y$ holds on R when every pair of legal tuples that agrees on X also agrees on Y. Formally,

$$t_1[X] = t_2[X] \Rightarrow t_1[Y] = t_2[Y]. \tag{4}$$

The attributes in X form the determinant, while the attributes in Y are functionally dependent on X [2], [7], [8].

The closure of X under a dependency set F, denoted X_F^+, is the set of all attributes functionally determined by X. Closure is computed from an initial set $C_0 = X$ using the recurrence

$$C_{k+1} = C_k \cup \bigcup_{(T,H) \in E_F,\, T \subseteq C_k} H. \tag{5}$$

The process stops at a fixed point where $C_{k+1} = C_k$. Because the closure can grow at most to U, the procedure terminates after finitely many attribute insertions.

Algorithm 1 Attribute Closure over Directed Hyperedges

Require: Attribute set X, hyperedge set E_F
Ensure: Closure X_F^+

$C \leftarrow X$
$changed \leftarrow true$
while $changed$ **do**
  $changed \leftarrow false$
  for each $(T, H) \in E_F$ **do**
    if $T \subseteq C$ **then**
      $N \leftarrow H \setminus C$
      if $N \neq \varnothing$ **then**
        $C \leftarrow C \cup N$
        $changed \leftarrow true$
      end if
    end if
  end for
end while
return C

D. Superkeys and Candidate Keys

A set of attributes $K \subseteq U$ is a *superkey* of $R(U)$ when it functionally determines every attribute in the schema:

$$K_F^+ = U. \tag{6}$$

A *candidate key* is a minimal superkey. Therefore, K is a candidate key if it satisfies both completeness and minimality:

$$K_F^+ = U, \tag{7}$$

$$(K \setminus \{a\})_F^+ \neq U, \quad \forall a \in K. \tag{8}$$

One candidate key may be selected as the primary key, while the remaining candidate keys are alternate keys. An attribute that belongs to at least one candidate key is called a prime attribute.

A useful reduction follows from the right-hand sides of the dependencies. Let

$$M = U - \mathrm{RHS}(F), \tag{9}$$

where $\mathrm{RHS}(F)$ is the union of all attributes appearing on dependency right-hand sides. Every attribute in M must occur in every candidate key because no dependency can derive it. Candidate-key search can begin with M and add subsets of $U - M$ in nondecreasing cardinality. Once a candidate key is found, all of its strict supersets can be pruned because they cannot be minimal.

Algorithm 2 Candidate-Key Enumeration by Closure

Require: Schema attributes U, dependency hyperedges E_F
Ensure: Set $\mathcal{K}$ of candidate keys

$M \leftarrow U - \mathrm{RHS}(F)$
$O \leftarrow U - M$
$\mathcal{K} \leftarrow \varnothing$
for each subset $S \subseteq O$ in nondecreasing $|S|$ **do**
  $K \leftarrow M \cup S$
  if no $K' \in \mathcal{K}$ satisfies $K' \subset K$ **then**
    if $Closure(K, E_F) = U$ **then**
      $\mathcal{K} \leftarrow \mathcal{K} \cup \{K\}$
    end if
  end if
end for
return $\mathcal{K}$

The closure test is polynomial in the schema and dependency sizes, although enumerating all candidate keys can require exponential search because a schema may contain exponentially many minimal keys.

E. Database Normalization

Normalization decomposes a relation into smaller relations to reduce redundancy and avoid insertion, deletion, and update anomalies. The analysis in this paper focuses on 2NF, 3NF, and BCNF [7], [8].

A relation is in First Normal Form (1NF) when every attribute value is atomic. Assuming 1NF, a relation is in Second Normal Form (2NF) when no non-prime attribute is partially dependent on a proper subset of a candidate key. For a composite candidate key K, a dependency $X \to A$ is a 2NF violation when $X \subset K$, $X \neq K$, and A is non-prime.

A relation is in Third Normal Form (3NF) when, for every nontrivial dependency $X \to A$ in F^+, at least one of the following holds:

1) X is a superkey; or
2) A is a prime attribute.

BCNF is stricter. A relation is in BCNF when the determinant X is a superkey for every nontrivial dependency $X \to A$.

A useful decomposition should preserve two properties. A decomposition $\mathcal{D} = \{R_1, \ldots, R_m\}$ has the *lossless-join* property when joining the projected relations reconstructs exactly the original legal relation. It is *dependency preserving* when the original dependencies can be enforced by checking dependencies within individual decomposed relations without computing their join.

The standard 3NF synthesis approach creates a relation for each dependency in a minimal cover, removes redundant contained relations, and adds a relation containing a candidate key when none of the generated relations already contains one [8]. Some resulting relations may also satisfy BCNF, depending on their projected dependencies.

III. Formal Problem Formulation

A. Input

The input consists of a relation schema and a set of functional dependencies:

$$R(U), \qquad U = \{A_1, A_2, \ldots, A_n\}, \tag{10}$$

$$F = \{X_1 \to Y_1, \ldots, X_m \to Y_m\}, \tag{11}$$

where $X_i, Y_i \subseteq U$ and $X_i, Y_i \neq \varnothing$.

The dependency set is transformed into a directed hypergraph

$$H_F = (U, E_F), \qquad E_F = \{(X_i, Y_i) \mid 1 \leq i \leq m\}. \tag{12}$$

Each attribute is a vertex, each determinant is a tail set, and each dependent is a head set.

B. Direction-Aware Reachability Problem

For a starting attribute set $S \subseteq U$, determine the set of reachable vertices under the activation rule

$$(T, H) \text{ is traversable } \iff T \subseteq V_{\text{visited}}. \tag{13}$$

When a hyperedge becomes traversable, every attribute in H is inserted into the visited set. The desired fixed point is identical to the attribute closure:

$$Reach_{H_F}(S) = S_F^+. \tag{14}$$

The traversal may additionally return the order in which attributes are discovered and the sequence of hyperedges used during propagation.

C. Candidate-Key Identification Problem

Determine every set $K \subseteq U$ satisfying Eqs. (7) and (8). Equivalently, the candidate-key problem is the search for all inclusion-minimal start sets whose directed-hypergraph reachability covers every schema vertex.

The mandatory seed set is

$$M = U - \bigcup_{i=1}^{m} Y_i. \tag{15}$$

Every valid candidate key must contain M. The remaining search space is 2^{U-M}, with closure used as the feasibility test and subset inclusion used as the minimality test.

D. Normalization Analysis Problem

Given the candidate keys and functional dependencies, determine the highest normal form satisfied by R. The analysis requires:

1) determining prime and non-prime attributes from the candidate-key set;
2) testing partial dependencies for 2NF;
3) testing the superkey-or-prime condition for 3NF;
4) testing whether every determinant is a superkey for BCNF; and
5) constructing a lossless and preferably dependency-preserving decomposition when violations are present.

TABLE I
Directed Hyperedges in the Program Instance

Hyperedge	Tail	Head
e_1	$\{A, B\}$	$\{C\}$
e_2	$\{A\}$	$\{G\}$
e_3	$\{C\}$	$\{D\}$
e_4	$\{G, H\}$	$\{I\}$
e_5	$\{D, E\}$	$\{F\}$

The output is a tuple

$$\mathcal{O} = (\mathcal{K}, N, \mathcal{D}), \tag{16}$$

where $\mathcal{K}$ is the candidate-key set, N is the highest normal form satisfied by the original relation, and $\mathcal{D}$ is the proposed decomposition.

E. Program Instance

The implemented example defines the attribute set

$$U = \{A, B, C, D, E, F, G, H, I\} \tag{17}$$

and the functional dependencies

$$F = \{AB \to C,\ A \to G,\ C \to D,\ GH \to I,\ DE \to F\}. \tag{18}$$

The corresponding directed hyperedges are listed in Table I.

From Eq. (15), the attributes absent from all right-hand sides are

$$M = \{A, B, E, H\}. \tag{19}$$

Its closure is computed as follows:

$$\begin{aligned} \{A, B, E, H\} &\xrightarrow{AB \to C} \{A, B, C, E, H\} \\ &\xrightarrow{A \to G} \{A, B, C, E, G, H\} \\ &\xrightarrow{C \to D} \{A, B, C, D, E, G, H\} \\ &\xrightarrow{GH \to I} \{A, B, C, D, E, G, H, I\} \\ &\xrightarrow{DE \to F} U. \end{aligned}$$

Therefore, M is a superkey. Every member of M is mandatory because none appears on a right-hand side, so removing any member makes it impossible to derive that attribute. Consequently,

$$\mathcal{K} = \{\{A, B, E, H\}\} \tag{20}$$

is the unique candidate-key set, and the prime attributes are A, B, E, and H.

The dependency $A \to G$ is a partial dependency from a proper subset of the composite candidate key to a non-prime attribute, so the original relation violates 2NF. It also violates 3NF and BCNF because the determinants AB, A, C, GH, and DE are not superkeys of the original relation, while their dependent attributes are non-prime.

A dependency-preserving 3NF synthesis produces

$$\begin{aligned} \mathcal{D} = \{&R_1(A, B, C),\ R_2(A, G),\ R_3(C, D), \\ &R_4(G, H, I),\ R_5(D, E, F),\ R_6(A, B, E, H)\}. \end{aligned} \tag{21}$$

The relation R_6 is added because none of the first five relations contains the candidate key. In each of R_1 through R_5, the determinant of the projected nontrivial dependency is a key of that relation. Relation R_6 has no projected nontrivial dependency from the given set. Hence, every relation in Eq. (21) satisfies BCNF with respect to the projected dependencies.

IV. Implementation

A. Hypergraph Representation

The Python prototype defines a class named `FunctionalDependencyHypergraph`. Its representation follows the formal model $H_F = (U, E_F)$:

- `nodes` is a set containing all attribute vertices;
- `hyperedges` is a list of tuples `(tail, head)`, where both elements are sets;
- `node_to_hyperedges` maps each attribute to the indices of all incident hyperedges;
- `tail_nodes` records attributes appearing in determinants; and
- `head_nodes` records attributes appearing in dependents.

The `add_fd` method inserts every determinant and dependent attribute into the vertex set, stores the directed hyperedge, and updates the incidence mapping for every attribute in the union of its tail and head. The mapping supports localized retrieval of relevant dependencies during BFS.

B. Directional BFS

The function `bfs_directional_fd` receives a hypergraph and a starting attribute set. It validates that all start attributes exist in the graph, initializes the visited set and queue, and repeatedly processes incident hyperedges. For each hyperedge (T, H), it computes

$$|T \cap V_{\text{visited}}| \tag{22}$$

and compares the value with $|T|$. The dependency is applied only when the two values are equal. Newly discovered head attributes are appended to the traversal order and the queue, while applied hyperedges are recorded in a dependency chain.

This implementation directly enforces the semantic requirement that every determinant attribute must be known. For example, encountering A alone cannot activate $AB \to C$. The hyperedge becomes traversable only after both A and B belong to the visited set.

The second traversal function, `bfs_prerequisite_order`, performs the same logical test through the set expression

$$T \subseteq V_{\text{visited}}. \tag{23}$$

It emphasizes that determinant attributes precede newly discovered dependent attributes. In the current code, both traversal functions produce the same reachable set for a given start set, although their printed diagnostic messages emphasize different interpretations.

C. Closure Computation

The function `compute_closure` implements Algorithm 1 directly. It initializes `closure` with the requested attribute set and repeatedly scans the stored hyperedges. If the tail is a subset of the closure, the function adds previously unknown head attributes. The scan terminates when one complete pass adds no attribute.

Unlike the BFS functions, closure computation scans all hyperedges during each iteration instead of only the hyperedges incident to the currently dequeued vertex. This makes it a direct implementation of the standard database closure algorithm and provides the clearest basis for superkey testing.

For the program's existing call `compute_closure(fd_hg, {"A"})`, the result is

$$\{A\}_F^+ = \{A, G\}. \tag{24}$$

The dependency $A \to G$ is applicable, while $AB \to C$ requires B, $GH \to I$ requires H, and the remaining dependencies require attributes that cannot be reached from A alone. Therefore, A is not a superkey.

For candidate-key verification, the same implemented function can be called with $\{A, B, E, H\}$, yielding all attributes in U. The minimality argument then follows from the mandatory-attribute property in Eq. (15).

D. Program Execution Flow

The `main` block executes the following sequence:

1) create an empty functional-dependency hypergraph;
2) insert the five dependencies in Eq. (18);
3) print every stored hyperedge;
4) run directional BFS from $\{A, C\}$;
5) run prerequisite-order BFS from $\{A, C\}$; and
6) compute the closure of $\{A\}$.

Starting from $\{A, C\}$, the traversals can apply $A \to G$ and $C \to D$. They cannot apply $AB \to C$, $GH \to I$, or $DE \to F$ because B, H, and E are missing. Ignoring nondeterministic ordering caused by Python set iteration, the reached attribute set is therefore

$$\{A, C, D, G\}. \tag{25}$$

This result illustrates why hyperedge activation must use complete-tail membership rather than ordinary adjacency.

E. Complexity

Let $n = |U|$, $m = |F|$, and

$$L = \sum_{(T,H) \in E_F} (|T| + |H|) \tag{26}$$

be the total hyperedge incidence size.

Constructing the hypergraph requires $O(L)$ insertions. The implemented closure function performs repeated full scans. At most n successful attribute-growth stages can occur, and each scan checks dependency tails, giving a conservative complexity of $O(nL)$. With per-hyperedge counters and reverse incidence lists restricted to tail membership, closure

TABLE II
ANALYSIS RESULTS FOR THE PROGRAM DEPENDENCY SET

Input	Result
BFS start $\{A, C\}$	Reached set $\{A, C, D, G\}$ using $A \to G$ and $C \to D$
Closure of $\{A\}$	$\{A, G\}$; therefore A is not a superkey
Closure of $\{A, B, E, H\}$	U; therefore the set is a superkey
Minimality test	Every member is mandatory, so $\{A, B, E, H\}$ is a candidate key
Normal-form test	Original schema violates 2NF, 3NF, and BCNF
Decomposition	Six relations in Eq. (21), each satisfying BCNF under projected dependencies

propagation can be improved to $O(n+L)$ for a single starting set.

The implemented BFS uses incidence mappings, but it may reconsider an unapplied hyperedge from several incident vertices and computes tail intersections during each check. Its practical cost depends on tail sizes and incidence frequency; a conservative bound is $O(Lr)$, where r is the maximum tail cardinality, with additional repeated checks possible before an edge becomes applicable. The memory requirement is $O(n + L)$.

Automatic enumeration of candidate keys has worst-case exponential behavior because it explores subsets of optional attributes. If $q = |U - M|$, the search may require up to 2^q closure tests, producing a bound of $O(2^q nL)$ with the current closure routine. Minimality pruning and mandatory attributes substantially reduce the search space in many practical schemas.

V. RESULTS AND DISCUSSION

A. Traversal and Closure Results

Table II summarizes the meaningful results of the implemented examples and the additional candidate-key verification using the existing closure function.

The comparison between $\{A\}_F^+$ and $\{A, B, E, H\}_F^+$ demonstrates the importance of composite start sets. A vertex-centered graph interpretation might suggest that starting from A should eventually reach any edge touching A, but the hypergraph semantics correctly block $AB \to C$ until B is present.

B. Relationship Between Traversal and Candidate Keys

The visited set returned by direction-aware traversal and the fixed point returned by closure represent the same mathematical reachability relation when both procedures eventually reconsider every hyperedge whose tail may become satisfied. Closure is especially suitable for key verification because only the final reachable set is required. BFS additionally provides an explanatory discovery order and dependency chain, which can help users understand why an attribute is reachable.

A candidate key can therefore be interpreted as a minimal set of initial vertices that activates enough hyperedges to reach all vertices. This interpretation is useful pedagogically because it turns a symbolic closure computation into a constrained graph traversal problem.

C. Normalization Interpretation

The unique candidate key is composite, and $A \to G$ depends on only one part of that key. This immediately establishes a 2NF violation. The remaining dependencies reveal transitive structure. For example, the key determines C through $AB \to C$, after which $C \to D$ determines D. Similarly, the key determines G, and together with H, the dependency $GH \to I$ determines I.

The decomposition in Eq. (21) separates these determinant-dependent groups. Each dependency can be enforced locally in one relation, and the added key relation $R_6(A, B, E, H)$ supports the lossless-join guarantee of 3NF synthesis. The result illustrates how hypergraph reachability provides the information needed for normalization, even though decomposition itself requires additional synthesis rules beyond BFS.

D. Implementation Scope and Limitations

The current program implements hypergraph construction, two traversal variants, and attribute closure. It does not yet contain a function that enumerates all candidate keys, computes a minimal cover, determines normal forms automatically, or generates decomposed schemas. The candidate key and decomposition in this paper are derived by applying the formal procedures to the program's dependency data and by reusing the implemented closure operation for verification.

The current implementation also uses Python sets, whose iteration order should not be treated as a stable traversal order. The reached set is deterministic, but the printed order may differ between runs or environments. In addition, `bfs_directional_fd` declares a `tail_progress` dictionary without using it; a counter-based implementation could avoid repeated full tail-intersection checks. Finally, the code accepts arbitrary hashable Python objects as attributes and performs no parsing or validation of textual functional-dependency notation.

These limitations define a clear extension path: add candidate-key enumeration using Algorithm 2, minimal-cover computation, 3NF synthesis, BCNF decomposition, and deterministic output ordering.

VI. CONCLUSION

This paper has modeled functional dependencies as directed hyperedges and interpreted attribute closure as direction-aware reachability. The model preserves composite determinants because a dependency is activated only when every attribute in its tail is reachable. Attribute closure then provides the formal bridge from graph traversal to superkey and candidate-key analysis.

For the dependency set implemented in the Python prototype, the mandatory attributes are $\{A, B, E, H\}$, and their closure contains the entire schema. Because none of these attributes can be derived from another dependency, the set is the unique candidate key. The original relation violates

2NF through $A \rightarrow G$ and also violates 3NF and BCNF. A dependency-preserving synthesis yields six relations, each of which satisfies BCNF with respect to the projected dependencies.

The prototype accurately implements hypergraph storage, direction-aware BFS, and iterative closure computation. Its present role is to provide the computational foundation for key and normalization analysis. Future development should automate candidate-key enumeration, minimal-cover construction, normal-form diagnosis, and schema decomposition while retaining traversal traces as explanations for each inferred attribute and design decision.

VII. Appendix

The source code of the program used in this paper can be found at GitHub: https:/github.com/BillyOWasTaken/fd-hypergraph

Acknowledgment

The author thanks the lecturers of IF2211 Algorithm Strategies and IF2240 Database courses at Institut Teknologi Bandung for the theoretical materials that motivated the integration of graph traversal and relational database design.

References

[1] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970, doi: 10.1145/362384.362685.

[2] C. Beeri, R. Fagin, and J. H. Howard, "A complete axiomatization for functional and multivalued dependencies in database relations," in *Proc. ACM SIGMOD Int. Conf. Management of Data*, 1977, pp. 47–61, doi: 10.1145/509404.509414.

[3] A. Bretto, *Hypergraph Theory: An Introduction*. Cham, Switzerland: Springer, 2013.

[4] G. Ausiello, G. F. Italiano, and U. Nanni, "Hypergraph traversal revisited: Cost measures and dynamic algorithms," in *Mathematical Foundations of Computer Science 1998*, Lecture Notes in Computer Science, vol. 1450. Berlin, Germany: Springer, 1998, pp. 1–16, doi: 10.1007/BFb0055754.

[5] R. Munir, "Breadth/Depth First Search (BFS/DFS)," Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/ rinaldi.munir/Stmik/2025-2026/13-BFS-DFS-(2026)-Bagian1.pdf

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, 2022.

[7] IF2240 Teaching Team, "Relational Database Design," Institut Teknologi Bandung, Bandung, Indonesia, 2026.

[8] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. New York, NY, USA: McGraw-Hill, 2019.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Billy Ontoseno Irawan
13524121