CMPS 277 – Principles of Database Systems

https://courses.soe.ucsc.edu/courses/cmps277/Fall11/01

Lecture #15
Functional Dependencies

- **Definition:** A relational schema \( R \) satisfies the functional dependency
  \[ A_1, \ldots, A_m \rightarrow B_1, \ldots, B_k \]
  if every instance \( r \) of \( R \) satisfies \( A_1, \ldots, A_m \rightarrow B_1, \ldots, B_k \).

- **Fact:** In effect, the above definition imposes a semantic restriction on the instances of \( R \), namely, we **disallow** all instances that violate the functional dependency \( A_1, \ldots, A_m \rightarrow B_1, \ldots, B_k \).

- **Notation:** If \( X \) and \( Y \) are sets of attributes of \( R \), then we write
  \[ X \rightarrow Y \]
  to denote the functional dependency with the members of \( X \) in the left-hand side and the members of \( Y \) in the right-hand side.
Definition: Assume that R is a relation schema, F is a set of FDs, and $X \rightarrow Y$ is a FD (all with attributes from R). We say that F logically implies $X \rightarrow Y$ (and write $F \models X \rightarrow Y$), if for every instance $r$ of R that satisfies F, we have that $r$ also satisfies $X \rightarrow Y$.

Example: Transitivity Rule
If R satisfies $A \rightarrow B$ and $B \rightarrow C$, then R must also satisfy $A \rightarrow C$. Hence,

$$\{A \rightarrow B, B \rightarrow C\} \models A \rightarrow C.$$  

Definition: $F^+ = \{X \rightarrow Y: F \models X \rightarrow Y\}$ is the closure of F.
Algorithmic Problems about Functional Dependencies

Problem 1: Given a relational schema $R$, a set $F$ of FDs on $R$ and a set $X$ of attributes of $R$, determine whether or not $X$ is a superkey of $R$.

Problem 2: Given a relational schema $R$ and a set $F$ of FDs on $R$, find all candidate keys.

Problem 3: Given a relational schema $R$, a set $F$ of FDs on $R$, and a FD $X \rightarrow Y$, determine whether or not $F \models X \rightarrow Y$ (i.e., $X \rightarrow Y \in F^+$)

- An algorithm for Problem 1 can be used to design an algorithm for Problem 2.
- Clearly, Problem 1 is a special case of Problem 3.
Algorithmic Problems about Functional Dependencies

Problem 3: Given a relational schema $R$, a set $F$ of FDs on $R$, and a FD $X \rightarrow Y$, determine whether or not $F \models X \rightarrow Y$.

Fact: The following are equivalent:
- $F \models X \rightarrow Y$, where $Y = \{B_1, \ldots, B_k\}$
- $F \models X \rightarrow B_i$, for every $B_i \in Y$.

Definition: $X^+ = \{B: F \models X \rightarrow B\}$ is the closure of $X$ with respect to $F$.

Fact:
- $F \models X \rightarrow Y$ if and only if $Y \subseteq X^+$.
- Hence, to tell whether $F \models X \rightarrow Y$, it suffices to compute $X^+$. 
The Closure Algorithm

**Input:** Relational schema R with attribute set U, set F of FDs, \( X \subseteq U \)

**Output:** \( X^+ = \{ B : F \models X \rightarrow B \} \).

**Initialization Step:** \( X_0 = X \)

**Recursive Step:**
\[
X_{n+1} = X_n \cup \{ B : \text{there is a FD } Y \rightarrow Z \text{ in } F \text{ such that } Y \subseteq X_n \text{ and } B \in Z \}
\]

**Stopping Rule:** When \( X_n = X_{n+1} \) for the first time, stop and output \( X_n \).
The Closure Algorithm: Summary

- The closure algorithm is a simple algorithm with a non-trivial proof of correctness.

- The running time of the closure algorithm is \textit{quadratic} in the size (length) of \( F \) and \( X \) (\textit{why?}).

- The closure algorithm can be refined to run in \textit{linear} time in the size of \( F \) and \( X \).

- The closure algorithm can be used to determine whether \( F \models X \rightarrow Y \), and also to compute superkeys and candidate keys.

- In particular, the \textbf{Implication Problem for FDs} is solvable in polynomial time (in fact, in quadratic time).
The Closure Algorithm: An Application

Example: R with attributes A, B, C, D, E

\[ F = \{ A \rightarrow C, B \rightarrow C, CD \rightarrow E \} \]

- Compute \( \{A,D\}^+ \)
  - \( X_0 = \{A,D\}, X_1 = \{A,C,D\}, X_2 = \{A,C,D,E\} = X_3 = X^+ \)
  - Consequently, \( AD \rightarrow CE \), but \( AD \not\rightarrow B \).

- Show that \( \{A,B,D\} \) is the only candidate key of R.
  - Use the closure algorithm to show that \( \{A,B,D\} \) is a superkey.
  - Use the closure algorithm to show that no other set of attributes is a candidate key (or reason directly).
Functional Dependencies and Relational Calculus

Fact: Every functional dependency $A_1, \ldots, A_m \rightarrow B$ can be expressed in relational calculus. More formally, there is a relational calculus formula $\psi$ such that for every database instance $r$, we have that the following are equivalent:

- $r \models A_1, \ldots, A_m \rightarrow B$
- $r \models \psi$.

Proof (by example): Assume that $R$ has attributes $A, B, C, D$. Then the following are equivalent for the FD $A, B \rightarrow C$.

- $r \models A, B \rightarrow C$.
- $r \models \forall x, y, z, w, z', w'(R(x, y, z, w) \land R(x, y, z', w') \rightarrow z = z')$.

Note: The formula $\forall x, y, z, w, z', w'(R(x, y, z, w) \land R(x, y, z', w') \rightarrow z = z')$ is an example of an equality-generating dependency (egd).
Equality-Generating Dependencies

Definition: An equality-generating dependency (egd) is a formula of relational calculus of the form:

\[ \forall x_1,\ldots,x_n(\varphi(x_1,\ldots,x_n) \rightarrow x_i = x_j), \]

where \( \varphi(x_1,\ldots,x_n) \) is a conjunction of atomic formulas (i.e., \( \varphi \) is a conjunctive query).

Examples:

- \[ \forall x_1,x_2,x_3(R(x_1,x_2) \land P(x_2,x_3) \land T(x_2) \rightarrow x_2 = x_3) \]
  - This is an egd, but not a FD.

- \[ \forall x_1,x_2,x_3(R(x_1,x_2) \land R(x_1,x_3) \rightarrow x_2 = x_3) \]
  - This is both an egd and a FD, namely \( A_1 \rightarrow A_2 \).
Inclusion Dependencies

Example: \( \text{ENROLLS}(\text{student-id}, \text{name}, \text{course}), \)
\( \text{PERFORM}(\text{student-id}, \text{course}, \text{grade}) \)

Consider the integrity constraint:
- “every student enrolled in a course is assigned a grade”

This is an example of an inclusion dependency;
it is denoted by:
\( \text{ENROLLS}[\text{student-id, course}] \subseteq \text{PERFORM}[\text{student-id, course, }]. \)
Inclusion Dependencies

**Definition:** An inclusion dependency (ID) is an expression of the form

\[ S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n], \]

where

- \( A_1, \ldots, A_n \) are distinct attributes from \( S \)
- \( B_1, \ldots, B_n \) are distinct attributes from \( T \).

- A database instance \( r \) satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if for every tuple \( s \in S \) with values \( c_1, \ldots, c_n \) for the attributes \( A_1, \ldots, A_n \), there is a tuple \( t \in T \) with values \( c_1, \ldots, c_n \) for the attributes \( B_1, \ldots, B_n \).

- A database schema satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if every instance of the schema satisfies this ID.
Inclusion Dependencies

Definition: An inclusion dependency (ID) is an expression of the form
\[ S[A_1,\ldots,A_n] \subseteq T[B_1,\ldots,B_n], \]
where
- \( A_1,\ldots,A_n \) are distinct attributes from \( R \)
- \( B_1,\ldots,B_n \) are distinct attributes from \( S \).

- A database instance \( r \) satisfies \( S[A_1,\ldots,A_n] \subseteq T[B_1,\ldots,B_n] \) if for every tuple \( s \in S \) with values \( c_1,\ldots,c_n \) for the attributes \( A_1,\ldots,A_n \), there is a tuple \( t \in T \) with values \( c_1,\ldots,c_n \) for the attributes \( B_1,\ldots,B_n \).

- A database schema satisfies \( S[A_1,\ldots,A_n] \subseteq T[B_1,\ldots,B_n] \) if every instance of the schema satisfies this ID.
Inclusion Dependencies

**Definition:** An *inclusion dependency (ID)* is an expression of the form
\[ S[A_1,...,A_n] \subseteq T[B_1,...,B_n], \]
where

- \( A_1,...,A_n \) are distinct attributes from \( R \)
- \( B_1,...,B_n \) are distinct attributes from \( S \).

- A database instance \( r \) satisfies \( S[A_1,...,A_n] \subseteq T[B_1,...,B_n] \) if for every tuple \( s \in S \) with values \( c_1,...,c_n \) for the attributes \( A_1,...,A_n \), there is a tuple \( t \in T \) with values \( c_1,...,c_n \) for the attributes \( B_1,...,B_n \).

- A database schema satisfies \( S[A_1,...,A_n] \subseteq T[B_1,...,B_n] \) if every instance of the schema satisfies this ID.
Inclusion Dependencies

**Definition:** An inclusion dependency (ID) is an expression of the form

\[ S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \]

where

- \( A_1, \ldots, A_n \) are distinct attributes from R
- \( B_1, \ldots, B_n \) are distinct attributes from S.

- A database instance \( r \) satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if for every tuple \( s \in S \) with values \( c_1, \ldots, c_n \) for the attributes \( A_1, \ldots, A_n \), there is a tuple \( t \in T \) with values \( c_1, \ldots, c_n \) for the attributes \( B_1, \ldots, B_n \).

- A database schema satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if every instance of the schema satisfies this ID.
Inclusion Dependencies

**Definition:** An inclusion dependency (ID) is an expression of the form

\[ S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n], \]

where
- \( A_1, \ldots, A_n \) are distinct attributes from \( R \)
- \( B_1, \ldots, B_n \) are distinct attributes from \( S \).

- A database instance \( r \) satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if for every tuple \( s \in S \) with values \( c_1, \ldots, c_n \) for the attributes \( A_1, \ldots, A_n \), there is a tuple \( t \in T \) with values \( c_1, \ldots, c_n \) for the attributes \( B_1, \ldots, B_n \).

- A database schema satisfies \( S[A_1, \ldots, A_n] \subseteq T[B_1, \ldots, B_n] \) if every instance of the schema satisfies this ID.
Inclusion Dependencies and Relational Calculus

Fact: Every inclusion dependency \( S[A_1,\ldots,A_n] \subseteq T[B_1,\ldots,B_n] \) can be expressed in relational calculus.

Proof (by example): Consider the ID

\( \text{ENROLLS}[\text{student-id},\text{course}] \subseteq \text{PERFORM}[\text{student-id},\text{course}] \),

which expresses the integrity constraint:

“every student enrolled in a course is assigned a grade”.

This ID is equivalent to the relational calculus formula

\[
\forall x,y,z \ (\text{ENROLLS}(x,y,z) \rightarrow \exists w \ \text{PERFORM}(x,z,w)).
\]

Note: The formula \( \forall x,y,z \ (\text{ENROLLS}(x,y,z) \rightarrow \exists w \ \text{PERFORM}(x,z,w)) \) is an example of a tuple-generating dependency (tgd).
Tuple-Generating Dependencies

Definition: A tuple-generating dependency (tgd) is a formula of relational calculus of the form:

$$\forall x_1,\ldots,x_n(\varphi(x_1,\ldots,x_n) \rightarrow \exists y_1,\ldots,y_m \psi(x'_1,\ldots,x'_k, y_1,\ldots,y_m)),$$

where

- $\varphi(x_1,\ldots,x_n)$ and $\psi(x'_1,\ldots,x'_k, y_1,\ldots,y_m)$ are conjunctions of atomic formulas
- The variables $x'_1,\ldots,x'_k$ are among the variables $x_1,\ldots,x_n$.

Note: In effect, a tuple-generating dependency asserts that one conjunctive query (namely, the one defined by $\varphi(x_1,\ldots,x_n)$) is contained in another conjunctive query (namely, the one defined by $\exists y_1,\ldots,y_m \psi(x'_1,\ldots,x'_k, y_1,\ldots,y_m)$).
Tuple-Generating Dependencies

Examples:
- Every inclusion dependency is a tuple-generating dependency.
- $\forall x, y, z \ (E(x, y) \land E(y, z) \rightarrow E(x, z))$
  - This is a tgd, but not an ID. It asserts that $E$ is transitive.
- $\forall x, y \ (E(x, y) \rightarrow \exists z \ (F(x, z) \land F(z, y)))$
  - This says that for every edge in $E$, there is a path of length 2 in $F$.
- $\forall x, y, z \ (P(x, y, z) \rightarrow R(x, y) \land T(y, z))$
  - This says that $P$ is decomposed to $R$ and $T$. 
Embedded Implicational Dependencies

**Definition:** A database integrity constraint is an **embedded implicational dependency** if it is either a tuple-generating dependency or an equality-generating dependency.

**Fact:** Embedded implicational dependencies contain as special cases the various classes of integrity constraints studied in the 1970s and the early 1980s, such as:

- Functional dependencies
- Join dependencies
- Inclusion dependencies.
- Multivalued dependencies.

(see the survey paper on database dependencies by Fagin and Vardi)
Relational Calculus in Databases

Note:

- Relational calculus has been used in databases in two different ways:
  - As a database query language
  - As a specification language for expressing integrity constraints.

- In what follows, we will see that relational calculus is also used to formalize critical data interoperability tasks, such as
  - Data integration and
  - Data exchange
The Data Interoperability Problem

- Data may reside
  - at several different sites
  - in several different formats (relational, XML, ...).

- Applications need to access all these data.

- Two different, but closely related, facets of data interoperability:
  - **Data Integration** (aka **Data Federation**):
  - **Data Exchange** (aka **Data Translation**):
Data Integration

Query heterogeneous data in different sources via a virtual global schema
Data Exchange

Transform data structured under a source schema into data structured under a different target schema.

Source Schema \( S \)  

Target Schema \( T \)

\( \Sigma \)

\( I \)  \( \longrightarrow \)  \( J \)
Data Exchange

Data Exchange is an old, but recurrent, database problem

- Phil Bernstein – 2003
  “Data exchange is the oldest database problem’

- EXPRESS: IBM San Jose Research Lab – 1977
  EXtraction, Pprocessing, and REStructuring Ssystem
  for transforming data between hierarchical databases.

- Data Exchange underlies:
  - Data Warehousing, ETL (Extract-Transform-Load) tasks;
  - XML Publishing, XML Storage, ...
Challenges in Data Interoperability

Fact:
- Data interoperability tasks require expertise, effort, and time.
- Human experts have to generate complex transformations that specify the relationship between schemas written as programs (e.g., in Java) or as SQL/XSLT scripts.
- At present, there is relatively little automation.

Question: How can we address these challenges?

Answer: Introduce a higher level of abstraction that makes it possible to separate the design of the relationship between schemas from its implementation.
Schema Mappings

- Schema mappings: High-level, declarative assertions that specify the relationship between two database schemas.

- Schema mappings constitute the essential building blocks in formalizing and studying data interoperability tasks, including data integration and data exchange.

- Schema mappings help with the development of tools:
  - Are easier to generate and manage (semi)-automatically;
  - Can be compiled into SQL/XSLT scripts automatically.
Outline

- Schema Mappings as a framework for formalizing and studying data interoperability tasks.

- Data Exchange and Solutions in Data Exchange
  - Universal Solutions and the Core.

- Query Answering in Data Exchange.

- Managing schema mappings via operators:
  - The composition operator
  - The inverse operator and its variants
The results that will be presented in this part of the course are based on work carried out in the past five years in collaboration with:

- Ron Fagin, IBM Almaden Research Center
- Lucian Popa, IBM Almaden Research Center
- Wang-Chiew Tan, UC Santa Cruz & IBM Almaden Research Center

Papers have appeared in the proceedings of ICDT, PODS, SIGMOD, and VLDB, and in the journals ACM TODS and TCS.

Additional collaborators include:

- Renée J. Miller, University of Toronto
- Ariel Fuxman, University of Toronto (now with Microsoft Search Labs)
- Jonathan Panttaja, UC Santa Cruz
- Balder ten Cate, UC Santa Cruz
### Schema Mappings

- **Schema Mapping** $M = (S, T, \Sigma)$
  - **Source** schema $S$, **Target** schema $T$
  - A set $\Sigma$ of high-level, declarative assertions (constraints) that specify the relationship between $S$-instances and $T$-instances.

- $\text{Inst}(M) = \{ (I, J) : I \text{ is an } S\text{-instance, } J \text{ is a } T\text{-instance, and } (I, J) \models \Sigma \}$. 
Schema Mappings & Data Exchange

- **Schema Mapping** $\mathbf{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$
  - **Source** schema $\mathbf{S}$, **Target** schema $\mathbf{T}$
  - A set $\Sigma$ of high-level, declarative assertions (constraints) that specify the relationship between $\mathbf{S}$-instances and $\mathbf{T}$-instances.

- **Data Exchange** via the schema mapping $\mathbf{M} = (\mathbf{S}, \mathbf{T}, \Sigma)$
  Transform a given **source** instance $I$ to a **target** instance $J$, so that $(I, J)$ satisfy the specifications $\Sigma$ of $\mathbf{M}$. 
Solutions in Schema Mappings

**Definition**: Schema Mapping $M = (S, T, \Sigma)$

If $I$ is a source instance, then a solution for $I$ is a target instance $J$ such that $(I, J)$ satisfy $\Sigma$.

**Fact**: In general, for a given source instance $I$,

- No solution for $I$ may exist (the constraints overspecify) or
- Multiple solutions for $I$ may exist; in fact, infinitely many solutions for $I$ may exist (the constraints underspecify).
Schema Mappings: Basic Problems

**Definition**: Schema Mapping \( M = (S, T, \Sigma) \)

- The **existence-of-solutions problem** \( \text{Sol}(M) \): (decision problem)
  Given a source instance \( I \), is there a solution \( J \) for \( I \)?

- The **data exchange problem associated with** \( M \): (function problem)
  Given a source instance \( I \), construct a solution \( J \) for \( I \), provided a solution exists.
Ideally, schema mappings should be
- expressive enough to specify data interoperability tasks;
- simple enough to be efficiently manipulated by tools.

**Question:** How are schema mappings specified?

**Answer:** Use a high-level, declarative language. In particular, it is natural to try to use relational calculus (first-order logic) as a specification language for schema mappings.

**Fact:** There is a fixed relational calculus sentence specifying a schema mapping $M^*$ such that $\text{Sol}(M^*)$ is undecidable.

Hence, we need to restrict ourselves to well-behaved fragments of relational calculus.
Let us consider some simple tasks that a schema mapping specification language should support:

- **Copy (Nicknaming):**
  - Copy each source table to a target table and rename it.

- **Projection:**
  - Form a target table by projecting on one or more columns of a source table.

- **Decomposition:**
  - Decompose a source table into two or more target tables.

- **Column Augmentation:**
  - Form a target table by adding one or more columns to a source table.

- **Join:**
  - Form a target table by joining two or more source tables.

- **Combinations of the above** (e.g., “join + column augmentation”)
Schema Mapping Specification Languages

- **Copy (Nicknaming):**
  \[ \forall x_1, \ldots, x_n (P(x_1, \ldots, x_n) \rightarrow R(x_1, \ldots, x_n)) \]

- **Projection:**
  \[ \forall x, y, z (P(x, y, z) \rightarrow R(x, y)) \]

- **Decomposition:**
  \[ \forall x, y, z (P(x, y, z) \rightarrow R(x, y) \land T(y, z)) \]

- **Column Augmentation:**
  \[ \forall x, y (P(x, y) \rightarrow \exists z R(x, y, z)) \]

- **Join:**
  \[ \forall x, y, z (E(x, z) \land F(z, y) \rightarrow R(x, y, z)) \]

- **Combinations of the above (e.g., “join + column augmentation”)**
  \[ \forall x, y, z (E(x, z) \land F(z, y) \rightarrow \exists w T(x, y, z, w)) \]
Question: What do all these tasks (copy, projection, decomposition, column augmentation, join) have in common?

Answer:

- They can be specified using **tuple-generating dependencies (tgds)**.

- In fact, they can be specified using a special class of tuple-generating dependencies known as **source-to-target tuple generating dependencies (s-t tgds)**.
Embedded Implicational Dependencies

- **Dependency Theory**: extensive study of constraints in relational databases in the 1970s and 1980s.

- **Embedded Implicational Dependencies**: Fagin, Beeri-Vardi, ... 
  Class of constraints with a balance between high expressive power and good algorithmic properties:
  - **Tuple-generating dependencies** (tgds)
    Inclusion and multi-valued dependencies are a special case.
  - **Equality-generating dependencies** (egds)
    Functional dependencies are a special case.
The relationship between source and target is given by formulas of relational calculus, called

Source-to-Target Tuple Generating Dependencies (s-t tgds)

∀ x (φ(x) → ∃ y ψ(x, y)), where

- φ(x) is a conjunction of atoms over the source;
- ψ(x, y) is a conjunction of atoms over the target;
- x and y are tuples of variables.

Example:

(Student(s) ∧ Enrolls(s,c)) → ∃ t ∃ g (Teaches(t,c) ∧ Grade(s,c,g))

(here, we have dropped the universal quantifiers in front of s-t tgds)
Schema Mapping Specification Language

- s-t tgds assert that: some conjunctive query over the source is contained in some other conjunctive query over the target.

\[(\text{Student (s)} \land \text{Enrolls(s,c)}) \rightarrow \exists t \exists g (\text{Teaches(t,c)} \land \text{Grade(s,c,g)})\]

- s-t tgds generalize the main specifications used in data integration:
  - They generalize LAV (local-as-view) specifications:
    \[P(x) \rightarrow \exists y \psi(x, y), \text{ where } P \text{ is a source schema.}\]
    Note: Copy, projection, and decomposition are LAV s-t tgds.
  - They generalize GAV (global-as-view) specifications:
    \[\varphi(x) \rightarrow R(x), \text{ where } R \text{ is a target relation}\]
    (they are equivalent to full tgds: \(\varphi(x) \rightarrow \psi(x)\),
    where \(\varphi(x)\) and \(\psi(x)\) are conjunctions of atoms).
    Note: Copy, projection, and join are GAV s-t tgds.