TECHNISCHE UNIVERSITÄT DRESDEN

DATABASE THEORY

Lecture 18: Dependencies

Markus Krötzsch

Knowledge-Based Systems

TU Dresden, 2rd July 2019

Review: Databases and their schemas

Lines:

Line	Type
85	bus
3	tram
F1	ferry
...	...

Stops:

SID	Stop	Accessible
17	Hauptbahnhof	true
42	Helmholtzstr.	true
57	Stadtgutstr.	true
123	Gustav-Freytag-Str.	false
...	...	...

Connect:

From	To	Line
57	42	85
17	789	3
...	...	...

Every table has a schema:

- Lines[Line:string, Type:string]
- Stops[SID:int, Stop:string, Accessible:bool]
- Connect[From:int, To:int, Line:string]

Adding constraints

Observation: Even with datatypes, schema information in databases so far is very limited.

Example 18.1: In the public transport example, one would assume that, e.g.,

- **SID** is a key in the Stops table, i.e., no two rows refer to the same stop ID,
- every **Line** mentioned in the Connect table also occurs as a **Line** in the Lines table,

and many more.

Adding constraints

Observation: Even with datatypes, schema information in databases so far is very limited.

Example 18.1: In the public transport example, one would assume that, e.g.,

- **SID** is a key in the Stops table, i.e., no two rows refer to the same stop ID,
- every **Line** mentioned in the Connect table also occurs as a **Line** in the Lines table,

and many more.

Can we express such schema-level information?

$\rightsquigarrow$ Dependencies

Functional dependencies and keys

A common type of simple dependencies are so-called functional dependencies

Definition 18.2: A functional dependency (fd) is an expression Table : $A \to B$, where Table is a relation name, and A and B are sets of attributes of Table. A key is an fd where B is the set of all attributes of Table.

A database instance $\mathcal{I}$ satisfies an fd as above if all tuples in Table$^{\mathcal{I}}$ that agree on the values of all attributes in A also agree on the values of all attributes in B.

Example 18.3: The key in the previous example corresponds to the fd

$$\text{Stops} : \text{SID} \to \text{SID}, \text{Stop}, \text{Accessible}$$

(one usually omits set braces when writing fds).

Inclusion dependencies

Inclusion dependencies can establish relationships between two tables:

Definition 18.4: An inclusion dependency (ind) is an expression Table1$[A]$ $\subseteq$ Table2$[B]$, where Table1 and Table2 are relation names, and A and B are lists of attributes of Table1 and Table2, respectively, that are of equal length.

A database instance $\mathcal{I}$ satisfies an ind as above if, for each tuple $\tau \in$ Table1$^{\mathcal{I}}$, there is a tuple $\tau' \in$ Table2$^{\mathcal{I}}$ such that $\tau[a_i] = \tau'[b_i]$ for all attributes $a_i \in A$ and $b_i \in B$ that occur in corresponding positions in both lists.

Example 18.5: The inclusion in the previous example corresponds to the ind

$$\text{Connect[Line]} \subseteq \text{Lines[Line]}.$$

Why dependencies?

Dependencies have many possible uses:

- Express constraints that a DBMS must guarantee (updates violating constraints fail)
- Improve physical data storage and indexing
- Optimise DB schema, e.g., by normalising tables

> **Example 18.6:** Consider a table Customers[Name,Street,ZIP,City] with key **Name** and another functional dependency Customers : ZIP $\to$ City, which suggests to normalise the table into two tables CustomersNew[Name,Street,ZIP] and Cities[ZIP,City].

- Optimise queries for the special type of databases that satisfy some constraints
- Take background knowledge into account to compare queries (containment, equivalence)
- Query answering under constraints: compute answers under the assumption that the database has been "repaired" to satisfy all constraints
- Construct views over databases for query answering, especially in data integration scenarios

In each case, it can also be helpful to **infer additional dependencies**

Generalisation

Observation Both kinds of dependencies have a logical if-then structure

$\leadsto$ we can vastly generalise these dependencies

Generalisation

Observation Both kinds of dependencies have a logical if-then structure

$\rightsquigarrow$ we can vastly generalise these dependencies

For the following definition, we consider again the unnamed perspective, which is more common for logical expressions:

Definition 18.7: A dependency is a formula of the form

$$\forall \vec{x}, \vec{y}.\varphi[\vec{x}, \vec{y}] \to \exists \vec{z}.\psi[\vec{x}, \vec{z}],$$

where $\vec{x}, \vec{y}, \vec{z}$ are disjoint lists of variables, and φ (the body) and ψ (the head) are conjunctions of atoms using variables $\vec{x} \cup \vec{y}$ and $\vec{x} \cup \vec{z}$, respectively. We allow equality atoms $s \approx t$ to occur in dependencies. The variables $\vec{x}$, which occur in body and head, are known as frontier.

It is common to omit the universal quantifiers when writing dependencies.

Semantically, we will consider dependencies as formulae of first-order logic (with equality).

Equality-generating dependencies generalise fds

An important special type of dependencies are as follows:

Definition 18.8: Equality-generating dependencies (egds) are dependencies with heads of the form $s \approx t$, where s, t are terms, and which do not contain existential qualifiers.

Equality-generating dependencies generalise fds

An important special type of dependencies are as follows:

Definition 18.8: Equality-generating dependencies (egds) are dependencies with heads of the form $s \approx t$, where s, t are terms, and which do not contain existential qualifiers.

We therefore find fds and keys to be special egds:

Observation 18.9: Every fd of the form Table : $A \to B$ can be decomposed into fds of the form Table : $A \to \{b\}$ for each $b \in B$. Such fds can be written as egds.

Example 18.10: Assuming the order of attributes to be as written, the earlier fd Customers : ZIP $\to$ City can be expressed as

$$\text{Customers}(x, y, z, v) \wedge \text{Customers}(x', y', z, v') \to v \approx v'.$$

Tuple-generating dependencies generalise inds

Another important kind of dependencies does not use equality:

Definition 18.11: Tuple-generating dependencies (tgds) are dependencies without equality atoms.

Tuple-generating dependencies generalise inds

Another important kind of dependencies does not use equality:

Definition 18.11: Tuple-generating dependencies (tgds) are dependencies without equality atoms.

We therefore find inds to be special tgds:

Observation 18.12: Every ind can be written as tgd.

Example 18.13: The ind Connect[Line] $\subseteq$ Lines[Line] can be expressed as

$$\text{Connect}(x, y, z) \to \exists v.\text{Lines}(z, v).$$

Full dependencies

Tgds without existential variable are called full tgds; tgds that are not full are sometimes called embedded
(note that this terminology is intuitive when considering inds)

Proposition 18.14: Full tgds can be expressed using several full tgds with only a single head.

Proof: Just write a full tgd $\varphi \to \psi$ as a set of tgds $\{\varphi \to H \mid H \in \psi\}$. □

Full dependencies

Tgds without existential variable are called full tgds; tgds that are not full are sometimes called embedded
(note that this terminology is intuitive when considering inds)

Proposition 18.14: Full tgds can be expressed using several full tgds with only a single head.

Proof: Just write a full tgd $\varphi \to \psi$ as a set of tgds $\{\varphi \to H \mid H \in \psi\}$. □

Example 18.15: Splitting heads into several dependencies is not generally admissible in tgds. For example, the tgd uncle$(x, y) \to \exists z.$child$(x, z) \wedge$ brother(z, y) entails the set $\{$uncle$(x, y) \to \exists z.$child(x, z), uncle$(x, y) \to \exists z.$brother$(z, y)\}$, but not vice versa.

Remark: In general, heads of tgds can still be decomposed to some extent, as long as each existential quantifier (and all occurrences of the bound variable) remains in one rule (sets of atoms connected by sharing existential variables have been called pieces).

Reasoning with dependencies

Reasoning tasks (1)

Some of the main reasoning tasks for working with dependencies are as follows:

Dependency implication:
Input: sets of dependencies Σ and $\{\sigma\}$
Output: "yes" if $\Sigma \models \sigma$; "no" otherwise

CQ containment under constraints:
Input: set of dependencies Σ and CQs q_1 and q_2
Output: "yes" if every answer to q_1 is also an answer to q_2 in every database $\mathcal{I}$ with $\mathcal{I} \models \Sigma$; "no" otherwise

For both reasoning tasks we consider only database instances that satisfy the given constraints.

Note: We may consider reasoning problems to refer only to finite databases, or to generalised (possibly infinite) databases. In general, this does not lead to the same results, so it has to be defined. Unless otherwise stated, we always allow for infinite models/databases here.

Reasoning tasks (2)

BCQ entailment under constraints:
Input: set of dependencies Σ, database instance $\mathcal{I}$, and boolean CQ q
Output: "yes" if every extension $\mathcal{I}' \supseteq \mathcal{I}$ with $\mathcal{I}' \models \Sigma$ also satisfies $\mathcal{I}' \models q$; "no" otherwise

Alternatively, we can also view $\mathcal{I}$ as a (syntactic) set of ground facts and ask if $\mathcal{I} \cup \Sigma \models q$ (a first-order logic entailment problem)

As usual, BCQ entailment is the decision-problem version for CQ answering

Note: Again, we allow for the extension $\mathcal{I}'$ to be infinite in general.

Reducing reasoning tasks

Theorem 18.16: Dependency implication, CQ containment under constraints, and BCQ entailment under constraints are equivalent problems.

Proof: Consider a set Σ of dependencies.

Reducing reasoning tasks

Theorem 18.16: Dependency implication, CQ containment under constraints, and BCQ entailment under constraints are equivalent problems.

Proof: Consider a set Σ of dependencies.

- For CQs q_1 and q_2, containment under constraints corresponds to the implication of a dependency $q_1 \to q_2$ by Σ.

Theorem 18.16: Dependency implication, CQ containment under constraints, and BCQ entailment under constraints are equivalent problems.

Proof: Consider a set Σ of dependencies.

- For CQs q_1 and q_2, containment under constraints corresponds to the implication of a dependency $q_1 \to q_2$ by Σ.
- For a dependency $\sigma = \varphi[\vec{x}, \vec{y}] \to \exists\vec{z}.\psi[\vec{x}, \vec{z}]$, let $\mathcal{I}_\varphi$ be the database instance corresponding to the CQ φ, obtained using a substitution θ that replaces each variable v in $\vec{x} \cup \vec{y}$ by a fresh constant c_v; then σ is entailed by Σ iff $\mathcal{I}_\varphi, \Sigma \models \exists\vec{z}.\psi\theta$.

Reducing reasoning tasks

Theorem 18.16: Dependency implication, CQ containment under constraints, and BCQ entailment under constraints are equivalent problems.

Proof: Consider a set Σ of dependencies.

- For CQs q_1 and q_2, containment under constraints corresponds to the implication of a dependency $q_1 \to q_2$ by Σ.
- For a dependency $\sigma = \varphi[\vec{x}, \vec{y}] \to \exists \vec{z}.\psi[\vec{x}, \vec{z}]$, let $\mathcal{I}_\varphi$ be the database instance corresponding to the CQ φ, obtained using a substitution θ that replaces each variable v in $\vec{x} \cup \vec{y}$ by a fresh constant c_v; then σ is entailed by Σ iff $\mathcal{I}_\varphi, \Sigma \models \exists \vec{z}.\psi\theta$.
- A BCQ q is entailed by a database instance $\mathcal{I}$ under constraints Σ iff query $t()$ is contained in q under constraints $\Sigma \cup \{t()\} \cup \{p(c_1, \ldots, c_n) \mid \langle c_1, \ldots, c_n \rangle \in p^{\mathcal{I}}\}$, where t is a fresh nullary predicate. $\square$

Note: In the last case, we use rules that contain ground heads but no body to express facts.

Reasoning is undecidable

The following should not come as a surprise:

Theorem 18.17: Query entailment under a set of tgd constraints is undecidable.

Reasoning is undecidable

The following should not come as a surprise:

Theorem 18.17: Query entailment under a set of tgd constraints is undecidable.

Proof (sketch): This can be shown by direct encoding of a deterministic TM in query entailment, similar to our proof of Trakhtenbrot's Theorem:

- Most axioms used there already are in the form of tgds
- Negative information in consequences of first-order implications are handled by transformation:
  $\varphi \to \neg H_1 \vee \neg H_2$ becomes $\varphi \wedge H_1 \wedge H_2 \to \mathtt{match}()$
- We can avoid equality by axiomatising its effects
- Halting of the TM can be recognised by a rule that implies $\mathtt{match}()$ in this case (to make it easy to recognise, we can transform the TM to have a unique halting state)

Checking entailment of BCQ $\mathtt{match}()$ then corresponds to checking if the TM halts. $\square$

Datalog and full tgds

Full tgds are closely related to Datalog:

- Syntactically, both types of rules are the same (we use $\leftarrow$ for Datalog instead of $\rightarrow$)
- Semantically, Datalog query answers (second-order model checking) correspond to entailments of the corresponding full tgds (first-order entailment)

The boundaries between both perspectives are blurred in modern works, in particular since tgds are increasingly used to define (query) views.
(an EDB/IDB distinction is sometimes made for tgds as well)

Datalog and full tgds

Full tgds are closely related to Datalog:

- Syntactically, both types of rules are the same (we use $\leftarrow$ for Datalog instead of $\rightarrow$)
- Semantically, Datalog query answers (second-order model checking) correspond to entailments of the corresponding full tgds (first-order entailment)

The boundaries between both perspectives are blurred in modern works, in particular since tgds are increasingly used to define (query) views.
(an EDB/IDB distinction is sometimes made for tgds as well)

From what we learned about Datalog, we conclude:

Theorem 18.18: For full tgds, dependency implication, as well as CQ containment and BCQ entailment under constraints is decidable and ExpTime-complete.

Note: We take a strict first-order view here, and use the reduction of Theorem 18.16. BCQ entailment can be decided, e.g., using bottom-up computation.

Summary and Outlook

Dependencies have many uses in database theory and practice

Most practical forms of dependencies can be captured in logical form, with the two most common general cases being

- Tuple-generating dependencies (tgds)
- Equality generating dependencies (egds)

There are several reasoning problems for dependencies, which are all equivalent (and generally undecidable)

Next topics:

- The Chase
- Languages of tgds with decidable entailment problems
- Outlook