Databases and Information Systems

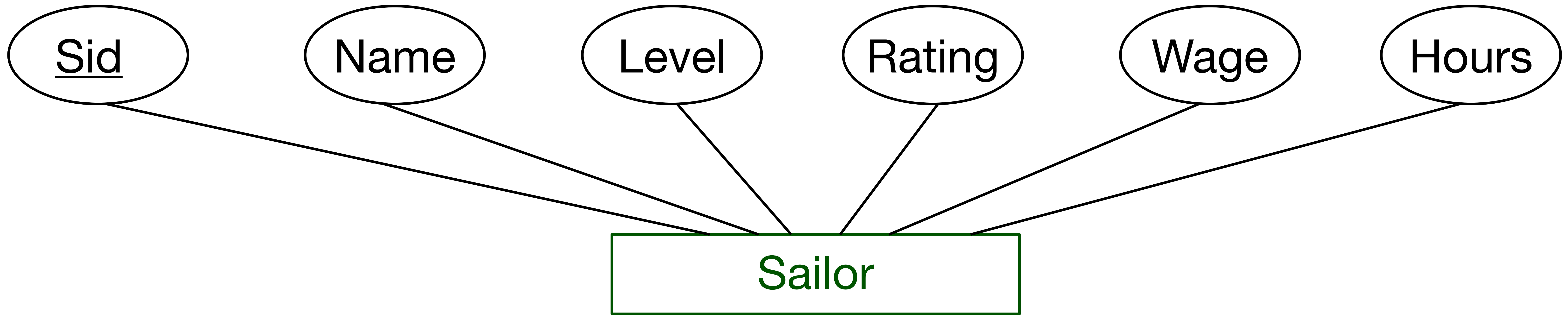
Functional Dependencies & Normal Forms
Views & Triggers

Dmitriy Traytel
slides mostly by Marcos Vaz Salles

UNIVERSITY OF COPENHAGEN

DIKU

Data Redundancy



S	N	L	R	W	H
22	dustin	48	8	10	40
31	lubber	22	8	10	30
64	horatio	35	5	7	30
29	brutus	35	5	7	32
58	rusty	35	8	10	40

- **Constraint**: all sailors with the same rating have the same wage (R → W)
- Problems due to data redundancy?

Source: Ramakrishnan & Gehrke

DIKU

Relation Decomposition

S	N	L	R	W	H
22	dustin	48	8	10	40
31	lubber	22	8	10	30
64	horatio	35	5	7	30
29	brutus	35	5	7	32
58	rusty	35	8	10	40

S	N	L	R	H
22	dustin	48	8	40
31	lubber	22	8	30
64	horatio	35	5	30
29	brutus	35	5	32
58	rusty	35	8	40

Wages

R	W
8	10
5	7

Source: Ramakrishnan & Gehrke

DIKU

Relation Decomposition

S	N	L	R	W	H
22	dustin	48	8	10	40
31	lubber	22	8	10	30
64	horatio	35	5	7	30
29	brutus	35	5	7	32
58	rusty	35	8	10	40

S	N	L	R	H
22	dustin	48	8	40
31	lubber	22	8	30
64	horatio	35	5	30
29	brutus	35	5	32
58	rusty	35	8	40

Wages

R	W
8	10
5	7

Problem?

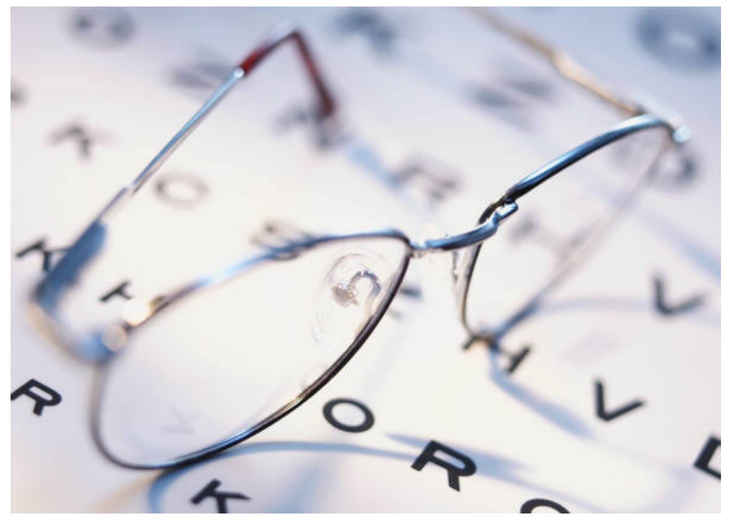
Source: Ramakrishnan & Gehrke

DIKU

Towards Normal Forms

- First question is to ask whether any schema refinement is needed
- If a relation is in a normal form (BCNF, 3NF etc.), certain anomalies are avoided/minimized
- If not, decompose relation to normal form
- Role of functional dependencies (FDs) in detecting redundancy:
  - Consider a relation R with 3 attributes, ABC
    - No FDs hold: There is no redundancy here.
    - Given A → B: Several tuples could have the same A value, and if so, they'll all have the same B value!

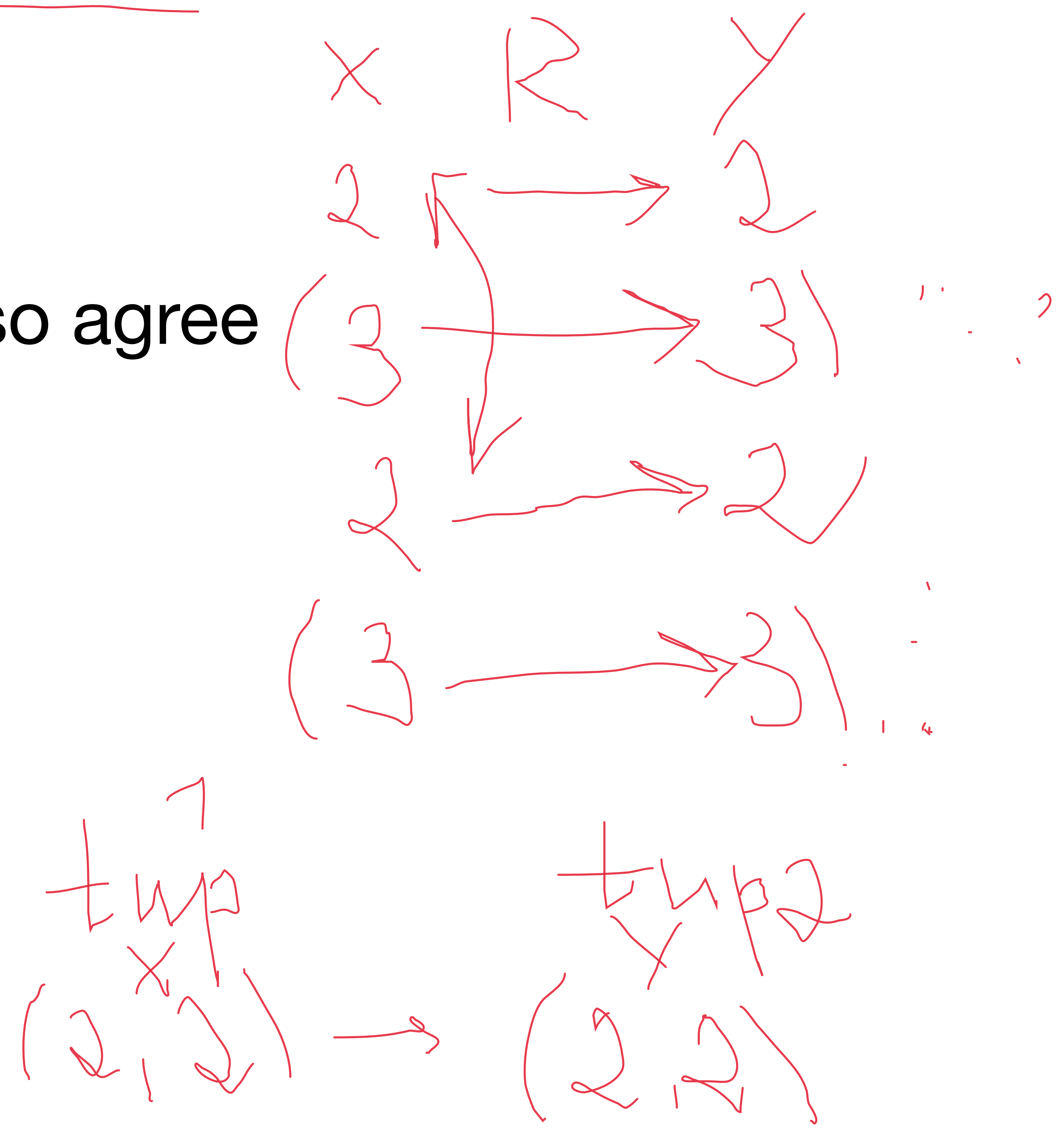
What should we learn today?



- Define and explain the notion of **functional dependencies** (FDs), and apply rules to reason about FDs including Armstrong's Axioms
- Review the notion of **keys** and how to **determine** them by reasoning on functional dependencies
- Explain the issues in **decomposition** of relations and verify whether decompositions are **lossless-join** and **dependency-preserving**
- Apply the **chase test** to determine if a decomposition is lossless-join
- Explain the multiple **normal forms** a relation can be in, including the Boyce-Codd Normal Form (**BCNF**) and the Third Normal Form (**3NF**)
- **Decompose** relations into BCNF or 3NF
- Explain the concepts of (**materialized**) **views** and **triggers**, argue for their use, and create them in SQL

Functional Dependencies (FDs)

- A functional dependency $\boxed{X \rightarrow Y}$ between sets of attributes X and Y holds over relation R if, for every allowable instance r of R:
  - $\pi_X(t) = \pi_X(u)$ implies $\pi_Y(t) = \pi_Y(u)$ for all $t \in r$ and $u \in r$
  - i.e., given two tuples in r, if the X values agree, then the Y values must also agree
- An FD is a statement about all allowable instances.
  - Must be identified based on the application's semantics
  - Given allowable instance r of R, we can check if r violates some FD f, but we cannot tell if f holds over R!
- K is a candidate key for R means that K → R
  - However, K → R does not require K to be minimal!



DIKU

Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - ssn $\rightarrow$ did, did $\rightarrow$ lot implies ssn $\rightarrow$ lot
- An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.
  - F^+ = closure of F is the set of all FDs that are implied by F.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$
  - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These are sound and complete inference rules for FDs!

Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - ssn → did, did → lot implies ssn → lot
- An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.
  - F^+ = closure of F is the set of all FDs that are implied by F.
- Armstrong's Axioms (X, Y, Z are sets of attributes):
  - Reflexivity: If $Y \subseteq X$, then $X \rightarrow Y$ $\longrightarrow$ trivial FDs
  - Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- These are sound and complete inference rules for FDs!

DIKU

Reasoning About FDs (Contd.)

- Couple of additional rules (that follow from Armstrongs' axioms):
  - Union: If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - Decomposition: If $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
- Example: Contracts(cid,sid,jid,did,pid,qty,value), and:
  - C is the key: $C \rightarrow CSJDPQV$
  - Project purchases each part using single contract: $JP \rightarrow C$
  - Department purchases at most one part from a supplier: $SD \rightarrow P$
- Can you infer $SDJ \rightarrow CSJDPQV$?

DIKU

Reasoning About FDs (Contd.)

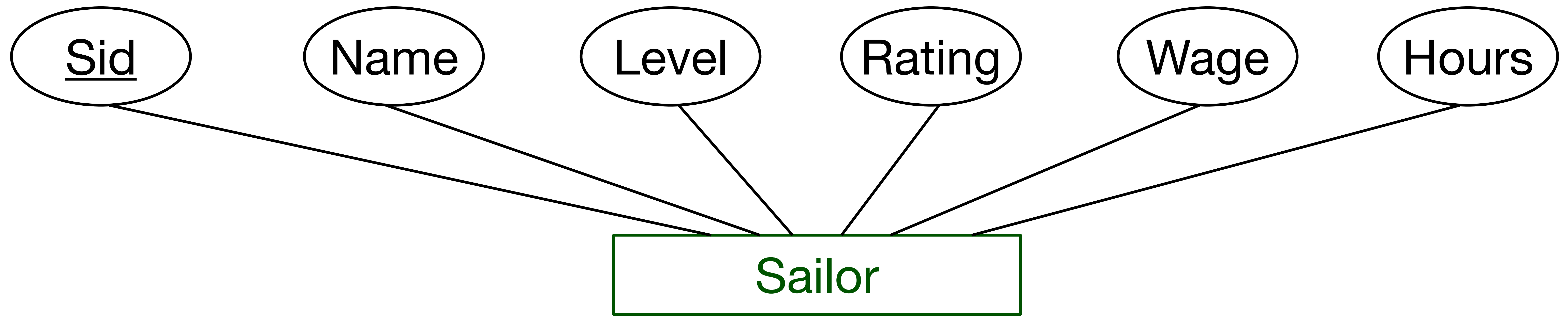
- Computing the closure of a set of FDs can be expensive. (Size of closure is exponential in # attrs!)
- Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure of a set of FDs F. An efficient check:
  - Compute attribute closure of X (denoted X^+) wrt F:
    - Set of all attributes A such that $X \rightarrow A$ is in F^+
    - There is a linear time algorithm to compute this.
  - Check if Y is in X^+
- Does $F = \{A \rightarrow B, B \rightarrow C, CD \rightarrow E\}$ imply $A \rightarrow E$?
  - i.e., is $A \rightarrow E$ in the closure F^+? Equivalently, is E in A^+?
  - Can be used to find keys!!!

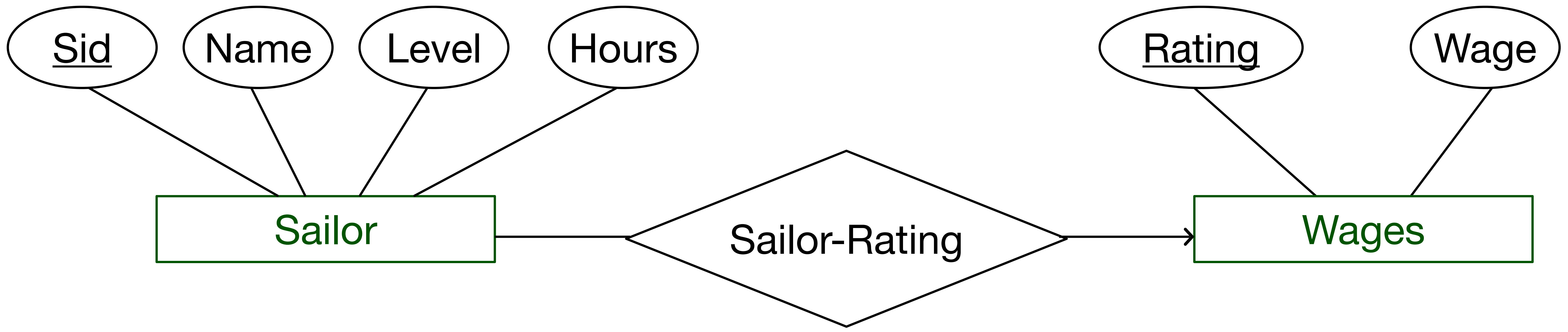
Finding Keys

- Example: Contracts(cid,sid,jid,did,pid,qty,value), and:
  - C is the key: $C \rightarrow CSJDPQV$
  - Project purchases each part using single contract: $JP \rightarrow C$
  - Department purchases at most one part from a supplier: $SD \rightarrow P$
- Discussion: Find all the keys of Contracts!
  - How do you go about it?

DIKU

Recall: Modifying ER Diagram





Source: Ramakrishnan & Gehrke (partial)

DIKU

Decomposition of a Relation Scheme

- Suppose that relation R contains attributes A1 ... An. A <u>decomposition</u> of R consists of replacing R by two or more relations such that:
  - Each new relation scheme contains a subset of the attributes of R (and no attributes that do not appear in R), and
  - Every attribute of R appears as an attribute of one of the new relations.
- Intuitively, decomposing R means we will store instances of the relation schemes produced by the decomposition, instead of instances of R.
- E.g., can decompose SNLRWH into SNLRH and RW.

DIKU

Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs $S \rightarrow SNLRWH$ and $R \rightarrow W$
  - Data duplication due to second FD
  - Will make this more precise during the definition of normal forms
- Decompose to SNLRH and RW
  - What should we be careful about?

DIKU

Problems with Decompositions

- There are three potential problems to consider:
  1. Some queries become more **expensive**
     - e.g., How much did sailor Joe earn? (salary = W*H)
  2. Given instances of the decomposed relations, we may **not** be **able to reconstruct** the corresponding instance of the original relation
     - Fortunately, not in the SNLRWH example.
  3. **Checking dependencies may require joining** the instances of the decomposed relations
     - Fortunately, not in the SNLRWH example.

- Tradeoff: Must consider these issues vs. redundancy.

Lossless-Join Decompositions

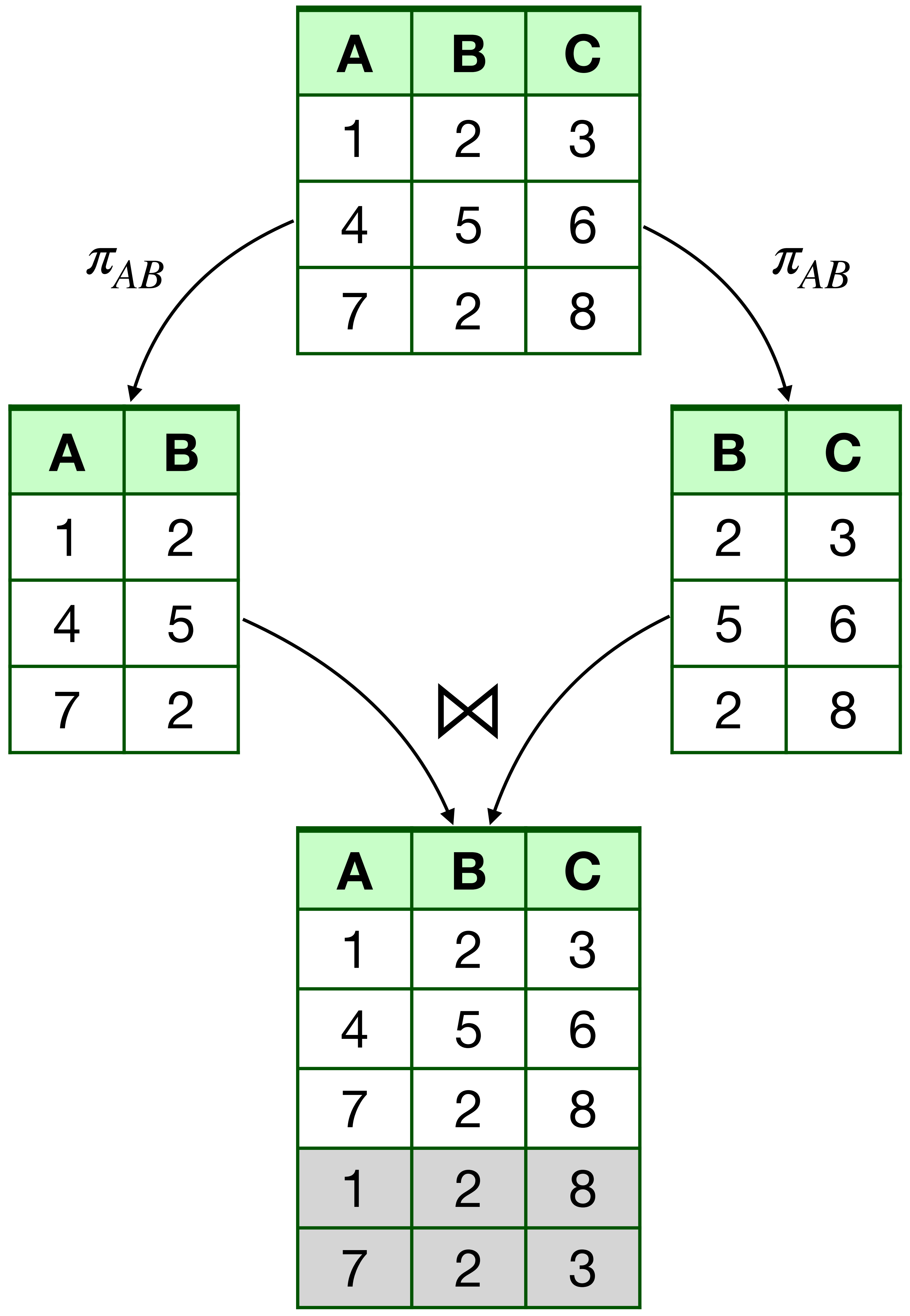
- Decomposition of R into X and Y is lossless-join w.r.t. a set of FDs F if, for every instance r that satisfies F: $\pi_X(r) \bowtie \pi_Y(r) = r$
- It is always true that $r \subseteq \pi_X(r) \bowtie \pi_Y(r)$
  - In general, the other direction does not hold!
- Definition extended to decomposition into 3 or more relations in a straightforward way.
- It is essential that all decompositions used to deal with redundancy be lossless-join! (Avoids Problem 2.)

DIKU

More on Lossless-Join

- The decomposition of R into X and Y is lossless-join w.r.t. F **if and only if** the F+ contains:
  - X ∩ Y → X,   or
  - X ∩ Y → Y

- In particular, the decomposition of R into UV and R - V is lossless-join if U → V holds over R



A	B	C
1	2	3
4	5	6
7	2	8

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

A	B	C
1	2	3
4	5	6
7	2	8
1	2	8
7	2	3

DIKU

The Chase Test

- Suppose tuple t comes back in the join.
  - Start by assuming t = abc… .
- Then t is the join of projections of some tuples of R, one for each R_i of the decomposition.
  - For each i, there is a tuple s_i of R that has a, b, c,… in the attributes of R_i.
  - s_i can have any values in other attributes.
- Can we use the given FD's to show that one of these tuples must be t?

Source: Bulskov (partial)

DIKU

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be $C \rightarrow D$ and $B \rightarrow A$.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

DIKU

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be $C \rightarrow D$ and $B \rightarrow A$.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

A	*B*	*C*	*D*
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be C → D and B → A.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

The tuples of R projected onto AB, BC, CD.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

Source: Bulskov (partial)

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be C → D and B → A.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

The tuples of R projected onto AB, BC, CD.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use $C \to D$

DIKU

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be $C \rightarrow D$ and $B \rightarrow A$.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

The tuples of R projected onto AB, BC, CD.

A	B	C	D
a	b	c_1	d_1
$\cancel{a_2}$ a	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use $B \rightarrow A$

Use $C \rightarrow D$

Source: Bulskov (partial)

DIKU

Example: The Chase

- Let R = ABCD, and the decomposition be AB, BC, and CD.
- Let the given FD's be $C \rightarrow D$ and $B \rightarrow A$.
- Suppose the tuple **t = abcd** is the join of tuples projected onto AB, BC, CD.

The tuples of R projected onto AB, BC, CD.

A	B	C	D
a	b	c_1	d_1
$\cancel{a_2}$ a	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use $B \rightarrow A$

Use $C \rightarrow D$

We've proved the second tuple must be *t*.

Source: Bulskov (partial)

DIKU

Example: Lossy Join

- Same relation $R = ABCD$ and same decomposition.
- But with only the FD $C \rightarrow D$.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	d_2
a_3	b_3	c	d

Source: Bulskov

DIKU

Example: Lossy Join

- Same relation $R = ABCD$ and same decomposition.
- But with only the FD $C \rightarrow D$.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

Use $C \rightarrow D$

DIKU

Example: Lossy Join

- Same relation $R = ABCD$ and same decomposition.
- But with only the FD $C \rightarrow D$.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

These three tuples are an example that shows the join is actually lossy. Tuple *abcd* is not in *R*, but we can project and rejoin to get *abcd*.

Use $C \rightarrow D$

DIKU

Example: Lossy Join

- Same relation $R = ABCD$ and same decomposition.
- But with only the FD $C \rightarrow D$.

A	B	C	D
a	b	c_1	d_1
a_2	b	c	$\cancel{d_2}$ d
a_3	b_3	c	d

These projections rejoin to form *abcd*.

These three tuples are an example that shows the join is actually lossy. Tuple *abcd* is not in *R*, but we can project and rejoin to get *abcd*.

Use $C \rightarrow D$

Dependency-Preserving Decomposition

- Consider CSJDPQV, C is key, JP → C and SD → P.
    - Decomposition: CSJDQV and SDP
    - (Is it lossless-join?)
    - Problem: Checking JP → C requires a join!
- Dependency-preserving decomposition (Intuitive):
    - If R is decomposed into X, Y and Z, and we enforce the FDs that hold on X, on Y and on Z, then all FDs that were given to hold on R must also hold. (Avoids Problem (3).)

DIKU

Dependency Preserving Decompositions (Contd.)

- Projection of set of FDs F: Projection of F onto X (denoted F_X) is the set of FDs $U \rightarrow V \in F^+$ (F's closure) such that $U, V \subseteq X$.

- Decomposition of R into X and Y is dependency preserving if $(F_X \cup F_Y)^+ = F^+$
  - i.e., if we consider only dependencies in the closure F^+ that can be checked in X without considering Y, and in Y without considering X, these imply all dependencies in F^+.

- Important to consider F^+, not F, in the definition of projection F_X:
  - ABC, $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, decomposed into AB and BC.
  - Is this dependency preserving? Is $C \rightarrow A$ preserved?

- Dependency preserving does not imply lossless-join (and vice versa):
  - ABC, $A \rightarrow B$, decomposed into AB and BC. (see first lossy join example)

DIKU

Boyce-Codd Normal Form (BCNF)

- Relation R with FDs F is in **BCNF** if, for all $X \rightarrow Y$ in F^+
  - $Y \subseteq X$ (called a trivial FD), or
  - X contains a key for R.

- In other words, R is in BCNF if the only non-trivial FDs that hold over R are key constraints.
  - No dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the A value, we cannot infer the C value in one tuple from the C value in the other.
  - If example relation was in BCNF and A → C, the 2 tuples would have to be identical (A is a key).

A	B	C
10	1	3
10	2	?

Source: Ramakrishnan & Gehrke

Decomposition into BCNF

- Consider relation R with FDs F. If X → Y violates BCNF, decompose R into R - Y and XY.
  - Repeated application of this idea will give us a collection of relations that are in BCNF; lossless-join decomposition, and guaranteed to terminate.
  - e.g., CSJDPQV, key C, JP → C, SD → P, J → S
  - To deal with SD → P, decompose into SDP, CSJDQV.
  - To deal with J → S, decompose CSJDQV into JS and CJDQV
- In general, several dependencies may cause violation of BCNF. The order in which we “deal with” them could lead to different sets of relations!

DIKU

BCNF and Dependency Preservation

- In general, there may not be a dependency preserving decomposition into BCNF.
  - e.g., CSZ, CS → Z, Z → C
  - Cannot decompose while preserving CS → Z; not in BCNF.
- Similarly, decomposition of CSJDPQV into SDP, JS and CJDQV is not dependency preserving (w.r.t. the FDs **JP** → **C**, SD → P and J → S).
  - However, it is a lossless-join decomposition.

Source: Ramakrishnan & Gehrke (partial)

Third Normal Form (3NF)

- Relation R with FDs F is in **3NF** if, for all $X \to Y$ in F^+
  - $Y \subseteq X$ (called a trivial FD), or
  - X contains a key for R, or
  - Each attribute in Y is part of some key for R (i.e., each attribute in Y is prime).
- Minimality of a key is crucial in third condition above!
- If R is in BCNF, then it is also in 3NF.
- If R is in 3NF, some redundancy is possible. It is a compromise, used when BCNF not achievable (e.g., no “good” decomposition, or performance considerations).
- Lossless-join, dependency-preserving decomposition of R into a collection of 3NF relations always possible.

Source: Ramakrishnan & Gehrke

Decomposition into 3NF

- Obviously, the algorithm for lossless join decomposition into BCNF can be used to obtain a lossless join decomposition into 3NF (typically, can stop earlier).

- To ensure dependency preservation, one idea:
  - If $X \rightarrow Y$ is not preserved, add relation XY.
  - Problem is that XY may violate 3NF!
  - E.g., consider the addition of CJP to 'preserve' $JP \rightarrow C$. What if we also have $J \rightarrow C$?

- Refinement: Instead of the given set of FDs F, use a minimal cover for F (also called minimal basis – see book)

Normal Forms

- First normal form
  - The domain of each attribute contains only atomic values (Codd)
  - There are no duplicate rows (Date)
  - Each entry contains exactly one value from the applicable domain [and nothing else] (Date)
  - All columns are regular (Date)
  - Often implemented by types and constraints

- Second normal form
  - 1NF
  - No non-prime attribute is dependent on any proper subset of any key of the table.

- Third normal form
  - 2NF
  - Every non-prime attribute is directly dependent on every superkey of the table.

Source: Bulskov (partial)

Exercise

- Courses(course,teacher,hour,room,student,grade)
- Abbreviated CTHRSG
- FDs: $C \to T$, $HR \to C$, $HT \to R$, $HS \to R$, $CS \to G$
- Determine all the keys
- Decompose into BCNF

Questions so far?

DIKU

Views

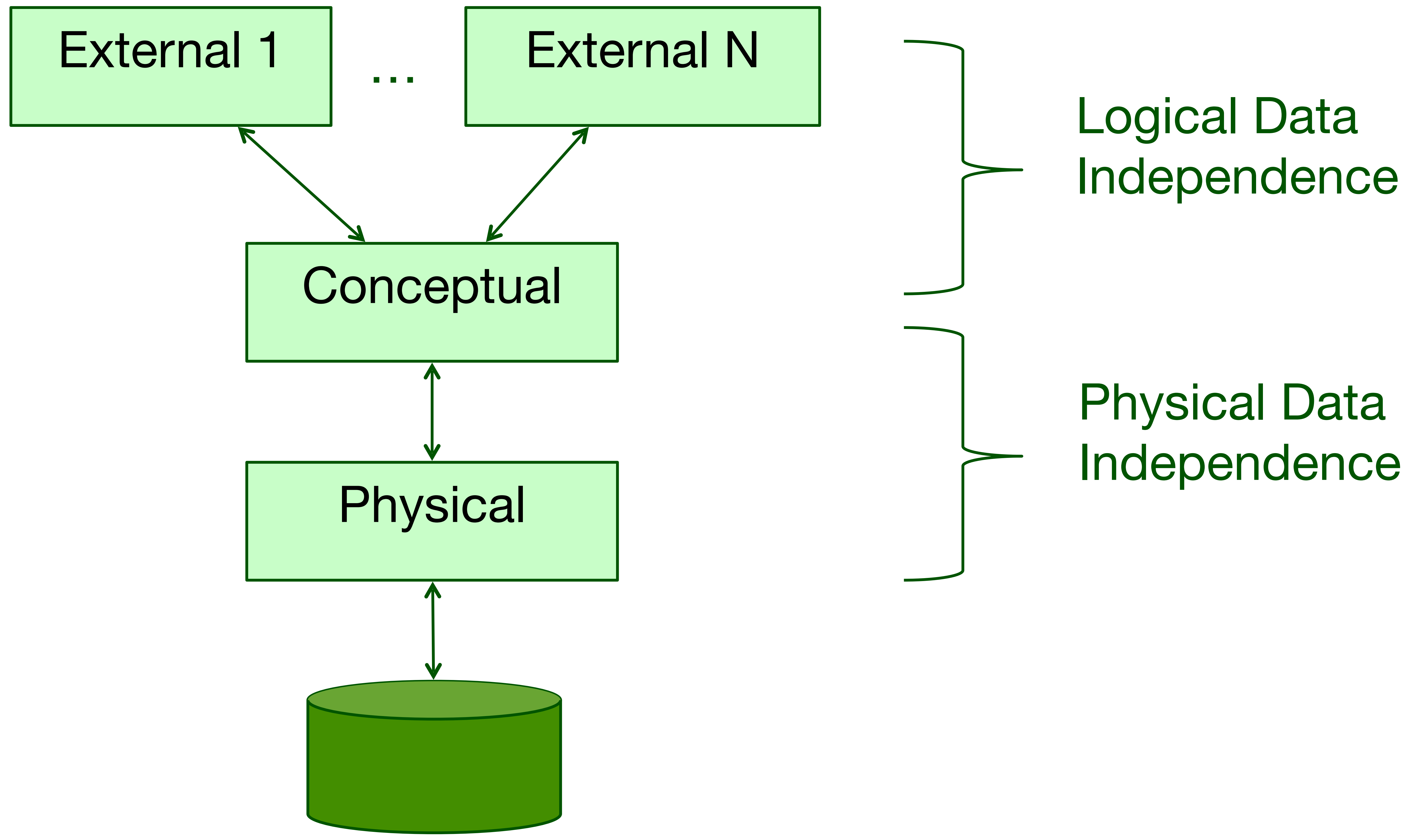
- Special construct to add logical table to **schema** of the database

```
CREATE VIEW Red_Green_Sailors (sid, sname) AS
  SELECT DISTINCT S.sid, S.sname
  FROM   Sailors S, Boats B, Reserves R
  WHERE  S.sid=R.sid AND R.bid=B.bid
    AND  (B.color=‘red’ OR B.color=‘green’)
```

```
DROP VIEW Red_Green_Sailors
```

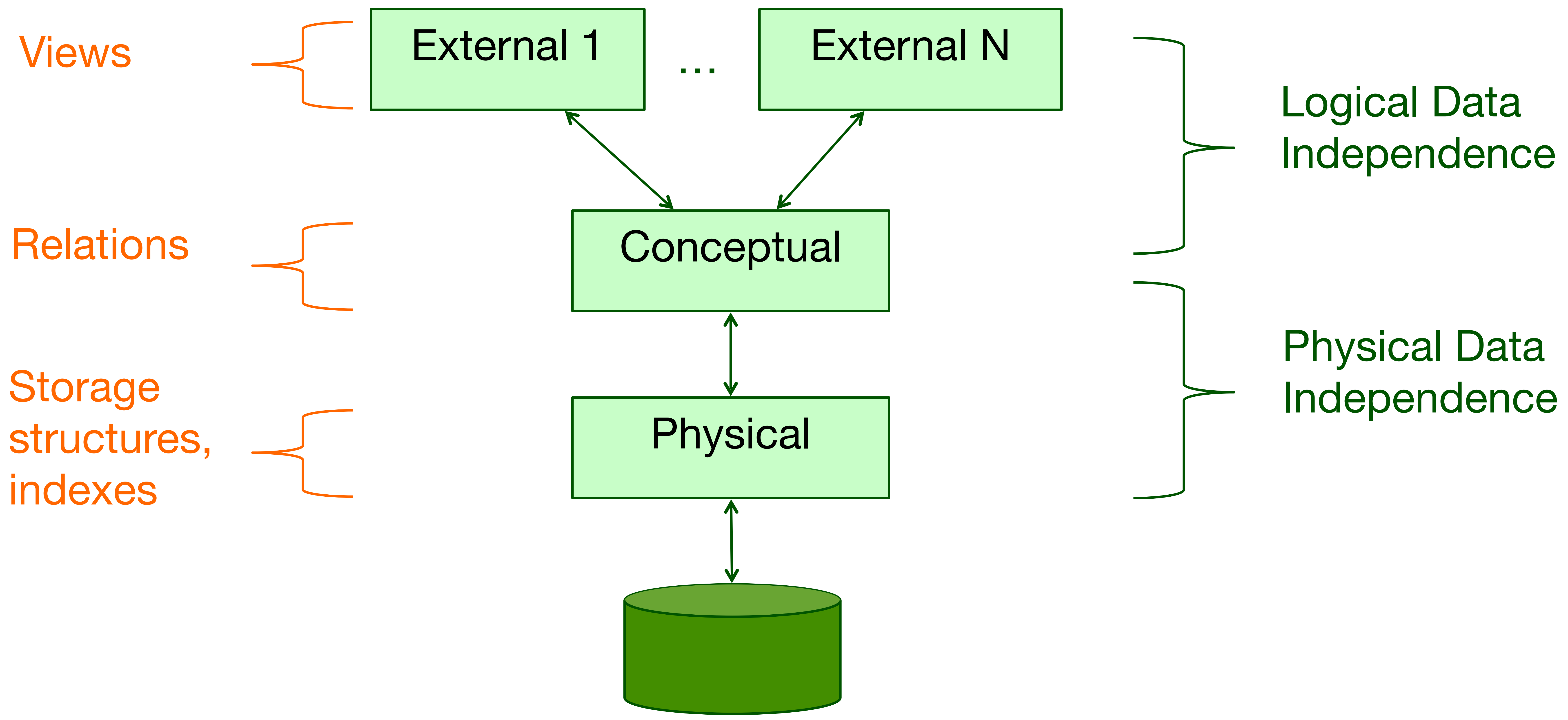
DIKU

Remember this picture?



DIKU

Remember this picture?


Views
Relations
Storage structures, indexes
External 1
...
External N
Conceptual
Physical
Logical Data Independence
Physical Data Independence

Why use views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access

Real applications tend to use lots and lots (and lots and lots!) of views

DIKU

Why use views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access

Real applications tend to use lots and lots (and lots and lots!) of views

- Virtual vs. Materialized Views
  - + Improve query performance
  - + Transparency to applications
  - – Materialized view could be LARGE!
  - – View maintenance under updates

Source: Widom (partial)

DIKU

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query

Source: Widom (partial)

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query

- But: Modifications to base data invalidate view
  - DBMS can be given a policy to refresh view, e.g., daily
  - Incremental maintenance possible in some cases

DIKU

Materialized View Example

```
CREATE MATERIALIZED VIEW CA-CS AS
SELECT C.cName, S.sName
FROM College C, Student S, Apply A
WHERE C.cName = A.cName AND S.sID = A.sID
AND C.state = ‘CA’ AND A.major = ‘CS’
```

- Can use CA-CS as if it’s a table (it is!)
  - DBMS will store a copy of result of view query
- But: Modifications to base data invalidate view
  - DBMS can be given a policy to refresh view, e.g., daily
  - Incremental maintenance possible in some cases
- Also: If you can update the view, base tables must stay in sync
  - DBMS must propagate updates through view (not always possible…)

View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes.
  - Refresh: Apply changes to the materialized view table.

- Maintenance policy: Controls when we do refresh.
  - Immediate: As part of the operation that modifies the underlying data tables.
    - + Materialized view is always consistent
    - – Updates are slowed down
  - Deferred: Some time later, as a separate operation.
    - – View becomes inconsistent
    - + Can scale to maintain many views without slowing down updates

View Maintenance

- Two steps:
  - Propagate: Compute changes to view when data changes.
  - Refresh: Apply changes to the materialized view table.

- Maintenance policy: Controls when we do refresh.
  - Immediate: As part of the operation that modifies the underlying data tables.
    - + Materialized view is always consistent
    - – Updates are slowed down
  - Deferred: Some time later, as a separate operation.
    - – View becomes inconsistent
    - + Can scale to maintain many views without slowing down updates
    - Lazy: Delay refresh until next query on view; then refresh before answering the query.
    - Periodic (Snapshot): Refresh periodically. Queries possibly answered using outdated version of view tuples. Widely used, especially for asynchronous replication in distributed databases, and for warehouse applications.
    - Event-based: E.g., refresh after a fixed number of updates to underlying data tables.

Source: Ramakrishnan & Gehrke

Triggers

- “Event-Condition-Action Rules”
  - When event occurs, check condition; if true, do action

- Why would we need triggers?

  1) Move monitoring logic from application into DBMS

  2) Enforce constraints
     - Beyond what constraint system supports
     - Automatic constraint “repair”

- Implementations vary significantly
  - Different DBMS may support different kinds of events and actions
  - Different DBMS may allow triggers to be written in different languages (e.g., through stored procedures)

Source: Widom (partial)

DIKU

Triggers in SQL

```
CREATE TRIGGER name
BEFORE|AFTER|INSTEAD OF events

[ referencing-variables ]

[ FOR EACH ROW ]

WHEN ( condition )

action
```

DIKU

A Simple Example of a Trigger

```
CREATE TRIGGER AUR_NetWorthTrigger          Trigger name in
                                            database schema
AFTER UPDATE OF netWorth ON MovieExec       Event: After update of attribute
REFERENCING
    OLD ROW AS OldTuple
    NEW ROW AS NewTuple                     Access to tuple values
FOR EACH ROW                                Row level trigger
WHEN (OldTuple.netWorth > NewTuple.netWorth)   Condition
       UPDATE MovieExec
       SET netWorth = OldTuple.netWorth     Action
       WHERE cert# = NewTuple.cert#;        Prevent lower values
                                            Reset to old value
```

NOTE: Not PostgreSQL syntax

- **Granularity**
  - Row-level: Event = Change of a single row (a single UPDATE statement might result in multiple events)
  - Statement-level: Event = Statement (a single UPDATE statement that changes multiple rows is a single event).

Source: Lassen (partial) 37

Aggregate Maintenance with Triggers

```
Orders(order_num, item_num, quantity, store_id, vendor_id)
Items(item_num, price)
VendorOutstanding(vendor_id, amount)
```

- Insertions

```
INSERT INTO orders VALUES (1000350,7825,100,'xxxxxx6944','vendor4');
```

- Queries (first without, then with redundant tables)

```
SELECT vendor_id, SUM(quantity * price)
FROM orders NATURAL JOIN items
GROUP BY vendor_id;
```

vs.

```
SELECT * FROM vendorOutstanding;
```

Source: Shasha and Bonnet (partial) 38

DIKU

Aggregate Maintenance with Triggers (contd.)

```
CREATE TRIGGER
trUpsertVendorOutstanding
AFTER INSERT ON orders
FOR EACH ROW
EXECUTE PROCEDURE
pyUpsertVendorOutstanding();
```

```
CREATE OR REPLACE FUNCTION pyUpsertVendorOutstanding()
RETURNS trigger
AS $$
       # new values inserted
       new_quantity = TD["new"]["quantity"]
       new_item_num = TD["new"]["item_num"]
       new_vendor_id = TD["new"]["vendor_id"]

       # Python code that accesses DB follows
       …
$$ LANGUAGE plpython3u;
```

See script trigger-example.sql in Absalon

Aggregate Maintenance with Triggers (contd.)

```
CREATE OR REPLACE FUNCTION pyUpsertVendorOutstanding()
RETURNS trigger
AS $$
         # new values inserted
         new_quantity = TD["new"]["quantity"]
         new_item_num = TD["new"]["item_num"]
         new_vendor_id = TD["new"]["vendor_id"]

         # prepare upsert SQL statement
         upsert_stmt = plpy.prepare(
                  ("INSERT INTO vendor_outstanding "
                   "as v(vendor_id, amount) VALUES"
                   "($1, $2 * (SELECT items.price "
                   "           FROM items WHERE items.item_num = $3)) "
                   "ON CONFLICT (vendor_id) DO "
                   "UPDATE SET amount = v.amount + EXCLUDED.amount"),
                   ["text", "int", "int"])

         # execute upsert on vendor_outstanding
         plpy.execute(upsert_stmt, [new_vendor_id, new_quantity, new_item_num])
$$ LANGUAGE plpythonu;
```

DIKU

Other Uses of Triggers

- Maintaining an audit trail or history of modifications
  - In an auction, record audit trail of bids per user
- Automatically populating attributes
  - Whenever a bid is updated by a user, record the current time (ignoring any other time value given by application)
- Making updates conditional
  - Disallow users to revise bids down, only allow bids to be revised to be higher than current value

Source: Lassen (partial)

DIKU

Tricky Issues in Triggers

- Row-level vs. Statement-level
  - **New/Old Row** and **New/Old Table**
  - **Before**, **Instead** Of, **After**

- Multiple triggers activated at same time

- Trigger actions activating other triggers (chaining)
  - Also self-triggering, cycles, nested invocations

- Conditions in **When** vs. as part of action

Source: Widom (partial)

DIKU

Triggers: Try it yourself!

```
Entry_card(card_no, username, start_val, end_val, num_failed_entries)
Building(bid, name, address, type)
Authorization(card_no, bid)
Entry_log(card_no, bid, entry_timestamp, entry_status)
```

1. Create a before insert row trigger that assigns the current time to the `entry_timestamp` of a new row being inserted into `Entry_log`. NOTE: To find the current timestamp in PostgreSQL, the CURRENT_TIMESTAMP function can be used.
2. Create a before insert row trigger on `Entry_log` that records the proper entry status of either 'accepted' or 'denied' in `entry_status`. An entry can only be accepted if the card is authorized for the building and if the card is valid and not blocked. In addition, when the entry is denied, then the trigger increments the `num_failed_entries` attribute for the card in the `Entry_card` table.

https://www.postgresql.org/docs/current/sql-createtrigger.html

https://www.postgresql.org/docs/current/plpython.html

What should we learn today?

- Define and explain the notion of **functional dependencies** (FDs), and apply rules to reason about FDs including Armstrong's Axioms
- Review the notion of **keys** and how to **determine** them by reasoning on functional dependencies
- Explain the issues in **decomposition** of relations and verify whether decompositions are **lossless-join** and **dependency-preserving**
- Apply the **chase test** to determine if a decomposition is lossless-join
- Explain the multiple **normal forms** a relation can be in, including the Boyce-Codd Normal Form (**BCNF**) and the Third Normal Form (**3NF**)
- **Decompose** relations into BCNF or 3NF
- Explain the concepts of (**materialized**) **views** and **triggers**, argue for their use, and create them in SQL