Boyce-Codd Normal Form (BCNF) Decomposition (III)

- Algorithm for BCNF decomposition

  void CalculateBCNFDecomposition(R, F, D)
  // Input: A relation schema R and a set F of FDs on R
  // Input/Output parameter: A decomposition $D = \{R_1, \ldots, R_n\}$ of R

  // Check whether R is already in BCNF
  if *RelationSchemaIsInBCNF*(R, F, f) **then return**

  // Let $f = X \rightarrow Y$ be the violating FD. Note that always $X \cup Y \subseteq R$ holds. We
  // conclude that also $X \rightarrow X^+$ is a violating FD since X is not a superkey of R.
  X^+ := *CalculateAttributeClosure*(F, X)

  // Decompose R into R_1 and R_2: The violating FD (extended to X^+ for
  // performance reasons) is put into an own schema R_1. The remaining
  // attributes including X are put into R_2. R_1 and R_2 only share the attributes in X.
  $R_1 := X^+$
  $R_2 := X \cup (R - X^+)$

Boyce-Codd Normal Form (BCNF) Decomposition (IV)

- Algorithm for BCNF decomposition (*continued*)

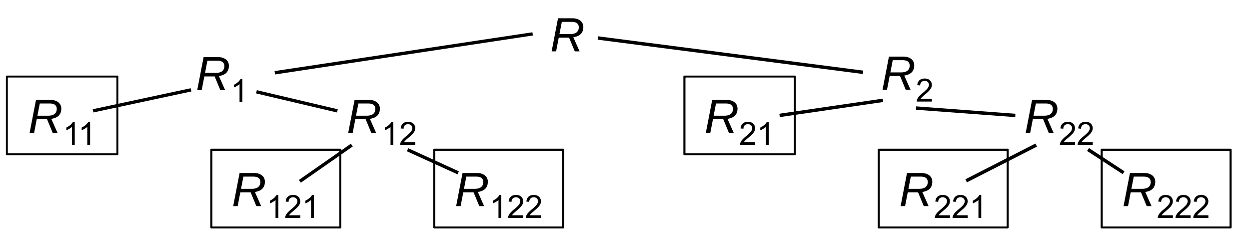
  // Update the decomposition to indicate that R is replaced by R_1 and R_2
  $D := D - \{R\} \cup \{R_1, R_2\}$

  // Compute the restrictions F_1 of F for R_1 and F_2 of F for R_2
  $F_1 := \mathrm{FD}\mathit{Restriction}(F, R_1)$
  $F_2 := \mathit{FDRestriction}(F, R_2)$

  // Recursively decompose R_1 and R_2, and update D if needed
  $\mathit{CalculateBCNFDecomposition}(R_1, F_1, D)$
  $\mathit{CalculateBCNFDecomposition}(R_2, F_2, D)$

- The decomposition D is recursively computed by splitting all non-BCNF relation schemas and updating D; the leaf nodes of the recursion tree represent the decomposition D of R into BCNF relation schemas

$D = \{R\} \Rightarrow D = \{R_1, R_2\} \Rightarrow D = \{R_{11}, \mathrm{R}_{12}, R_2\} \Rightarrow D = \{R_{11}, \mathrm{R}_{121}, \mathrm{R}_{122}, R_2\} \Rightarrow D = \{R_{11}, \mathrm{R}_{121}, \mathrm{R}_{122}, R_{21}, R_{22}\} \Rightarrow D = \{R_{11}, \mathrm{R}_{121}, \mathrm{R}_{122}, R_{21}, R_{221}, R_{222}\}$



Boyce-Codd Normal Form (BCNF) Decomposition (V)

- The BCNF decomposition algorithm is here not implemented as a function but as a procedure where the decomposition *D* is assumed to be a reference argument of the procedure in order to be able to have the most recent version of *D* available in the recursion and to update *D* in the recursion if needed
- Given a relation schema *R,* a set *F* of FDs on *R*, and a decomposition *D* that is initialized with *D* = {*R*}, we call the BCNF decomposition algorithm by

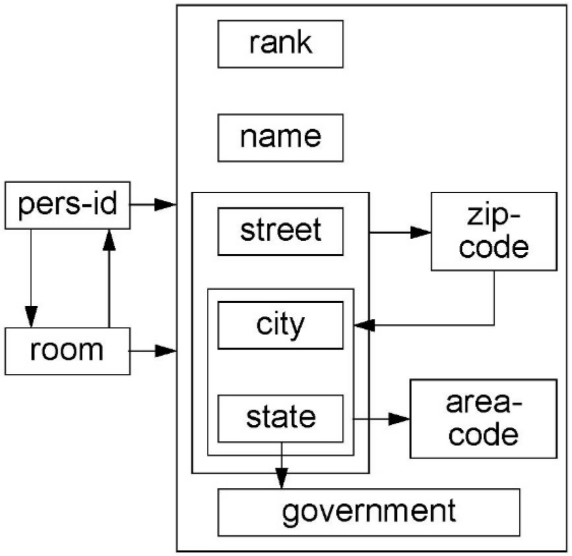
  CalculateBCNFDecomposition(*R*, *F*, *D*)
- An iterative version that would allow us to call the algorithm by

  D := *CalculateBCNFDecomposition*(*R*, *F*)

  is, of course, also possible but would be lengthier in its description

Boyce-Codd Normal Form (BCNF) Decomposition (VI)

- Example: Universal relation schema R(pers-id (A), name (B), rank (C), room(D), city (E), street (F), zipcode (G), area-code (H), state (I), government (J)) with $F = \{A \rightarrow BCDEFGHIJ, D \rightarrow ABCEFGHIJ, EFI \rightarrow G, EI \rightarrow H, I \rightarrow J, G \rightarrow EI\}$

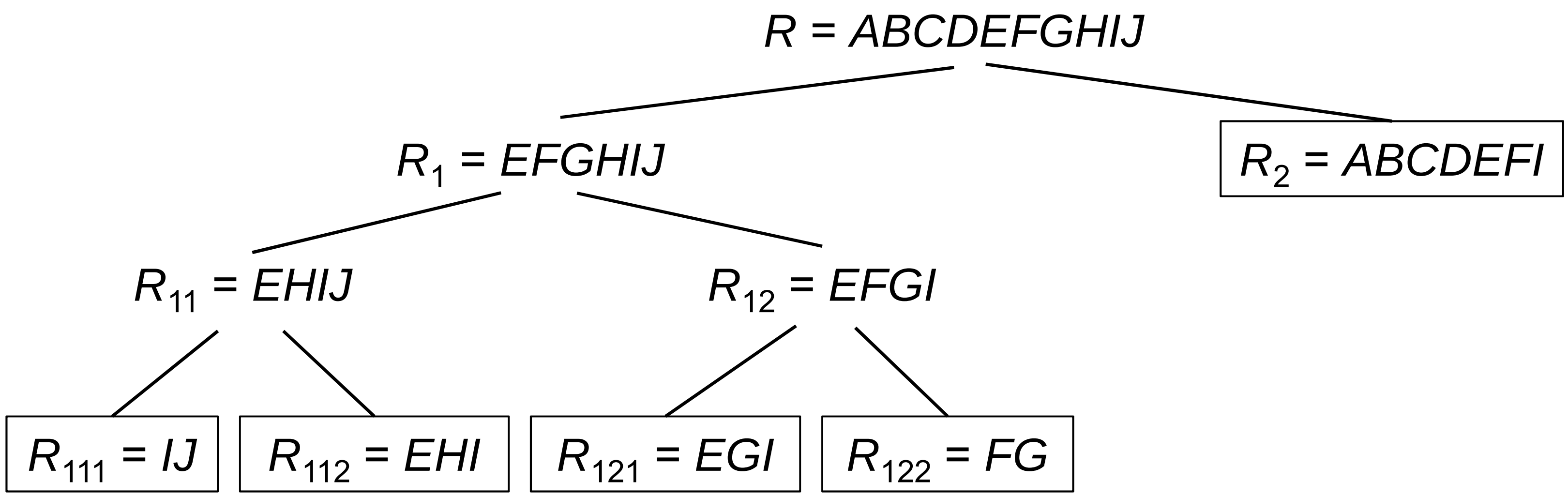


- R is not in the BCNF: $A^+ = R$, $D^+ = R$, first violating FD is $EFI \rightarrow G$ since EFI is not a superkey ($EFI^+ = EFGHIJ \neq R$)
- Split R into $R_1 = EFGHIJ$ and $R_2 = ABCDEFI$
- Compute the restrictions F_1 and F_2. We obtain $F_1 = \{EFI \rightarrow GHIJ, EI \rightarrow HJ, I \rightarrow J, G \rightarrow EHIJ\}$ and $F_2 = \{A \rightarrow BCDEFI, D \rightarrow ABCEFI\}$
- R_1 is not in the BCNF: $EFI^+ = EFGHIJ = R_1$, first violating FD is $EI \rightarrow HJ$ since EI is not a superkey ($EI^+ = EHIJ \neq R_1$)
- Split R_1 into $R_{11} = EHIJ$ and $R_{12} = EFGI$ and compute the restrictions F_{11} and F_{12}. We obtain $F_{11} = \{EI \rightarrow HJ, I \rightarrow J\}$ and $F_{12} = \{EFI \rightarrow G, G \rightarrow EI\}$
- R_{11} is not in the BCNF: $EI^+ = EHIJ$, the violating FD is $I \rightarrow J$ since I is not a superkey ($I^+ = IJ \neq R_{11}$)

Boyce-Codd Normal Form (BCNF) Decomposition (VII)

- Split R_{11} into $R_{111} = IJ$ and $R_{112} = EHI$, and compute the restrictions F_{111} and F_{112}. We obtain $F_{111} = \{I \rightarrow J\}$ and $F_{112} = \{EI \rightarrow H\}$
- R_{111} is in the BCNF since $IJ^+ = R_{111}$ holds and no other FD can be applied to R_{111}. R_{112} is in the BCNF since $EI^+ = R_{112}$ holds and no other FD can be applied to R_{112}.
- R_{12} is not in the BCNF: $EFI^+ = EFGI$, the violating FD is $G \rightarrow EI$ since G is not a superkey ($G^+ = EGI \neq R_{12}$)
- Split R_{12} into $R_{121} = EGI$ and $R_{122} = FG$, and compute the restrictions F_{121} and F_{122}. We obtain $F_{121} = \{G \rightarrow EI\}$ and $F_{122} = \varnothing$ (no nontrivial FDs, FG is the key)
- R_{121} is in the BCNF since $G^+ = R_{121}$ holds and no other FD can be applied to R_{121}. R_{122} is in the BCNF since any two-attribute relation schema is in the BCNF (see below).
- R_2 is in the BCNF since $A^+ = R_2$ holds, $D^+ = R_2$ holds, and no other FD can be applied to R_2.
- The decomposition D of R is $D = \{R_{111}, R_{112}, R_{121}, R_{122}, R_2\} = \{IJ, EHI, EGI, FG, ABCDEFI\}$

Boyce-Codd Normal Form (BCNF) Decomposition (VIII)



- Applying the predicate *IsDependencyPreserving2* reveals that the decomposition *D* is *not* dependency preserving under *F* (not shown here)

- Using the original attributes, we obtain *D* = {{state, government}, {city, area-code, state}, {city, zipcode, state}, {street, zipcode}, {pers-id, name, rank, room, city, street, state}}

Boyce-Codd Normal Form (BCNF) Decomposition (IX)

- Any two-attribute relation schema $R = \{X, Y\}$ is in the BCNF
  - The BCNF decomposition algorithm terminates since every time we split a relation schema R, the two resulting schemas each have fewer attributes than R
  - When we arrive at two attributes for a resulting relation schema, the schema is surely in the BCNF
  - We consider the possible cases:
    - There are no nontrivial FDs. Then the BCNF condition holds because only a nontrivial FD can violate this condition. $\{X, Y\}$ is the only key here.
    - $X \rightarrow Y$ holds but $Y \rightarrow X$ does not hold. Then X is the only key, and each nontrivial FD can only have X on the left-hand side. Thus there is no violation of the BCNF condition.
    - $Y \rightarrow X$ holds but $X \rightarrow Y$ does not hold. This is symmetric to the previous case.
    - Both $X \rightarrow Y$ and $Y \rightarrow X$ hold. Then both X and Y are keys. Any FD has at least one of them on the left-hand side. Thus there can be no violation of the BCNF condition.

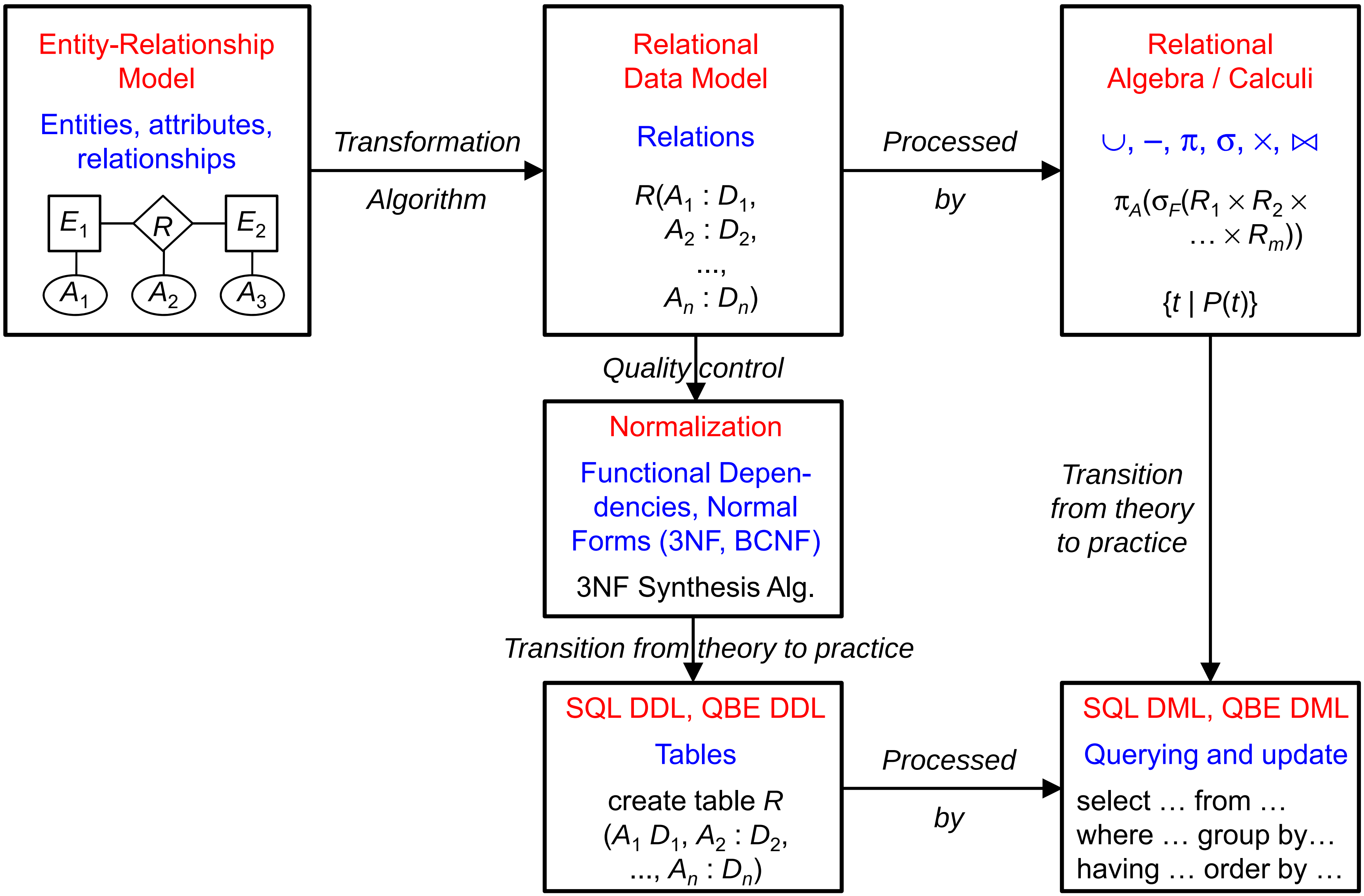
Boyce-Codd Normal Form (BCNF) Decomposition (X)

- Recommendation to normalize a universal relation schema
  - Normalize into a 3NF decomposition by using the 3NF synthesis algorithm to obtain the properties of lossless join decomposition and dependency preservation
  - Frequently, the result is not only in the 3NF but also in the BCNF
  - If not, further decompose any schema in the 3NF decomposition that is not in the BCNF by using the BCNF decomposition algorithm
  - If the result is not dependency-preserving, revert to the 3NF design

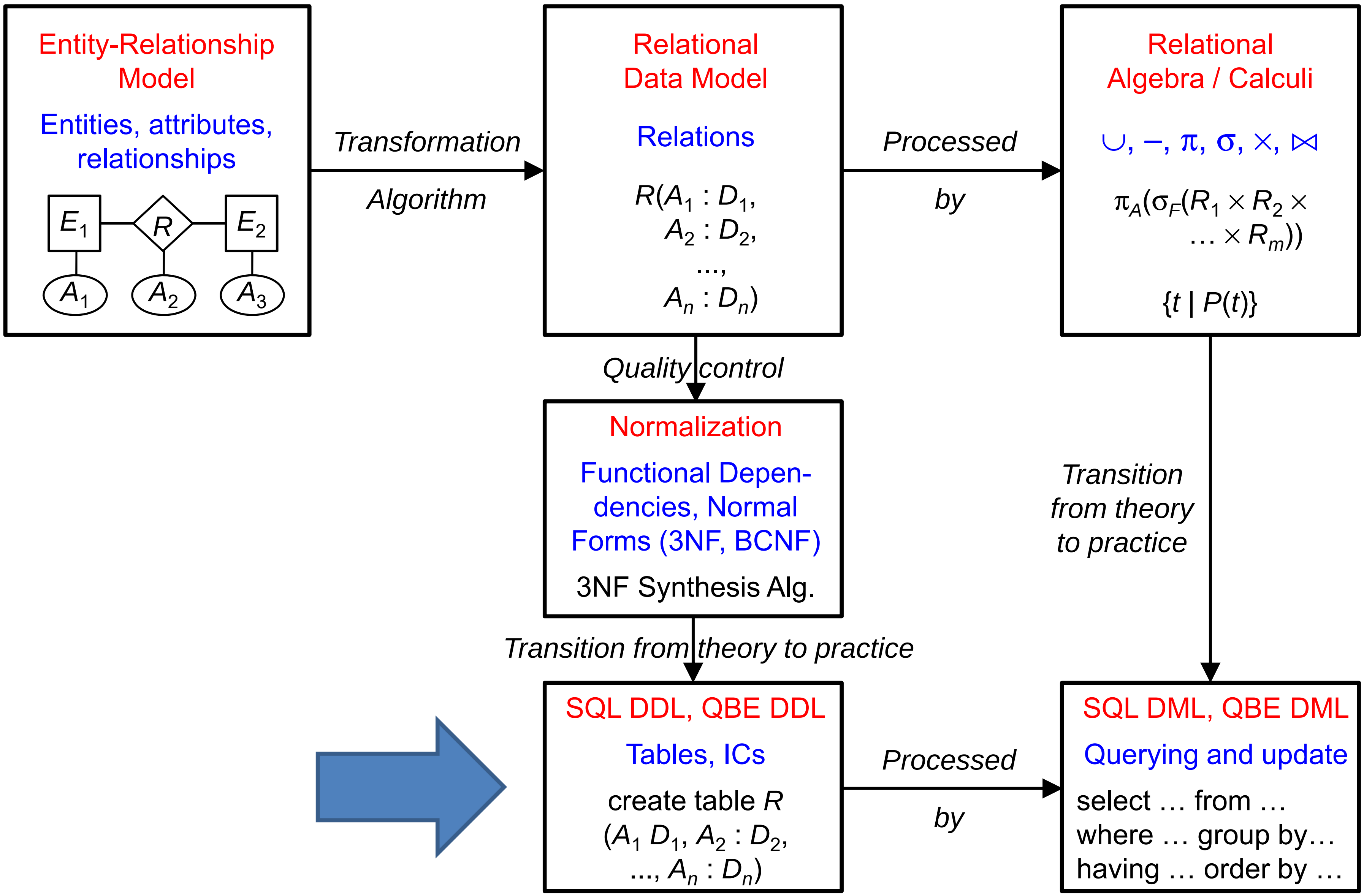
- Other recommendations
  - Since the 3NF decomposition process is not unique, compute all 3NF decompositions based on all different minimal covers, and select one which satisfies the properties that the decomposition has a small number of schemas, all schemas are in the BCNF, all schemas together have a small number of attributes, etc.
  - Such efforts are worthwhile to do since the goal is to ensure a high quality of the database schema

Integrity Constraints (ICs)

The Big Picture (I)



The Big Picture (II)



Introduction (I)

- Integrity constraints (ICs)
  - are an instrument to ensure that changes of the database by authorized users do not lead to a loss of data consistency
  - protect against accidental damage to the database
  - serve for a restriction of the database states to those ones that really exist in the real world
  - are derived from the rules in the miniworld that the database represents
  - are imposable on the underlying data model (and not the data) and can therefore already be specified during the creation of a schema

- Advantages
  - Consistency conditions are specified only once
  - Consistency conditions are checked automatically by the DBMS
  - APs do not need to care about a check of the conditions

Introduction (II)

- Static integrity constraints relate to restrictions of the possible database states
  - Example: German professors may only have ranks C1, C2, C3, or C4
- Dynamic integrity constraints to restrictions of the possible database state transitions
  - Example: Professors may only be promoted, but not demoted; their rank may not be set from C4 to C3
- Examples for ICs (colloquially)
  - No customer name may appear more than once in the relation *customers*.
  - A customer name cannot be *null*.
  - Each customer name in the relation *orders* must appear in the relation *customer*.
  - No account of a customer is allowed to be less than USD -100.00.
  - The account of Mr White may not be overchecked.
  - Only those products can be ordered for which at least one supplier exists.
  - The bread price may not be increased

Introduction (III)

- Classification of static ICs
  - Domain constraints: ICs imposed on single attributes in a relation schema
  - Tuple constraints: ICs imposed on tuples regarding its different attributes
  - Key constraints: ICs imposed on a table to make all tuples unique by means of a unique, non-null value as a primary key or candidate key
  - Referential integrity: ICs imposed on the tuples between two relations to maintain dependencies and consistency between them
- Classification of dynamic ICs
  - General constraints: ICs that may extend over several relations and that have to be always satisfied by a database
  - Triggers: ICs that are user-defined procedures and provide powerful constraint support beyond the built-in ICs
- Static ICs are declared in SQL by the
  - **create table** command as part of the database schema
  - **alter table** … **add constraint** … command to add them to an already existing database schema

Domain Constraints (I)

- They refer to integrity constraints that are implicitly or explicitly imposed on single attributes in a relation schema (type integrity)

- Each attribute value must be atomic

- The atomicity of values is determined by the SQL data types (number data types, string data types, time-related data types, binary data types)

- Each data type has a special value *null* by default

- The *null* value is the default value of any attribute value for which no other value is provided during insertion and for which a *null* value is permitted

- not null constraint
  - This IC attached to an attribute declaration is specified if *null* is not permitted as a value for a particular attribute, i.e., a value unequal to *null* must be provided for such an attribute

Domain Constraints (II)

- not null constraint (*continued*)
  - Example: In the table schema *students* we specify

    create table students
    (…, name **varchar**(30) **not null**, …);
  - This IC prohibits the insertion of a *null* value for the attribute and the update of the attribute value with a *null* value
  - This IC automatically holds for any prime attribute indicated by the **primary key** constraint or the **unique** constraint (see below)
- default clause
  - A default value can be specified for an attribute by appending the clause **default** <value> to an attribute definition
  - The default value is a constant or the result of an expression and is inserted into any new tuple if an explicit value is not provided for that attribute (if not specified, the default value is *null*)
  - Example: In the table schema *students* we can specify

    create table students(…, sem **int** **default** 1, …);

Domain Constraints (III)

- check clause
  - This clause follows an attribute definition or domain definition (see below), specifies a restriction or condition on the attribute or domain values, and allows the definition of enumeration types and range types
  - Examples
    - In the table schema *professors* we can specify

      create table professors
      (…, rank **char**(2) **check**(rank **in** ‘C1’, ‘C2’, ‘C3’, ‘C4’), …);
    - It is also possible to put the clause at the end of the table schema

      create table professors
      (…, rank **char**(2), …, **check**(rank **in** ‘C1’, ‘C2’, ‘C3’, ‘C4’));
    - In the table schema *lectures* we can specify

      create table lectures
      (…, credits **int** **check**(credits >= 1 **and** credits <= 3), …); or

      create table lectures
      (…, credits **int** **check**(credits **between** 1 **and** 3), …);

Tuple Constraints (I)

- They represent restrictions of the values that a tuple can take with respect to its different attributes

- A tuple constraint can be defined by the *check* clause during the specification of a relation schema

- Examples
  - In a relation *occupancy*, where a tuple stores the seat reservation from a train station *origin* to a station *destination*, *origin* should be unequal to *destination*

```
create table occupancy
(…,
 origin varchar(50),
 destination varchar(50),
 …,
 check(origin <> destination));
```

Tuple Constraints (II)

- Examples (*continued*)
  - Handling *null* values in foreign keys
    - A composite foreign key is only checked if *all* of the attribute values involved are *not null*
    - In general, a composite foreign key that *partially* contains *null* values does not make much sense
    - If a foreign key is supposed to be totally *null* or totally defined, this must be explicitly expressed by a *check* clause
    - Given a relation schema $R(A_1, \ldots, A_n, B_1, \ldots, B_m)$ with the foreign key attributes $B_1, \ldots, B_m$, we can write a constraint that enforces the foreign key attribute values to be all *null* or all *not null*

      check ((B_1 **is null and** … **and** B_m **is null**) **or**

      (B_1 **is not null and** … **and** B_m **is not null**))

Key Constraints (I)

- They refer to integrity constraints that
  - uniquely identify each tuple in a relation
  - uniquely determine all the prime and nonprime attribute values in a tuple
  - distinguish each tuple from all the other tuples in a relation
- primary key constraint
  - This IC refers to a minimal set of attributes that uniquely identifies each tuple of a relation; one primary key per table schema
  - If a primary key has a single attribute, a primary key clause can be added to the attribute or listed at the end of the table schema, e.g.,

    create table students(reg-id **int** **primary key**, …);

    create table students(reg-id **int** **not null**, …, **primary key**(reg-id), …);
  - If a primary key has more than one attribute, it is listed at the end of the table schema, e.g.,

    create table attends(reg-id **int**, id **int**, **primary key**(reg-id, id));
  - Each primary key attribute is implicitly (or explicitly) declared as **not null** (entity integrity)

Key Constraints (II)

- unique constraint
  - This IC specifies alternative unique keys, i.e., candidate keys (several candidate keys are allowed)
  - If a candidate key has a single attribute, a unique clause can be added to the attribute or listed at the end of the table schema, e.g.,

    create table professors(…, room **int unique**, …);

    create table professors(…, room **int not null**, …, **unique**(room), …);
  - If a candidate key has more than one attribute, it is listed at the end of the table schema in the form

    create table T(…, A_1 D_1, … , A_n D_n, …, **unique**(A_1, …, A_n), …);
  - No two tuples in the relation can be equal on all the listed attributes
  - Candidate key attributes are permitted to be *null* unless they have explicitly been declared to be *not null*

Referential Integrity (I)

- Referential integrity ensures that a value which appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation; thus, it establishes consistent relationships between tables
- The referential integrity constraint is specified between two relations and is used to maintain the consistency between tuples in the two relations
- The referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an *existing tuple* in that relation
- Example

assistants			
pers-id	name	room	boss
3002	Platon	156	2125
3003	Aristoteles	199	2125
3004	Wittgenstein	101	2126
3005	Rhetikus	130	2127
3006	Newton	120	2127
3007	Spinoza	155	2134

professors			
pers-id	name	rank	room
2125	Sokrates	C4	226
2126	Russel	C4	232
2127	Kopernikus	C3	310
2133	Popper	C3	052
2134	Augustinus	C3	309
2136	Curie	C4	036
2137	Kant	C4	007

Referential Integrity (II)

- Let r and s be relations with the schemas R and S. Let $K \subseteq R$ be the primary key of R. Then $F \subseteq S$ is called foreign key of S that references relation schema R if it satisfies the following rules:
  1. The sets K and F have the same number of attributes, i.e., $|K| = |F|$
  2. The attributes in F have the same domains as the primary key attributes in K, i.e., $dom(F) = dom(K)$
  3. $\forall\, t_s \in s : (\forall\, A \in F : t_s[A] = null) \vee (\exists\, t_r \in r : t_s[F] = t_r[K])$
- The attributes in F are said to reference or refer to the relation schema R
- The relation s is called the referencing relation, and the relation r is called the referenced relation
- If these three conditions hold, a referential integrity constraint from S to R (from s to r) is said to hold
- Example: relations *customer*, *order*, *product* (*order* models m:n-relationship)
  - Possible problems if referential integrity is not fulfilled:
    - Customer orders products that do not exist
    - Products can be ordered by a customer who does not exist

Referential Integrity (III)

- Relational Algebra characterization of referential integrity: Let r and s be relations with the schemas R and S. Let $K \subseteq R$ be the primary key of R. Then $F \subseteq S$ is called foreign key of S that references relation schema R if it satisfies the following rules:
  1. The sets K and F have the same number of attributes, i.e., $|K| = |F|$
  2. The attribute sets F and K have the same domains: $dom(F) = dom(K)$
  3. $\pi_F(S) \subseteq \pi_K(R)$ [subset dependency]
- Use of the **foreign key** (…) **references** … clause to establish referential integrity between two relations in SQL

```
create table assistants
    (pers-id  int not null,
     name     varchar(30) not null,
     room     int unique,
     boss     int,
     primary key (pers-id),
     foreign key (boss) references
                    professors(pers-id),
     check (boss is not null));
```

```
create table professors
    (pers-id  int not null,
     name     varchar(30) not null,
     room     int unique,
     rank     char(2),
     primary key (pers-id),
     check(rank in 'C1', 'C2', 'C3', 'C4'));
```

Referential Integrity (IV)

- Maintenance of referential integrity under data manipulation operations
  - Given: Relations r and s with schemas R and S resp., $K \subseteq R$ is the primary key of R, $F \subseteq S$ is a foreign key of S
  - The insertion of a tuple t into relation s is only possible if

    $t[F] \in \pi_K(r) \vee (\forall\, A \in F : t[A] = null)$

    Assumption: None of the attributes $A \in F$ is declared to be *not null*
  - An update of the values of the foreign key attributes F in a tuple $t \in s$ is only possible if

    $t_u[F] \in \pi_K(r) \vee (\forall\, A \in F : t_u[A] = null)$

    where $t_u \in s$ is the updated version of $t \in s$

    Assumption: None of the attributes $A \in F$ is declared to be *not null*
  - An update of the values of the primary key attributes K in a tuple $t \in r$ is only possible if $\sigma_{F=t[K]}(s) = \emptyset$
  - The deletion of a tuple $t \in r$ is only possible if $\sigma_{F=t[K]}(s) = \emptyset$

Referential Integrity (V)

- Standard behavior of the DBMS when violating a referential integrity constraint: rejection of the action that caused the violation, i.e., the transaction performing the update action is rolled back

- Attempt to update or delete a tuple in the referenced relation *R* that has a matching tuple in the referencing relation *S* depends on the referential action specified using the **on delete** or **on update** sub-clauses of the **foreign key** clause

- Possible referential actions for the **on delete** sub-clause
  - **on delete cascade**: The tuple (including its primary key) of the referenced relation *R* as well as all the matching tuples (including their foreign keys) in a referencing relation *S* are deleted.
  - **on delete set null**: The tuple of the referenced relation *R* is deleted, and the foreign key values in all matching tuples of a referencing relation *S* are set to *null*. This option is only valid if the foreign key attributes in *S* do not have the *not null* constraint specified.