FOL Modeling of Integrity Constraints (Dependencies)

Alin Deutsch
Computer Science and Engineering, University of California San Diego
deutsch@cs.ucsd.edu

SYNONYMS

relational integrity constraints; dependencies

DEFINITION

Integrity constraints (also known as *dependencies* in the relational model) are domain-specific declarations which indicate the intended meaning of the data stored in a database. They complement the description of the structure of the data (e.g. in the relational model the structure is given by listing the names of tables and the names and types of their attributes). Integrity constraints express properties that must be satisfied by all instances of a database schema that can arise in the intended application domain (e.g. "no two distinct employees may have the same ssn value", "departments have a single manager", etc.).

HISTORICAL BACKGROUND

The reference textbook [1] provides a comprehensive, unifying overview of the many special classes of relational dependencies, their modeling in first-order logic (FOL), and the key problems in the study of dependencies. This entry is a condensed form of Chapter 10 in [1] (excluding the material on reasoning about dependencies). For more detail on the motivation and history of relational dependency theory, see the excellent survey papers [9, 14, 22].

Historically,*functional* dependencies were the first class to be introduced (by Codd [3]). *Multi-valued* dependencies were discovered independently in [24, 10, 5]. These were followed by a proliferation of dependency classes, typically expressed using ad-hoc syntax. These include *join* dependencies, and *inclusion* dependencies (a.k.a. referential integrity constraints) [4]. Fagin [10] first introduced *embedded multi-valued* dependencies (MVDs that hold in the projection of a relation), while [15] introduced the distinct class of *projected* JDs. Other classes include *subset* dependencies [20], *mutual* dependencies [17], *generalized mutual* dependencies [16], *transitive* dependencies [18], *extended transitive* dependencies [19], and *implied* dependencies [12].

The movement towards ever-refined classifications of dependencies was prompted mainly by research on automatic reasoning about dependencies, in particular their axiomatization. This movement was soon countered by the drive to unify the treatment of the various classes by finding a formalism that subsumes all of them. Nicolas [17] is credited with first observing that FDs, MVDs and others have a natural representation in first-order logic. In parallel, [2] introduced *tuple-generating* and *equality-generating* dependencies expressed in a tableaux-based notation, shown to be equivalent in expressive power to Fagin's *typed embedded dependencies* [8], which were expressed in first-order logic. The class DED$^{\neq}$ was introduced in [7] as an extension of embedded dependencies with disjunction and non-equalities (see also [13] for a particular case of DED$^{\neq}$s). [6] further extends the DED$^{\neq}$ class to allow negated relational atoms. Research on using arbitrary first-order logic sentences to specify constraints includes [11, 17, 21].

In contrast, *algebraic dependencies* are an alternative unifying framework developed by Yannakakis and Papadimitriou [23]. Algebraic dependencies are statements of containment between queries written in relational algebra, and have the same expressive power as first-order logic.

SCIENTIFIC FUNDAMENTALS

This section lists a few fundamental classes of relational dependencies, subsequently showing how they can be expressed in first-order logic. In the following, $R(U)$ denotes the schema of a relation with name R and set of attributes U.

Functional dependencies. A *functional dependency (FD)* on relations of schema $R(U)$ is an expression of the form

$$R : X \to Y \tag{1}$$

where $X \subseteq U$ and $Y \subseteq U$ are subsets of R's attributes. Instance r of schema $R(U)$ is said to *satisfy* FD fd, denoted $r \models fd$, if whenever tuples $t_1 \in r$ and $t_2 \in r$ agree on all attributes in X, they also agree on all attributes in Y:

$$r \models fd \;\Leftrightarrow\; \text{for every } t_1, t_2 \in r \text{ if } \pi_X(t_1) = \pi_X(t_2) \text{ then } \pi_Y(t_1) = \pi_Y(t_2).$$

Here, $\pi_X(t)$ denotes the projection of tuple t on the attributes in X.

For instance, consider a relation of schema

review(paper, reviewer, track)

listing a conference track a paper was submitted to, and a reviewer it was assigned to. The fact that every paper can be submitted to a single track is stated by the functional dependency

$$review : \; paper \to track.$$

Key dependencies. In the particular case when $Y = U$, a functional dependency of form (1) is called a *key dependency*, and the set of attributes X is a called a *key for* R.

Join dependencies. A *join dependency (JD)* on relations of schema $R(U)$ is an expression of the form

$$R : \bowtie [X_1, X_2, \ldots, X_n] \tag{2}$$

where for each $1 \leq i \leq n$, $X_i \subseteq U$, and $\bigcup_{1 \leq i \leq n} X_i = U$. Instance r of schema $R(U)$ *satisfies* JD jd, denoted $r \models jd$, if the n-way natural join of the projections of r on each of the attribute sets X_i yields r:

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

Here, $\Pi_X(r)$ denotes the projection of relation r on the attributes in X.

In the example, assume that a paper may be submitted for consideration by various tracks (e.g. poster or full paper), and that reviewers are not tied to the tracks. It makes sense to expect that for any given paper p, any track information listed with a reviewer of p is also listed with all other reviewers of p, since track and reviewer information are not correlated. This is expressed by requiring that the join of the projection of *review* on *paper, track* and of the projection on *paper, reviewer* yields the *review* table:

$$review : \bowtie [\{paper, track\}, \{paper, reviewer\}].$$

Multi-valued dependencies. In the particular case when $n = 2$, a join dependency of form (2) is called a *multi-valued dependency (MVD)*. Because MVDs were historically introduced and studied before JDs, they have their own notation: an MVD $R : \bowtie [X_1, X_2]$ is denoted

$$R : \; X \twoheadrightarrow Y, \tag{3}$$

where $X = X_1 \cap X_2$ and $Y = X_1 \setminus X_2$.

In the running example, the join dependency turns out to be a multi-valued dependency, which can be expressed using the following MVD-specific syntax:

$$review :\ paper \twoheadrightarrow track.$$

Inclusion dependencies. Functional and join dependencies and their special-case subclasses each pertain to single relations. The following class of dependencies can express connections between relations. An *inclusion dependency (IND)* on pairs of relations of schemas $R(U)$ and $S(V)$ (with R and S not necessarily distinct) is an expression of the form

$$R[X] \subseteq S[Y] \tag{4}$$

where $X \subseteq U$ and $Y \subseteq V$. Inclusion dependencies are also known as *referential constraints*. Relations r and s of schemas $R(U)$, respectively $S(V)$ satisfy inclusion dependency id, denoted $r, s \models id$, if the projection of r on X is included in the projection of s on Y:

$$r, s \models id \Leftrightarrow\ \Pi_X(r) \subseteq \Pi_Y(s).$$

When R and S refer to the same relation name, then $r = s$ in the above definition of satisfaction.

In the running example, assume that the database contains also a relation of schema

$$PC(member, affiliation)$$

listing the affiliation of every program committee member. Then one can require that papers be reviewed only by PC members (no external reviews allowed) using the following IND:

$$review[reviewer] \subseteq PC[member].$$

Foreign key dependencies. In the particular case when Y is a key for relations of schema S ($S :\ Y \to V$), INDs of form (4) are called *foreign key dependencies*. Intuitively, in this case the projection on X of every tuple t in r contains the key of a tuple from the "foreign" table s.

In the running example, assuming that every PC member is listed with only one primary affiliation, *member* is a key for *PC*, so the IND above is really a foreign key dependency.

Expressing Dependencies in First-Order Logic

Embedded dependencies. Despite their independent introduction and widely different syntax, it turns out that all classes of dependencies illustrated above (and many more, including the ones mentioned in the historical background section) can be expressed using a fragment of the language of first-order logic. This fragment is known as the class of *embedded dependencies*, which are formulas of the form

$$\forall x_1 \ldots \forall x_n\ \varphi(x_1, \ldots, x_n) \to \exists z_1 \ldots \exists z_k\ \psi(y_1, \ldots, y_m), \tag{5}$$

where $\{z_1, \ldots, z_k\} = \{y_1, \ldots, y_m\} \setminus \{x_1, \ldots, x_n\}$, and φ is a possibly empty and ψ is a non-empty conjunction of relational and equality atoms. A *relational atom* has form $R(w_1, \ldots, w_l)$, and an *equality atom* has form $w = w'$, where each of $w, w', w_1, \ldots, w_l$ are variables or constants. The particular case when all atoms in ψ are equalities yields the class known as *equality-generating dependencies (EGD)*, while the case when only relational atoms occur in ψ defines the class of *tuple-generating dependencies (TGD)*.

The power of embedded dependencies is illustrated next by expressing the classes of dependencies described above.

Functional dependencies. Assume without loss of generality that in (1), $|X| = k$, $|Y| = l$, and $|U \backslash (X \cup Y)| = m$, and that the ordering of attributes in U is $U = X, Y, Z$. Then any functional dependency of form (1) is expressible as the embedded dependency (actually an EGD):

$$\forall x_1 \ldots \forall x_k \ \forall y_1 \ldots \forall y_l \ \forall y'_1 \ldots \forall y'_l \ \forall z_1 \ldots \forall z_m \ \forall z'_1 \ldots \forall z'_m$$
$$R(x_1, \ldots, x_k, y_1, \ldots, y_l, z_1, \ldots, z_m) \wedge R(x_1, \ldots, x_k, y'_1, \ldots, y'_l, z'_1, \ldots, z'_m) \rightarrow y_1 = y'_1 \wedge \ldots \wedge y_l = y'_l$$

In the particular case when X is a key, $m = 0$ and there are no z_i, z'_i variables in the above embedded dependency.

The functional dependency *review* : *paper* $\rightarrow$ *track* in the running example, can be expressed as the following embedded dependency:

$$\forall p \forall r \forall t \forall r' \forall t' \ \mathit{review}(p, r, t) \wedge \mathit{review}(p, r', t') \rightarrow t = t'.$$

Join dependencies. Join dependencies of form (2), are expressed by observing that for every relation r of schema $R(U)$, the inclusion

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

holds trivially. Therefore only the opposite inclusion needs to be expressed,

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

This end requires the following notation. Recalling that the set of attributes U in the schema of R is ordered, let $pos(A)$ denote the position of an attribute $A \in X$ in the ordered set U. For a set of attributes $X \subseteq U$, $pos(X)$ denotes the set $\{pos(A) \mid A \in X\}$. In the following, given a tuple of variables $\bar{u}$, $\bar{u}[k]$ denotes the k^{th} variable in $\bar{u}$.

- Let $\{\bar{u}_i\}_{1 \leq i \leq n}$ be a family of tuples of $|U|$ variables each, such that for every $1 \leq i < j \leq n$ and every $1 \leq k \leq |U|$, $\bar{u}_i[k] = \bar{u}_j[k]$ if and only if $k \in pos(X_i \cap X_j)$.
- Let $\bar{w}$ be a tuple of $|U|$ variables, such that for every $1 \leq k \leq |U|$ and every $1 \leq i \leq n$, if $k \in pos(X_i)$ then $\bar{w}[k] = \bar{u}_i[k]$. It is easy to check that $\bar{w}$ is well-defined: indeed, since $\bigcup_i X_i = U$, for each k there is at least one i with $k \in pos(X_i)$. Moreover, by definition of the family $\{\bar{u}_i\}_i$, $\bar{u}_i[k] = \bar{u}_j[k]$ whenever $k \in pos(X_i)$ and $k \in pos(X_j)$.
- Finally, let $V = \{v_1, \ldots v_m\} = \bigcup_{i=1}^{n} \bar{u}_i$ (with each tuple $\bar{u}_i$ viewed as a set of variables). Notice that variables occurring in several $\bar{u}_i$ tuples appear just once among V.

Then the join dependency of form (2) is given by the embedded dependency (a TGD, really):

$$\forall v_1 \ldots \forall v_m \ R(\bar{u}_1) \wedge \ldots \wedge R(\bar{u}_n) \rightarrow R(\bar{w}).$$

Join dependency *review* : $\bowtie$ [{*paper*, *track*}, {*paper*, *reviewer*}] is expressed as the following embedded dependency:

$$\forall p \forall r_1 \forall t_1 \forall r_2 \forall t_2 \ \mathit{review}(p, r_1, t_1) \wedge \mathit{review}(p, r_2, t_2) \rightarrow \mathit{review}(p, r_2, t_1).$$

Inclusion dependencies. To express inclusion dependencies, assume without loss of generality that in (4) $R(U) = R(Z, X)$ and $S(V) = S(Y, W)$. Then the inclusion dependency (4) is captured by the following embedded dependency (TGD):

$$\forall z_1 \ldots \forall z_{|Z|} \ \forall x_1 \ldots \forall x_{|X|} \ R(z_1, \ldots, z_{|Z|}, x_1, \ldots, x_{|X|}) \rightarrow \exists w_1 \ldots \exists w_{|W|} \ S(x_1, \ldots, x_{|X|}, w_1, \ldots, w_{|W|}).$$

In the running example, the inclusion dependency $\mathit{review}[\mathit{reviewer}] \subseteq PC[\mathit{member}]$ is expressible as the embedded dependency

$$\forall p \forall r \forall t \ \mathit{review}(p, r, t) \rightarrow \exists a \ PC(r, a).$$

Other classes of dependencies. Embedded dependencies turn out to be sufficiently expressive to capture virtually all other classes of dependencies studied in the literature.

Other Constraints

By employing more expressive sub-languages of first-order logic, one can capture additional, naturally occurring constraints. For instance, extending embedded dependencies with disjunction, one obtains the language of *disjunctive embedded dependencies (DED)* of form

$$\forall \bar{x}\ \varphi(\bar{x}) \rightarrow \bigvee_{i=1}^{l}\ \exists \bar{z}_i\ \psi_i(\bar{y}_i), \tag{6}$$

where $\bar{x}$ is a tuple of variables, and so are $\bar{z}_i, \bar{y}_i$ for every $1 \leq i \leq l$. Analogously to (5), $\bar{z}_i = \bar{y}_i \setminus \bar{x}$. φ and each ψ_i are conjunctions of relational and equality atoms as in (5). If in addition one allows *non-equality atoms* of the form $w \neq w'$, one obtains the class of DEDs with non-equality, DED$^{\neq}$, which is in turn a fragment of first-order logic.

Cardinality constraints. The language DED$^{\neq}$ can express cardinality constraints. In the running example, $\delta_1 \in$ DED$^{\neq}$ states that every paper has at least two reviews:

$$(\delta_1) \quad \forall p \forall r_1 \forall t\ \mathit{review}(p, r_1, t) \rightarrow \exists r_2\ \mathit{review}(p, r_2, t) \wedge r_1 \neq r_2.$$

δ_2 below states that every paper receives at most two reviews:

$$(\delta_2) \quad \forall p \forall r_1 \forall r_2 \forall r_3 \forall t\ \mathit{review}(p, r_1, t) \wedge \mathit{review}(p, r_2, t) \wedge \mathit{review}(p, r_3, t) \wedge r_1 \neq r_2 \rightarrow r_3 = r_1 \vee r_3 = r_2.$$

Note that the conjunction of δ_1 and δ_2 requires each paper to receive precisely two reviews.

Domain constraints. The language DED$^{\neq}$ can be employed to restrict the domain of an attribute. Such restrictions are commonly known as *domain constraints*. For instance, in the running example, this is how to specify that the conference has only three kinds of tracks: "research", "industrial" and "demo":

$$(\delta_3) \quad \forall p \forall r \forall t\ \mathit{review}(p, r, t) \rightarrow t = \text{“}\mathit{research}\text{”} \vee t = \text{“}\mathit{industrial}\text{”} \vee t = \text{“}\mathit{demo}\text{”}.$$

Representational constraints. Many application are based on data models that are richer than the relational model (e.g. object-oriented, object-relational, XML, RDF models). However, they often leverage the mature relational technology by supporting the storage of their data in a relational database. The resulting relations satisfy certain constraints that stem from the original data model. This entry refers to them as *representational constraints*. In order to maintain the relational storage and to efficiently process queries over it, it is imperative to exploit these representational constraints. An obstacle to doing so is the fact that, depending on the original model they encode relationally, representational constraints tend to not fit neatly into any of the classes of relational integrity constraints devised for native relational data. Again, first-order logic comes to the rescue.

Representational constraints for the relational representation of XML are illustrated next. While there are many possible representations, they are all equivalent to the following simple one [7] which captures the fact that in the XML data model, elements are the tagged nodes of a tree. The tree is represented using the following relations (among others):

elem(node, tag) *child(source,target)* *desc(source,target)*

where the *elem* relation lists for every element e, the identifier of the tree node modeling e, and the tag of e; the *child* table is the edge relation of the XML tree, according to which *source* is the identifier of the parent node and *target* the identifier of the child node; *desc* is the descendant relation in the tree, whose *target* node is a descendant of the *source* node. Any instance storing an actual XML tree in these tables must satisfy, among others, the following constraints, all expressible in DED$^{\neq}$: every element has at most one tag (expressed in (7) below); every element has at most one parent (8); children of a node are also descendants of this node (9); the descendant relation is transitive (10); if two elements have a common descendant, then they either coincide or one is the descendant of the other (11).

(7) $\forall n \forall t_1 \forall t_2 \quad elem(n, t_1) \wedge elem(n, t_2) \rightarrow t_1 = t_2$

(8) $\forall n \forall p_1 \forall p_2 \quad child(p_1, n) \wedge child(p_2, n) \rightarrow p_1 = p_2$

(9) $\forall s \forall t \quad child(s, t) \rightarrow desc(s, t)$

(10) $\forall s \forall u \forall t \quad desc(s, u) \wedge desc(u, t) \rightarrow desc(s, t)$

(11) $\forall n_1 \forall n_2 \forall d \quad desc(n_1, d) \wedge desc(n_2, d) \rightarrow n_1 = n_2 \vee desc(n_1, n_2) \vee desc(n_2, n_1)$

KEY APPLICATIONS*

The role of integrity constraints is to incorporate more semantics into the data model. This in turn enables an improved schema design, as well as the delegation to the database management system (DBMS) of the task of enforcing and exploiting this semantics.

Schema design. Integrity constraints are useful for selecting the most appropriate schema for a particular database application domain. It turns out that the same information can be stored in tables in many ways, some more efficient than others with respect to avoiding redundant storage of data, improved update and query performance, and better readability. While in the early days of database application development, appropriate schema selection started out as an art, it quickly evolved into a science enabling automatic schema design tools (also known as "wizards"). Such tools are based on database theory research that proposes schema design methodology starting from a single, "universal relation", which is then decomposed into new relations that satisfy desirable normal forms that take advantage of the known integrity constraints [1]. For instance, in the running example, both the FD *review* : *paper* $\rightarrow$ *track* and the MVD *review* : *paper* $\twoheadrightarrow$ *track* suggest decomposing relation *review* into two tables, one associating papers with their track, and one associating papers with their reviewers, to avoid the redundant listing of track information with every reviewer.

Automatic integrity enforcement. For the purpose of integrity enforcement, the DBMS automatically checks every update operation for compliance with the declared integrity constraints, automatically rejecting non-compliant updates. The database administrator can therefore rest assured that the integrity of her data will be preserved despite any bugs in the applications accessing the database. This guarantee is all the more important when considering that several applications are usually running against the same database. These applications are not always under the control of the database administrator, and are often developed by third parties, which renders their code unavailable for verification. Even with full access to the application code, verification (like all software verification) is technically challenging and does not scale well with increasing number of applications or modifications to their code.

Query optimization. The query optimizer can exploit its constraint-derived understanding of the data to automatically rewrite queries for more efficient execution. Delegating this task to the DBMS ensures that queries are efficiently executed even when they are issued by applications whose developers write the queries in sub-optimal forms due to insufficient insight into the integrity constraints. More importantly, automatic optimization inside the DBMS handles the queries generated by tools rather than human developers. These queries are usually quite far from optimal. This problem is especially prevalent when queries over a rich data model M are translated to relational queries over the relational representation of M (e.g. XQuery queries over relational storage of XML). The *chase* is a very powerful tool for reasoning about (and exploiting in optimization) dependencies specified in first-order logic (see [1] for references to the papers that independently discovered the chase, for various classes of dependencies).

A unified view of dependencies. Given the plethora of classes of dependencies formulated and studied independently in the literature, the task of specifying the meaning of an application domain via integrity constraints would be daunting if the developer had to fit them into these classes. In addition, tailoring the tasks of schema design, optimization and integrity enforcement to every class of dependencies (and every combination of such classes) is impractical. First-order logic provides a formalism for the simple specification of integrity

constraints, with well-understood semantics and sufficient expressive power to capture all common classes of dependencies, and beyond. The insight that virtually all dependency classes are expressible in the same formalism set the foundation for their uniform treatment in research and applications.

CROSS REFERENCE*

Equality-Generating Dependencies
Tuple-Generating Dependencies
Chase

RECOMMENDED READING

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*. Addison-Wesley, 1995.

[2] C. Beeri and M. Y. Vardi. The implication problem for data dependencies. In *International Conference on Algorithms, Languages and Programming*, pages 73–85, 1981.

[3] E. F. Codd. Relational completeness of database sublanguages. In R. Rustin, editor, *Courant Computer Science Symposium 6: Data Base Systems*, pages 65–98. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[4] C. J. Date. Referential integrity. In *International Conference on Very Large Databases*, pages 2–12, 1981.

[5] Claude Delobel. Normalization and hierarchical dependencies in the relational data model. *ACM Trans. on Database Systems*, 3(3):201–222, 1978.

[6] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting queries using views with access patterns under integrity constraints. *Theor. Comput. Sci*, 371(3):200–226, 2007.

[7] Alin Deutsch and Val Tannen. XML queries and constraints, containment and reformulation. *Theor. Comput. Sci.*, 336(1):57–87, 2005.

[8] R. Fagin. Horn clauses and database dependencies. *J. ACM*, 29(4):952–985, 1982.

[9] R. Fagin and M. Y. Vardi. The theory of data dependencies: A survey. In M. Anshel and W. Gewirtz, editors, *Mathematics of Information Processing: Proceedings of Symposia in Applied Mathematics*, volume 34, pages 19–27. American Mathematical Society, Providence, RI, 1986.

[10] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems*, 2(3):262–278, 1977.

[11] H. Gallaire and J. Minker. *Logic and Databases*. Plenum Press, New York, 1978.

[12] S. Ginsburg and S.M. Zaiddan. Properties of functional dependency families. *JACM*, 29(4):678–698, 1982.

[13] Gösta Grahne and Alberto O. Mendelzon. Tableau techniques for querying information sources through global schemas. In *ICDT*, pages 332–347, 1999.

[14] Paris C. Kannellakis. Elements of relational database theory. In J. Van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 1074–1156. Elsevier, Amsterdam, 1991.

[15] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi. On the foundations of the universal relation model. *ACM Transactions on Database Systems*, 9(2):283–308, 1984.

[16] A. O. Mendelzon and D. Maier. Generalized mutual dependencies and the decomposition of database relations. In *International Conference on Very Large Databases*, pages 75–82, 1979.

[17] J.-M. Nicolas. First order logic formalization for functional, multivalued, and mutual dependencies. *Acta Informatica*, 18(3):227–253, 1982.

[18] J. Paredaens. Transitive dependencies in a database scheme. Technical Report R387, MBLE, Brussels, 1979.

[19] D. S. Parker and K. Parsaye-Ghomi. Inference involving embedded multivalued dependencies and transitive dependencies. In *ACM SIGMOD Symposium on the Management of Data*, pages 52–57, 1980.

[20] Yehoshua Sagiv and Scott F. Walecka. Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies. *J. ACM*, 29(1):103–117, 1982.

[21] M. Y. Vardi. On decomposition of relational databases. In *IEEE Conference on Foundations of Computer Science*, pages 176–185, 1982.

[22] M. Y. Vardi. Trends in theoretical computer science. In E. Borger, editor, *Fundamentals of dependency theory*, pages 171–224. Computer Science Press, Rockville, MD, 1987.

[23] M. Yannakakis and C. Papadimitriou. Algebraic dependencies. *Journal of Computer and System Sciences*, 25(2):3–41, 1982.

[24] Carlo Zaniolo. *Analysis and Design of Relational Schemata for Database Systems*. PhD thesis, University of California, Los Angeles, 1976. Technical Report UCLA-Eng-7669, Department of Computer Science.