course:

Database Systems (A7B36DBS)

lecture 9:

Relational design – algorithms

Doc. RNDr. Irena Holubova, Ph.D.

Acknowledgement:
The slides were kindly lent by Doc. RNDr. Tomas Skopal, Ph.D.,
Department of Software Engineering, Charles University in Prague

Today's lecture outline

- schema analysis
  - basic algorithms (attribute closure, FD membership and redundancy)
  - determining the keys
  - testing normal forms
- normalization of universal schema
  - decomposition (to BCNF)
  - synthesis (to 3NF)

Attribute closure

functional dependency

- closure X^+ of attribute set X according to FD set F
  - principle: we iteratively derive all attributes „F-determined" by attributes in X
  - complexity O(m*n), where ***n*** is the number of attributes and ***m*** is number of FDs

Left-hand side of FD

Right-hand side of FD

```
algorithm AttributeClosure(set of dependencies F, set of attributes X) :
   returns set X+
   ClosureX := X; DONE := false; m = |F|;
   while not DONE do
        DONE := true;
        for i := 1 to m do
              if (LS[i] ⊆ ClosureX and RS[i] ⊄ ClosureX) then
                    ClosureX := ClosureX ∪ RS[i];
                    DONE := false;
              endif
        endfor
   endwhile
   return ClosureX;
```

The trivial FD is used (algorithm initialization) and then transitivity (test of left-hand side in the closure). The composition and decomposition usage is hidden in the inclusion test.

Example – attribute closure

$F = \{a \to b, bc \to d, bd \to a\}$

$\{b,c\}^+ = ?$

1. $ClosureX := \{b,c\}$ (initialization)
2. $ClosureX := ClosureX \cup \{d\} = \{b,c,d\}$ ($bc \to d$)
3. $ClosureX := ClosureX \cup \{a\} = \{a,b,c,d\}$ ($bd \to a$)

$\{b,c\}^+ = \{a,b,c,d\}$

Membership test

- we often need to check <u>if a FD $X \rightarrow Y$ belongs to F^+</u>, i.e., to solve the problem $\{X \rightarrow Y\} \in F^+$
- materializing F^+ is not practical, we can employ the attribute closure

```
algorithm IsDependencyInClosure(set of FDs F,
                                FD X → Y)
   return Y ⊆ AttributeClosure(F, X);
```

Redundancy testing

The membership test can be easily used when testing <u>redundancy</u> of

- FD X → Y in F
- attribute *a* in X (according to F and X → Y)

```
algorithm IsDependencyRedundant(set of FDs F, FD X → Y ∈ F)
   return IsDependencyInClosure(F – {X → Y}, X → Y);

algorithm IsAttributeRedundant(set of FDs F, FD X → Y ∈ F, attr. a ∈ X)
   return IsDependencyInClosure(F, X – {a} → Y);
```

In the following slides we find useful the algorithm for <u>reduction</u> of the left-hand side of a FD:

```
algorithm ReduceAttributes(set of FDs F, FD X → Y ∈ F)
   X' := X;
   for each a ∈ X do
        if IsAttributeRedundant(F, X' → Y, a) then X' := X' – {a};
   endfor
return X';
```

Minimal cover

- for all FDs we <u>test redundancies and remove them</u>

```
algorithm GetMinimumCover(set of dependencies F)
   : returns minimal cover G
   decompose each FD in F into elementary FDs
   for each X → Y in F do
       F :=    (F –
               {X → Y}) ∪
               {ReduceAttributes(F, X → Y) → Y};
   endfor
   for each X → Y in F do
       if IsDependencyRedundant(F, X → Y)
        then F := F – {X → Y};
   endfor
return F;
```

removing redundant attributes

removing redundant FDs

Determining (first) key

- the algorithm for attribute redundancy testing could be used directly for <u>determining a key</u>
- redundant attributes are iteratively removed from left-hand side of trivial FD A $\rightarrow$ A

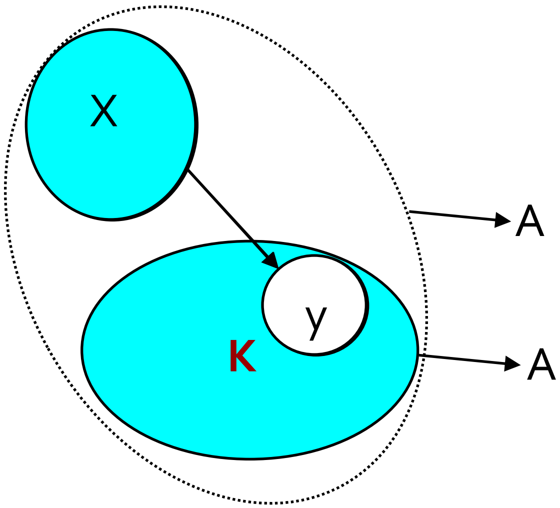
```
algorithm GetFirstKey(set of deps. F, set of attributes A)
  : returns a key K;
  return ReduceAttributes(F, A → A);
```

<u>Note:</u> Because multiple keys can exists, the algorithm finds only one of them.

Which one? It depends on the traversing of the attribute set within the algorithm `ReduceAttributes`.

Determining all keys, the principle

Let us have a schema S(A, F).
Simplify *F* to minimal cover.



1. Find any key **K** (see the previous slide).
2. Take a FD **X** → **y** in *F* such that **y** ∈ **K** or terminate if not exists (there is no other key).
3. Because **X** → **y** and **K** → **A**, it transitively holds also **X**{**K** – **y**} → **A**, i.e., **X**{**K** – **y**} is super-key.
4. Reduce FD **X**{**K** – **y**} → **A** so we obtain key **K'** on the left-hand side.
   This key is surely different from **K** (we removed **y**).
5. If **K'** is not among the determined keys so far, we add it, declare **K**=**K'** and continue from step 2.
   Otherwise we finish.

Determining all keys, the algorithm

- Formally: **Lucchesi-Osborn algorithm**
  - having an already determined key, we search for equivalent sets of attributes, i.e., other keys
- NP-complete problem (theoretically exponential number of keys/FDs)

```
algorithm GetAllKeys(set of FDs F, set of attr. A)
   : returns set of all keys Keys;
   let all dependencies in F be non-trivial
   K := GetFirstKey(F, A);
   Keys := {K};
   for each K in Keys do
      for each X → Y in F do
         if (Y ∩ K ≠ ∅ and ¬∃K' ∈ Keys : K' ⊆ (K ∪ X) – Y) then
            N := ReduceAttributes(F, ((K ∪ X) – Y) → A);
            Keys := Keys  ∪ {N};
         endif
      endfor
   endfor
return Keys;
```

Example – determining all keys

Contracts(A, F)

A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId,
p = PartId, q = Quantity, v = Value}

F = {**c** → *all*, **sd** → **p**, **p** → **d**, **jp** → **c**, **j** → **s**}

1. Determine the first key – Keys = {**c**}
2. Iteration 1: take **jp** → **c** that has a part of the last key on the right-hand side (in this case the whole key – **c**) and **jp** is not a super-set of already determined key
3. **jp** → *all* is reduced (no redundant attribute), i.e.,
   Keys = {**c, jp**}
4. Iteration 2: take **sd** → **p** that has a part of the last key on the right-hand side (**jp**),
   {**jsd**} is not a super-set of **c** nor **jp**, i.e., it is a key candidate
5. in **jsd** → *all* we get redundant attribute **s**, i.e.,
   Keys = {**c, jp, jd**}
6. Iteration 3: take **p** → **d**, however, **jp** was already found so we do not add it
7. Finish as the iteration 3 resulted in no key addition.

Testing normal forms

- NP-complete problem
  - **we must know all keys** – then it is sufficient to test a FD in F, so we do not need to materialize F^+
  - or, just one key needed, but also needing extension of F to F^+
- fortunately, in practice determination of keys is fast
  - thanks to limited size of F and „separability" of FDs

Design of database schemas

Two ways of modelling a relational database:

1. we get a set of relational schemas (as either direct relational design or conversion from conceptual model)
   - normalization performed separately on each table
   - the database could get unnecessarily highly "granularized" (too many tables)

2. considering the whole database as a bag of (global) attributes results in a single universal database schema – i.e., one big table + a single set of FDs
   - normalization performed on the universal schema
   - less tables (better „granulating")
   - „classes/entities" are generated (recognized) as the consequence of FD set

- both approaches could be combined – i.e.,
  - create a conceptual database model
  - convert it to relational schemas
  - merge and/or normalize some of the schemas

Relational schema normalization

- just one way – <u>decomposition</u> to multiple schemas
  - or merging some „abnormal" schemas and then decomposition
- different criteria
  - **data integrity preservation**
    - **lossless join**
    - **dependency preserving**
  - requirement on normal form (3NF or BCNF)
- manually or algorithmically

Why to preserve integrity?

If the decomposition is not limited, we can decompose the table into several single-column ones that surely are all in BCNF.

Company	HQ	Altitude
Sun	Santa Clara	25 m
Oracle	Redwood	20 m
Microsoft	Redmond	10 m
IBM	New York	15 m

Company,
HQ → Altitude

⇒

Company
Sun
Oracle
Microsoft
IBM

Company

HQ
Santa Clara
Redwood
Redmond
New York

HQ

Altitude
25 m
20 m
10 m
15 m

Altitude

Clearly, there is something wrong with such a decomposition…

…it is **lossy** and it does not **preserve dependencies**

Lossless join

- a property of decomposition that **ensures correct** joining (**reconstruction**) of the universal relation from the decomposed ones

used in algorithm

- Definition 1:
  Let $R(\{X \cup Y \cup Z\}, F)$ be universal schema, where $Y \rightarrow Z \in F$.
  Then decomposition $R_1(\{Y \cup Z\}, F_1)$, $R_2(\{Y \cup X\}, F_2)$ is lossless.

- Alternative Definition 2:
  Decomposition of $R(A, F)$ into $R_1(A_1, F_1)$, $R_2(A_2, F_2)$ is lossless, if $A_1 \cap A_2 \rightarrow A_1$ or $A_2 \cap A_1 \rightarrow A_2$

used for checking

projection

- **Alternative Definition 3:**
  Decomposition of $R(A, F)$ into $R_1(A_1, F_1)$, ..., $R_n(A_n, F_n)$ is lossless, if $R = *_{i=1..n} R_i[A_i]$.

natural join

Example – lossy decomposition

Company	Uses DBMS	Data managed
Sun	Oracle	50 TB
Sun	DB2	10 GB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

Company, Uses DBMS

Company	Uses DBMS
Sun	Oracle
Sun	DB2
Microsoft	MSSQL
Microsoft	Oracle

Company, Uses DBMS

Company	Data managed
Sun	50 TB
Sun	10 GB
Microsoft	30 TB

Company, Data managed

„reconstruction“
(natural join)

Company	Uses DBMS	Data managed
Sun	Oracle	50 TB
Sun	Oracle	10 GB
Sun	DB2	10 GB
Sun	DB2	50 TB
Microsoft	MSSQL	30 TB
Microsoft	Oracle	30 TB

Company, Uses DBMS, Data managed

Example – lossless decomposition

Company	HQ	Altitude
Sun	Santa Clara	25 m
Oracle	Redwood	20 m
Microsoft	Redmond	10 m
IBM	New York	15 m

Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

HQ	Altitude
Santa Clara	25 m
Redwood	20 m
Redmond	10 m
New York	15 m

<u>Company</u>,
HQ $\rightarrow$ Altitude

<u>Company</u>

<u>HQ</u>

„reconstruction"
(natural join)

Dependency preserving

- a decomposition property that ensures **no FD will be lost**
- Definition:
  Let $R_1(A_1, F_1)$, $R_2(A_2, F_2)$ be decomposition of $R(A, F)$. Then such decomposition preserves dependencies if $F^+ = (\cup_{i=1..n} F_i)^+$.
- Dependency preserving could be violated in two ways
  - during decomposition of F we do not derive all valid FDs – we lose FD that should be preserved **in a particular schema**
  - even if we derive all valid FDs (i.e., we perform projection of F^+), we may lose a FD that is valid **across the schemas**

Example – dependency preserving

Company	HQ	Altitude
Sun	Santa Clara	25 m
Oracle	Redwood	20 m
Microsoft	Redmond	10 m
IBM	New York	15 m

<u>Company</u>,
HQ $\rightarrow$ Altitude

dependencies not preserved, we lost
HQ $\rightarrow$ Altitude

Company	Altitude
Sun	25 m
Oracle	20 m
Microsoft	10 m
IBM	15 m

<u>Company</u>

Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

<u>HQ</u>

dependencies
preserved

Company	HQ
Sun	Santa Clara
Oracle	Redwood
Microsoft	Redmond
IBM	New York

<u>Company</u>

HQ	Altitude
Santa Clara	25 m
Redwood	20 m
Redmond	10 m
New York	15 m

<u>HQ</u>

The "Decomposition" algorithm

- algorithm for decomposition into BCNF, **preserving lossless join**
- may **not preserve dependencies**
  - not an algorithm property – sometimes we simply cannot decompose into BCNF with all FDs preserved

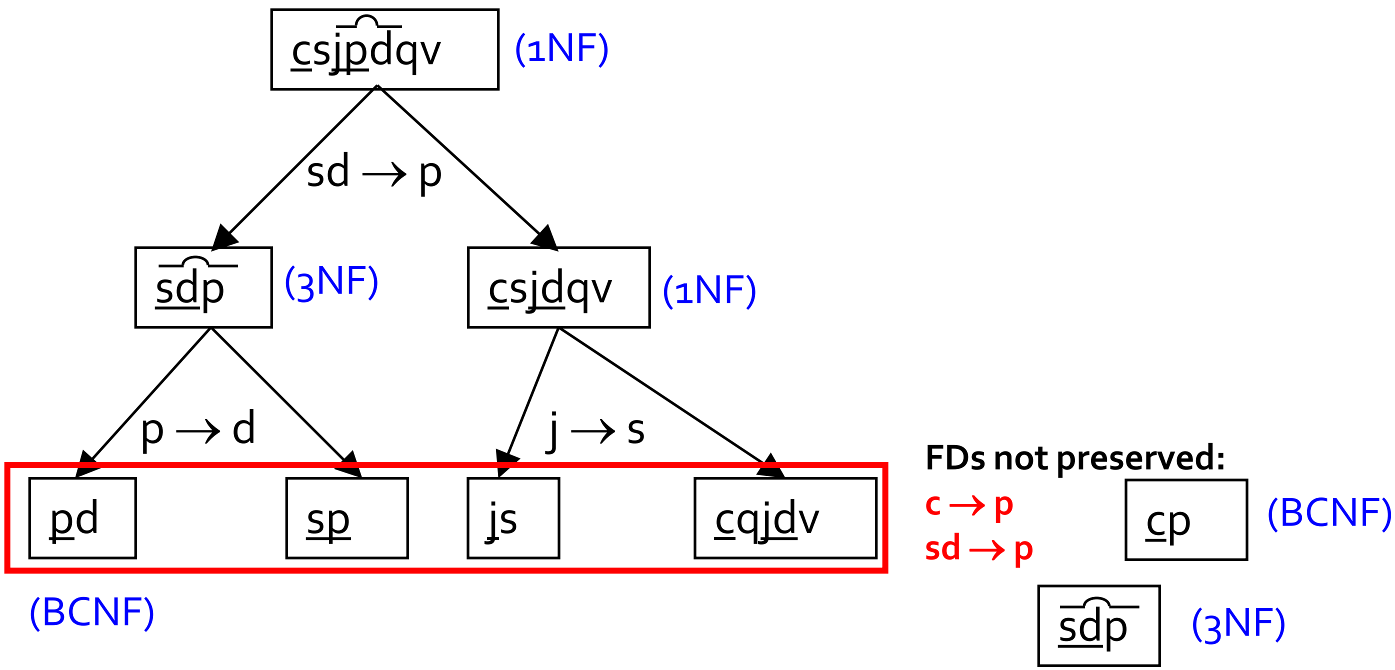
algorithm ***Decomposition***(set of elem. FDs. F, set of attributes A) : **returns set** $\{R_i(A_i, F_i)\}$
 Result := {R(A, F)};
 Done := **false**;
 Create F^+;
 while not Done **do**
  if $\exists R_i(F_i, A_i) \in$ Result not being in BCNF **then** *// if there is a schema in the result violating BCNF*
   Let $X \rightarrow Y \in F_i$ such that $X \rightarrow A_i \notin F^+$. *// X is not (super-)key and so $X \rightarrow Y$ violates BCNF*
   Result := (Result – $\{R_i(A_i, F_i)\}$) $\cup$ *// we remove the schema being decomposed*
   $\{R_i(A_i - Y$, cover($F, A_i - Y$))} $\cup$ *// we add the schema being decomposed without attributes Y*
   $\{R_j(X \cup Y$, cover($F, X \cup Y$))} *// we add the schema with attributes XY*
  else
   Done := **true**;
  endwhile
 return Result;

This partial decomposition on two tables is **lossless**, we get two schemas that both contain X, while the second one contains also Y and it holds $X \rightarrow Y$.
X is now in the second table a super-key and $X \rightarrow Y$ is no more violating BCNF (in the first table there is not Y anymore).

<u>Note:</u> Function cover(X, F) returns all FDs valid on attributes from X, i.e., a subset of F^+ that contains only attributes from X. Therefore it is necessary to compute F^+.

Example – decomposition

Contracts(A, F)
A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId, q = Quantity, v = Value}
F = {c → all, sd → p, p → d, jp → c, j → s}



The "Synthesis" algorithm

- algorithm for decomposition into 3NF, **preserving dependencies**
  - basic version **not preserving lossless joins**

algorithm ***Synthesis***(set of elem. FDs F, set of attributes A) : **returns set** $\{R_i(F_i, A_i)\}$
    G = minimal cover of F
    compose FDs having equal left-hand side into a single FD
    every composed FD forms a scheme R_i (A_i, F_i) of decomposition
return $\cup_{i=1..n}\{R_i\ (A_i, F_i)\}$

- lossless joins can be preserved by adding another schema into the decomposition that contains ***universal key***
  - i.e., a key from the original universal schema
- a schema in decomposition that is a subset of another one can be deleted
- we can try to merge schemas that have functionally equivalent keys, but such an operation can violate 3NF (or BCNF if achieved)!
  - i.e., we can try to minimize the number of relations

Example – synthesis

Contracts(A, F)

A = {c = ContractId, s = SupplierId, j = ProjectId, d = DeptId, p = PartId,
q = Quantity, v = Value}

F = {$c \to sjdpqv$, $sd \to p$, $p \to d$, $jp \to c$, $j \to s$}

Minimal cover:

- There are no redundant attributes in FDs.
- Reundant FDs $c \to s$ and $c \to p$ were removed.
- G = {$c \to j$, $c \to d$, $c \to q$, $c \to v$, $sd \to p$, $p \to d$, $jp \to c$, $j \to s$}

Composition:

- G' = {$c \to jdqv$, $sd \to p$, $p \to d$, $jp \to c$, $j \to s$}

Result:

- R_1({cqjdv}, {$c \to jdqv$}), R_2({sdp}, {$sd \to p$}), ~~R_3({pd}, {$p \to d$}),~~ R_4({jpc}, {$jp \to c$}), R_5({js}, {$j \to s$}))

(subset of R_2)

Equivalent keys: {c, jp, jd}

R_1({cqjpdv}, {$c \to jdqv$, $jp \to c$}), R_2({sdp}, {$sd \to p$, $p \to d$}), R_5({js}, {$j \to s$}))

merging R_1 and R_4
(however, now $p \to d$ violates BCNF)