The University of Manchester

H7

H8

Designing Databases: Functional Dependencies and Normalization Theory

Fundamentals of Databases

Alvaro A A Fernandes, SCS, UoM

[COMP23111 Handouts 07-08 of 12]

Acknowledgements

- These slides are adaptations (mostly minor, but some major) of material authored and made available to instructors by **Thomas Connolly and Carolyn Begg** to accompany their textbook **Database Systems: A Practical Approach to Design, Implementation, and Management, 5th Edition. Addison-Wesley, 2010, 978-0-321-52306-8** and by **Raghu Ramakrishnan and Johannes Gehrke** to accompany their textbook **Database Management Systems, 3rd ed., McGraw-Hill, 2002, 978-0071230575**
- Copyright remains with them and the publishers, whom I thank.
- Some slides had input from Sandra Sampaio, whom I thank.
- All errors are my responsibility.
- Note that, for copyright reasons, the mandatory readings regarding the topics treated here come from **Raghu Ramakrishnan and Johannes Gehrke. Database Management Systems, 3rd ed., McGraw-Hill, 2002, 978-0071230575**

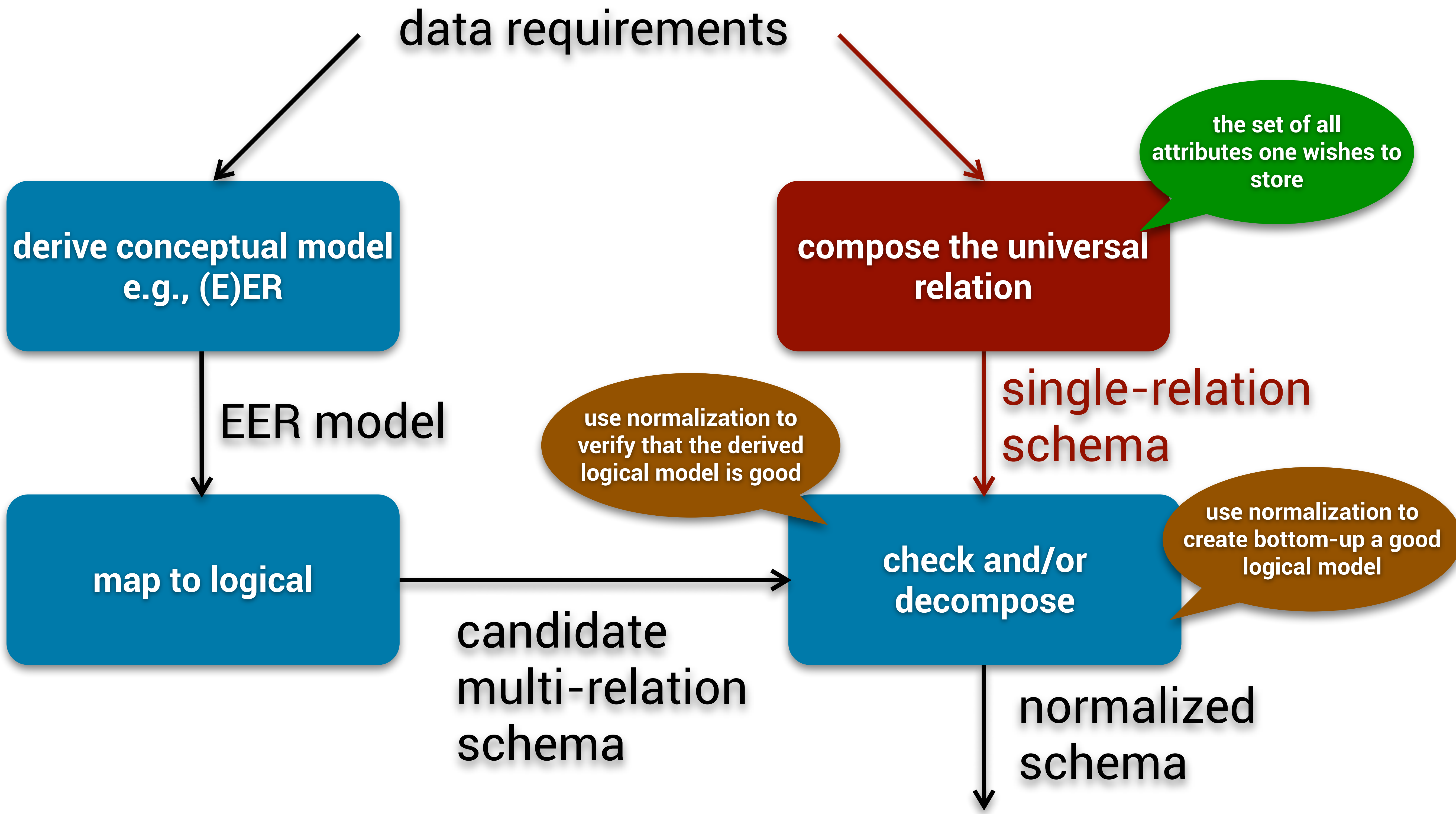
In Previous Handouts

- We learned how data models lead to a distinction between schemas and instances that enables a logical view of the data.
- We learned about the relational approach to logical data modelling.
- We learned about the relational algebra and SQL, both its DDL and DML capabilities and its querying constructs.
- We learned of the practical benefits of performing conceptual modelling before logical design and why the ER approach serves well this purpose.
- We learned how a conceptual model can be mapped into a logical model that is capable of being implemented in a relational DBMS.

In This Pair of Handouts

- We assume that a logical model in the form of a relational schema already exists.
- For example, it may have been designed from scratch rather then derived from an (E)ER model.
- Or it may have been originally derived from an (E)ER model but later modified, added to, and so on.
- The question the designer must answer in these scenarios is: "*Is this a good logical model?*"
- We'll learn criteria that allow us to give a more specific and useful answer to the question above.
- We'll also learn how, building on the notion of functional dependency (FD), we can use normalization to systematically improve a logical model if it fails to meet such quality criteria .

Two Routes To Good Design


data requirements
derive conceptual model e.g., (E)ER
EER model
map to logical
candidate multi-relation schema
compose the universal relation
the set of all attributes one wishes to store
single-relation schema
use normalization to verify that the derived logical model is good
check and/or decompose
use normalization to create bottom-up a good logical model
normalized schema

Normal Forms

- **Normalization** is the process of decomposing unsatisfactory (i.e., potentially improvable) relations by breaking up their attributes into smaller relations.
- The keys and FDs (more on which, later) of a relation determine its normal form.
- A **normal form** indicates the particular quality level the corresponding relation schema is at in its design.
- **1NF**: A relation is in first normal form if every field contains only atomic values (i.e., no structure, and neither lists nor sets, etc.)
- **2NF**, **3NF**, and **BCNF** are normal forms that are defined in terms of the keys and FDs of a relation schema.
- 3NF and BCNF are the targets in good design, but there exist even more stringent normal forms, which we won't cover.

Practical Considerations

- Normalization is carried out in practice with the goal that the resulting schema designs are of high quality, i.e., have desirable properties.
- The practical utility of ever more stringent normal forms becomes questionable when the constraints on which they are based are hard to understand or to detect.
- Database designers do not need to normalize to the highest possible normal form.
- Some times, performance considerations pragmatically recommend denormalizing (or not normalizing any further).
- Denormalization is the process of storing the join of higher normal form relations as a base relation in a lower normal form.

The Evils of Redundancy

- Redundancy is at the root of several problems associated with relational schemas:
  - redundant storage
  - update (i.e., insert, delete and modify) anomalies
- Integrity constraints, and FDs in particular, can be used to identify schemas with such problems and to suggest refinements.
- The main refinement technique is **decomposition** (e.g., replacing a relation ABCD with two relations, such as AB and BCD, or, alternatively ACD and ABD).
- Decomposition should be used only if we have suitable answers to questions such as:
  - Is there a benefit in decomposing a relation? E.g., will it reduce redundancy?
  - What costs (if any) does the decomposition bring? E.g., could it increase the need to run joins, which are normally expensive (on AB and BCD, or ACD and ABD, in order to reconstruct ABCD, in the examples above)?

Data Redundancy and Update Anomalies

- The associated problems can be illustrated by comparing
  - the Staff and Branch relations
  - and the alternative StaffBranch relation

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

these two?

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

or just this one?

StaffBranch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

Data Redundancy and Update Anomalies

- The StaffBranch relation has redundant data:
  - The details of a branch are repeated for every member of staff.
- In contrast, in the Branch relation:
  - The branch information appears only once for each branch in the Branch relation.
  - Only the branch number (branchNo) is repeated in the Staff relation.
  - But we know, from our study of (E)ER, that this is to represent the assignment of each member of staff to a branch.

Data Redundancy and Update Anomalies

- Relations that contain redundant information may potentially suffer from update anomalies.
- Update anomalies can result from all of:
  - Insertion
  - Deletion
  - Modification

Data Redundancy and Update Anomalies

- In the example we are using, an insertion anomaly is as follows:
    - We can't simply insert a new member of staff: we must have also already assigned her/him to a branch.
    - We also can't have branches without at least one member of staff assigned to it.

Data Redundancy and Update Anomalies

- In the example we are using, a deletion anomaly is as follows:
  - If we were to delete the last staff assigned to a branch, we'd have to delete the branch itself too.

Data Redundancy and Update Anomalies

- In the example we are using, a modification anomaly is as follows:
  - Assume we wish to change the values of an attribute of a particular branch (say, the address of branch B003).
  - We are then compelled to update the address field in the tuple of every staff assigned to B003.
  - If we don't, the database is left in an inconsistent state because branch B003 will seem to have different addresses depending on which staff tuple we take the branch address from.

Notation and Terminology

- In normalization theory, we often write a set of attributes, e.g., R = {A,B,C} as the string ABC.
- If t is a tuple in R and A is a set of attributes in the schema of R, t[A] denotes the projection with attribute list A over t.
- For example, if R has schema ABC, and t = (1,2,3), then t[AB] is (1,2). Also, if we write S = AB, then t[S] = (1,2).
- A **superkey** S of a relation schema R is a set of attributes of R with the property that for no two tuples t_1 and t_2 we can have $t_1[S] = t_2[S]$.
- A superkey is, therefore, unique but not irreducible.

Notation and Terminology

- A **key** K is a superkey with the additional property that removal of any attribute from K will cause K not to be a superkey (i.e., not unique) any more.
- If a relation schema has more than one key, each is called a **candidate key**.
- One of the candidate keys is arbitrarily designated to be the **primary key**, and the others are called **secondary keys**.
- A **prime attribute** must be a member of some candidate key.
- A **nonprime attribute** is not a member of any candidate key.

Functional Dependencies

- A **functional dependency (FD)**

  $$X \rightarrow Y$$

  holds over a relation R if, for every allowable instance *r* of R and for every pair of tuples t_1 and t_2 in *r*, it follows that

  $$t_1[X] = t_2[X] \text{ implies } t_1[Y] = t_2[Y]$$

- So, given two tuples in r, if the X values agree, then the Y values must also agree, where X and Y are sets of attributes.
- An FD $X \rightarrow A$ is said to be **trivial** if every attribute in A also appears in X.

Functional Dependencies (FDs)

- An FD is a statement about all allowable relations.
  - It must be identified based on the semantics of application.
  - Given some allowable instance r of R and an FD f, we can try to use r to check if f <u>does not hold</u> over R, but we cannot normally use r to check if f <u>does hold</u> over R.
- If K is a candidate key for R, then K→R
  - However, K→R does not mean K is **minimal**
  - K could be a superkey, and therefore, possibly reducible

Using FDs to Identify Update Anomalies: An Example

- Consider the following example relation schema:
  - Hourly_Employees (SSN, name, location, rating, hourly_wages, hours_worked)
- We will denote this relation schema by listing its attributes in abbreviated form as
  - SNLRWH
  - Recall, this is really the set of attributes {S,N,L,R,W,H}.
  - We will also refer to SNLRWH by the relation name Hourly_Employees
- Here are some FDs on Hourly_Employees:
  - SSN is the key: S→SNLRWH
  - rating determines hourly_wages: R→W

An Example

- Problems due to R→W :
  - Update anomaly: Can we change W in just the 1st tuple of SNLRWH?
  - Insertion anomaly: What if we want to insert an employee and don't know the hourly wage for his rating?
  - Deletion anomaly: If we delete all employees with rating = 5, we lose the information about the wage for rating = 5.
- Will two smaller tables be better?

S	N	L	R	W	H
123-22-3666	Attishoo	48	8	10	40
231-21-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

S	N	L	R	H
123-22-3666	Attishoo	48	8	40
231-21-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

R	W
8	10
5	7

Reasoning About FDs

- Given some FDs, we can usually infer additional FDs:
  - SSN→dept_id and dept_id→location together imply SSN→location
- An FD f is implied by a set of FDs F if f holds whenever all FDs in F hold.
- We denote by F^+ the **closure** of F, i.e., the set of all FDs that are implied by F by a set of axioms.
- FDs have been axiomatized.
- The set of axioms is known as **Armstrong's Axioms (AA)**.
- Let X, Y, Z be sets of attributes:
  - **Reflexivity**: If $X \subseteq Y$, then $Y \rightarrow X$
  - **Augmentation**: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z
  - **Transitivity**: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- This is a sound and complete axiomatization of FDs

Reasoning About FDs

- Sound
  - Given a set of FDs F specified on a relation R, any dependency that we can infer from F using **AA,** holds in every instance of R that satisfies the dependencies in F.
- Complete
  - Using **AA** repeatedly to infer dependencies from F, until no more dependencies can be inferred, results in the set of all dependencies that can be inferred from F

Reasoning About FDs

- The following can be derived from **AA**:
  - **Union**: If X→Y and X→Z, then X→YZ
  - **Decomposition**: If X→YZ, then X→Y and X→Z
  - **Composition**: If X→Y and U→V, then XU→YV
  - **Pseudotransitivity**: If X→Y and YZ→U, then XZ→U
- Note that Reflexivity generates what we have called trivial FDs.
- Another axiom that generates a particular type of trivial FD is:
  - **Self-determination**: X→X

Reasoning About FDs

- Consider Contracts(contract_id, supplier_id, project_id, dept_id, part_id, quantity, value), denoted CSJDPQV, with the following meaning:
  - The contract C with value V is an agreement that supplier S will supply Q items of part P to project J associated with department D.
- The following are known to hold:
  - C is the key, so $C \rightarrow CSJDPQV$
  - A project purchases each part using a single contract, so $JP \rightarrow C$
  - A department purchases at most one part from a supplier, so $SD \rightarrow P$

Reasoning About FDs

- From which we can infer using **AA**:
  - JP→C, C→CSJDPQV and transitivity imply JP→CSJDPQV
  - SD→P and augmentation implies SDJ→JP
  - SDJ→JP, JP→CSJDPQV and transitivity imply SDJ→CSJDPQV
- Notice that (unlike multiplication, say) we cannot cancel out, i.e., SD→CSDPQV is not equivalent to, e.g., SDJ→CSJDPQV
- From C→CSJDPQV, using decomposition, we can infer
  - C→C, C→S, C→J, and so on.
- Using reflexivity, we can state several trivial FDs, e.g.,
  - JP→J, JP→P, and so on.

Reasoning About FDs

- Computing the closure of a set of FDs can be expensive, since the size of the closure is exponential in the number of attributes.
- Typically, we just want to check if a given FD $X \rightarrow Y$ is in the closure F^+ of a set of FDs F.
- An efficient check is as follows:
  - First, compute the attribute closure of X (denoted X^+) w.r.t. F :
    - This is the set of all attributes A such that $X \rightarrow A$ is in F^+
    - There is a linear time algorithm to compute this.
  - Then, check if Y is in X^+
- For example, does $F = \{A \rightarrow B, B \rightarrow C, CD \rightarrow E\}$ imply $A \rightarrow E$? In other words, is $A \rightarrow E$ in F^+?
- We first must check whether E in A^+?

Computing Attribute Closures

```
// compute the attribute closure $X^+$ given a set of FDs F
$X^+$ := X
repeat:
        if there is an FD $U \rightarrow V$ in F such that $U \subseteq X^+$:
            $X^+$ := $X^+ \cup V$
until there is no change in $X^+$
```

An Example

- Does F = {A→B, B→C, CD→E} imply A→E?
  - We start with A^+ = A
  - We pattern-match A→? to A→B, so we add B to A^+ = AB
  - The attribute closure has changed, so we continue
  - We pattern-match B→? to B→C, so we add C to A^+ = ABC
  - The attribute closure has changed, so we continue
  - No subset U of A^+ pattern-matches U→?, so we add no more attributes
  - The attribute closure has not changed, so we stop
  - Since E is not in A^+, A→E is not in F^+, i.e., it is not implied by F

FDs for Normalization

- In an FD X→A, X is sometimes called the **determinant**.
- Determinants ideally will have the smallest set of attributes needed for the FD to hold.
- FDs with minimal determinants are referred to as **full** and are preferred for normalization.
- In a full FD X→A, A is not functionally dependent on any subset of X.
- FDs whose determinants are not minimal are referred to as **partial**.

Full v. Partial FDs

- Let f = staffNo, sName → branchNo
- Let f' = staffNo → branchNo
- Then, given f', f is not full, since staffNo ⊂ staffNo, sName

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

StaffBranch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

Identifying Functional Dependencies

- Identifying all FDs between a set of attributes is relatively simple if the meaning of each attribute and the relationships between the attributes are well understood.
- This information should be provided by the enterprise in the form of discussions with users and/or documentation such as the users' requirements specification.

Identifying Functional Dependencies

- However, if the users are unavailable for consultation and/or the documentation is incomplete then it may be necessary for the database designer to use their common sense and/or experience to provide the missing information temporarily.

Identifying Functional Dependencies Analytically

- To identify FDs analytically, we examine semantics of attributes in StaffBranch relation (see database instance shown before).
- We may need to assume (for the time being) that position held and branch determine a member of staff's salary.
- With this information, we identify the FDs for the StaffBranch relation as:
  - staffNo → sName, position, salary, branchNo, bAddress
  - branchNo → bAddress
  - bAddress → branchNo
  - branchNo, position → salary
  - bAddress, position → salary

Identifying Functional Dependencies Empirically

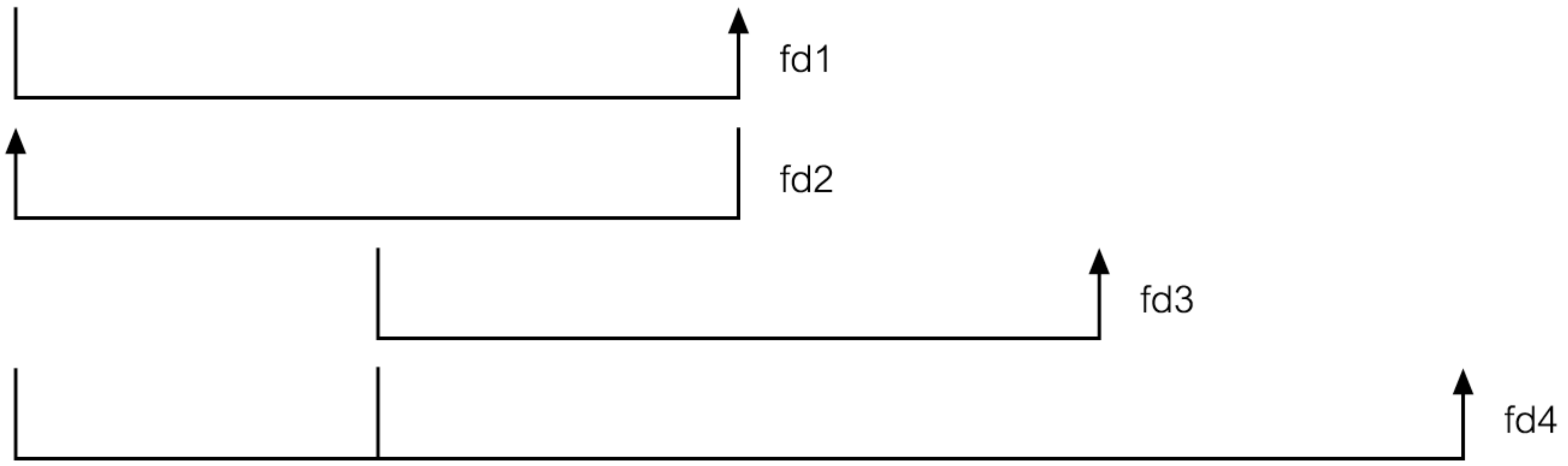
- We take a sample of the data.
- We then consider the attributes A, B, C, D, and E in the Sample relation (shown in the next slide).
- It is crucial to establish that the sampled data values are truly representative of all possible values that can be held by attributes A, B, C, D, and E.
- For these slides, assume they are despite the very small amount of data shown in the next slide.

Identifying Functional Dependencies Empirically

Sample the data!

Sample Relation

A	B	C	D	E
a	b	z	w	q
e	b	r	w	p
a	d	z	w	t
e	d	r	w	q
a	f	z	s	t
e	f	r	s	t



Identifying Functional Dependencies Empirically

- If so, then the FDs between attributes A to E in the Sample relation are:
  - $A \rightarrow C$ (fd1)
  - $C \rightarrow A$ (fd2)
  - $B \rightarrow D$ (fd3)
  - $A, B \rightarrow E$ (fd4)

Identifying the Primary Key for a Relation Using FDs

- One of the main purposes of identifying a set of FDs for a relation is to specify the set of integrity constraints that must hold on a relation.
- The first integrity constraint to consider is the primary key (or identity) constraint.
- We need to identify the candidate keys, one of which we must select to be the primary key for the relation.

Identifying the Primary Key for a Relation Using FDs

- Consider again the five FDs in the StaffBranch relation:
  - staffNo → sName, position, salary, branchNo, bAddress
  - branchNo → bAddress
  - bAddress → branchNo
  - branchNo, position → salary
  - bAddress, position → salary

- Now, we consider the determinants, i.e.:
  - staffNo
  - branchNo
  - bAddress
  - (branchNo, position)
  - (bAddress, position)

Identifying the Primary Key for a Relation Using FDs

- To identify all candidate key(s):
  - we identify all the (possibly singleton) sets of attributes that uniquely identify each tuple in the relation
  - we consider the requirement that all attributes that are not part of a candidate key should be non-trivially functionally dependent on the key.
- We conclude that the only candidate key and therefore primary key for StaffBranch relation, is the full determinant staffNo.
- All other determinants are functionally dependent on staffNo and therefore violate the requirement.

Identifying the Primary Key for a Relation Using FDs

- Now, consider again the Sample relation above.
- It has the four FDs:
  - $A \rightarrow C$
  - $C \rightarrow A$
  - $B \rightarrow D$
  - $A, B \rightarrow E$
- The determinants are:
  - A
  - B
  - C
  - (A, B)
- The only determinant that functionally determines all the other attributes of the relation is (A, B).
- (A, B) is therefore the primary key for the relation.

Normal Forms

- Returning to the issue of schema refinement, the first question to ask is whether any refinement is needed.
- If a relation is in a certain normal form (BCNF, 3NF etc.), it is known that certain kinds of problems are avoided/minimized.
- This can be used to help us decide whether decomposing the relation will help.
- As well as detecting redundancies, the presence of FDs in a relation helps us identify in which normal form the relation is and decide whether decomposing the relation is worthwhile.

Normal Forms

- Consider a relation R with 3 attributes ABC.
  - In the absence of integrity constraints, any set of triples is a legal instance and there is no potential for redundancy, so if no FDs hold, there is no redundancy.
  - However, if A→B, and if several tuples could have the same A value, they'll all have the same B value.
- BCNF is often a good target in practice.

Boyce-Codd Normal Form (BCNF)

- A relation R with FDs F is in **BCNF** if, for all X→A in F^+,
  - A ∈ X (i.e., it is 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.
  - There is no dependency in R that can be predicted using FDs alone.
  - If we are shown two tuples that agree upon the X value, we cannot infer the A value in one tuple from the A value in the other.
  - If XYA is in BCNF, then the two tuples must be identical.

X	Y	A
x	y1	a
x	y2	?

Boyce-Codd Normal Form (BCNF)

- An easier-to-remember, informal way of characterizing BCNF was popularized by Chris J. Date as follows:
  - A relation is in BCNF if "*Every fact is a fact about the key, the whole key and nothing but the key.*".
  - (To which a wit appropriately added, "*So help me Codd!*")
- But he also warns that this presupposes that there is just one key, which may not be true.

Third Normal Form (3NF)

- A relation R with FDs F is in **3NF** if, for all X→A in F^+,
  - A ∈ X (i.e., it is a trivial FD), or
  - X contains a key for R, or
  - A is part of some key for R.
- Minimality of a key (see the last condition above) is crucial for characterization:
  - If R is in BCNF, it is obviously in 3NF, but the reverse is not always true.
  - If R is in 3NF, some redundancy is possible.

Third Normal Form (3NF)

- 3NF is a compromise, often used when BCNF is not desirable/achievable (either because there is no “good” decomposition, or because performance considerations do not recommend BCNF).
- Lossless-join, dependency-preserving decomposition (discussed later) of a relation into a collection of 3NF relations is always possible.

What Does 3NF Achieve?

- If 3NF is violated by X→A, one of the following holds:
  - X is a subset of some key K
    - We store (X, A) pairs redundantly.
  - X is not a proper subset of any key.
    - There is a chain of FDs K→X→A, which means that we cannot associate an X value with a K value unless we also associate an A value with an X value.

What Does 3NF Achieve?

- Even if a relation is in 3NF, problems could arise.
    - For (a somewhat contrived) example, imagine a relation Reservations = (<u>sailor_id, boat_id, date</u>, card) = SBDC with the meaning that a sailor s has reserved the boat b on date d and paid with card c.
    - The following FD is given: S→C, i.e., a sailor uses a unique card to pay for reservations.
    - S is not a key and C is not part of a key, indeed SBD is the only key.
    - This relation is therefore not in 3NF: SC pairs are stored redundantly.
    - However, if cards uniquely identify their owners, i.e., C→S, then CBS is now also a key, and if so, S→C no longer violates the 3NF conditions.
    - Nonetheless, SC pairs are redundantly stored.
- Thus, 3NF is indeed a compromise relative to BCNF.

Decomposition of a Relation Schema

- Suppose that relation R contains attributes A1 ... An.
- A decomposition of R consists of replacing R by two or more relations such that:
  - Each new relation schema 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 schemas produced by the decomposition, instead of instances of R.
  - E.g., on a previous example, we can decompose SNLRWH into SNLRH and RW.

Example Decomposition

- Decompositions should be used only when needed.
  - SNLRWH has FDs S→SNLRWH and R→W
  - The second FD causes violation of 3NF: W values repeatedly associated with R values.
  - The easiest way to fix this is to create a relation RW to store these associations, and to remove W from the main schema:
  - I.e., we decompose SNLRWH into SNLRH and RW
- The information to be stored consists of SNLRWH tuples.
- If we just store the projections of these tuples onto SNLRH and RW, are there any potential problems that we should be aware of?

Problems with Decompositions

- There are three potential problems to consider:
  - Some queries become more expensive.
    - e.g., How much did Joe earn? (salary = W∗H)
  - Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation.
    - Fortunately, this is not the case in the SNLRWH example.
  - Checking some dependencies may require joining the instances of the decomposed relations.
    - Fortunately, this is not the case in the SNLRWH example.
- So, there is a tradeoff between incurring these problems and avoiding redundancy and update anomalies.

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:

  $$\pi_X(r) \bowtie \mu_Y(r) = r$$

  - It is always true that $r \subseteq \pi_X(r) \bowtie \mu_Y(r)$
  - In general, the other direction (i.e., $\supseteq$) does not hold.
  - The decomposition is lossless-join only if it does.
- The definition can be extended to decomposition into more than two relations in a straightforward way.
- It is essential that all decompositions used to deal with redundancy be lossless-join to ensure reconstructibility.

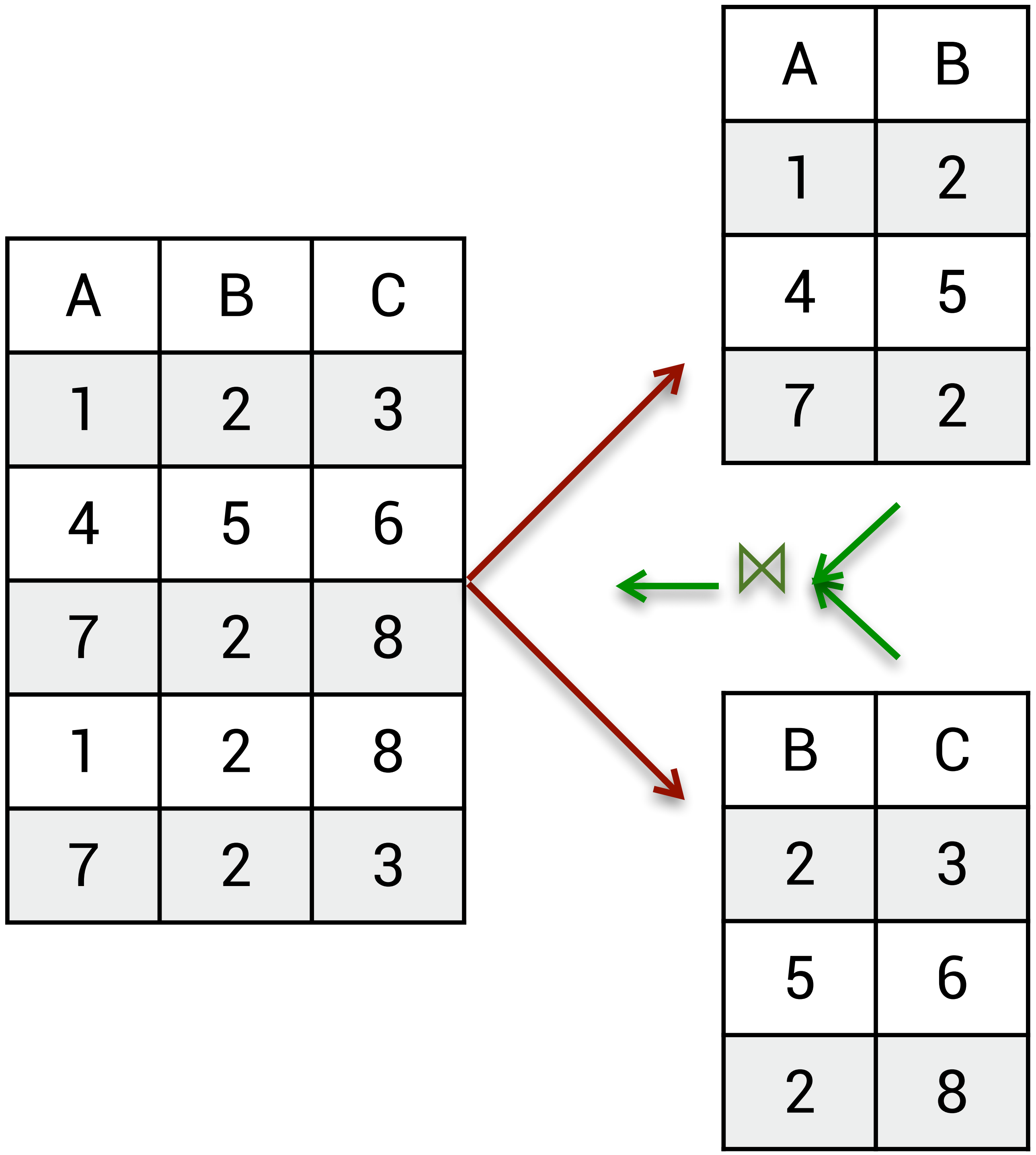
More on Lossless Join

- The decomposition of R into X and Y is lossless-join w.r.t. F if and only if the closure of F contains:

  $X \cap Y \rightarrow X$, or

  $X \cap Y \rightarrow Y$

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



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

A	B
1	2
4	5
7	2

B	C
2	3
5	6
2	8

Dependency-Preserving Decomposition

- Consider CSJDPQV, C is key, JP→C and SD→P.
- A lossless-join decomposition is CSJDQV and SDP
- The problem is that it is now harder to enforce JP→C: it requires a join.
- We say the decomposition above is not **dependency-preserving**.
- Intuitively, a dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on insertion or modification (note that deletions cannot cause FD violation).

Dependency-Preserving Decomposition

- To define it more formally, we need the notion of a projection of FDs.
- Let R be decomposed into two relations with attribute sets X and Y and let F be a set of FDs over R.
- The **projection** F_X of F on X is the set of FDs in F^+ that involve only attributes in X, i.e., $F_X = \{U \rightarrow V \in F^+ \mid U \subseteq X \text{ and } V \subseteq X\}$.
- A decomposition of R into X and Y is **dependency-preserving** if $(F_X \cup F_Y)^+ = F^+$
  - i.e., if we compute the closure of the union of the projections of F on X and of F on Y, we end up with the closure of F.

Dependency-Preserving Decomposition

- It is important to consider F^+, not F, in the definition:
  - Assume R with schema ABC is decomposed into AB and BC.
  - Assume $F = \{A \rightarrow B, B \rightarrow C, C \rightarrow A\}$.
  - Of these, $A \rightarrow B$ is in F_{AB} and $B \rightarrow C$ is in F_{BC}.
  - Is this dependency-preserving? Is $C \rightarrow A$ preserved? It is not implied by the FDs in F_{AB} and in F_{BC}.
- $F^+ = F \cup \{A \rightarrow C, B \rightarrow A, C \rightarrow B\}$, therefore $F_{AB} \cup F_{BC}$ contains $A \rightarrow B$, $B \rightarrow C$, $B \rightarrow A$, $C \rightarrow B$, and the closure of $F_{AB} \cup F_{BC}$ contains $C \rightarrow A$ (by transitivity from $C \rightarrow B$ and $B \rightarrow A$).
- So, $C \rightarrow A$ is preserved and the proposed decomposition is dependency-preserving.

Dependency-Preserving Decomposition

- A direct application of the definition gives us an algorithm to test whether a decomposition is dependency-preserving but the algorithm is exponential on the number of FDs (but a polynomial algorithm exists).
- We have seen that some lossless-join decompositions are not dependency-preserving.
- Likewise, there can be dependency preserving decompositions that are not lossless-join.
- A simple example is a relation ABC with FD $A \rightarrow B$ that is decomposed into AB and BC.

Limits to Normalization

- If a relation schema R is not in BCNF, it it possible to obtain a lossless-join decomposition of R into a collection of relation schemas in BCNF.
- There may be no dependency-preserving decomposition of R into a collection of relation schemas in BCNF.
- It is always possible to decompose R into a collection of lossless-join, dependency-preserving relation schemas in 3NF.

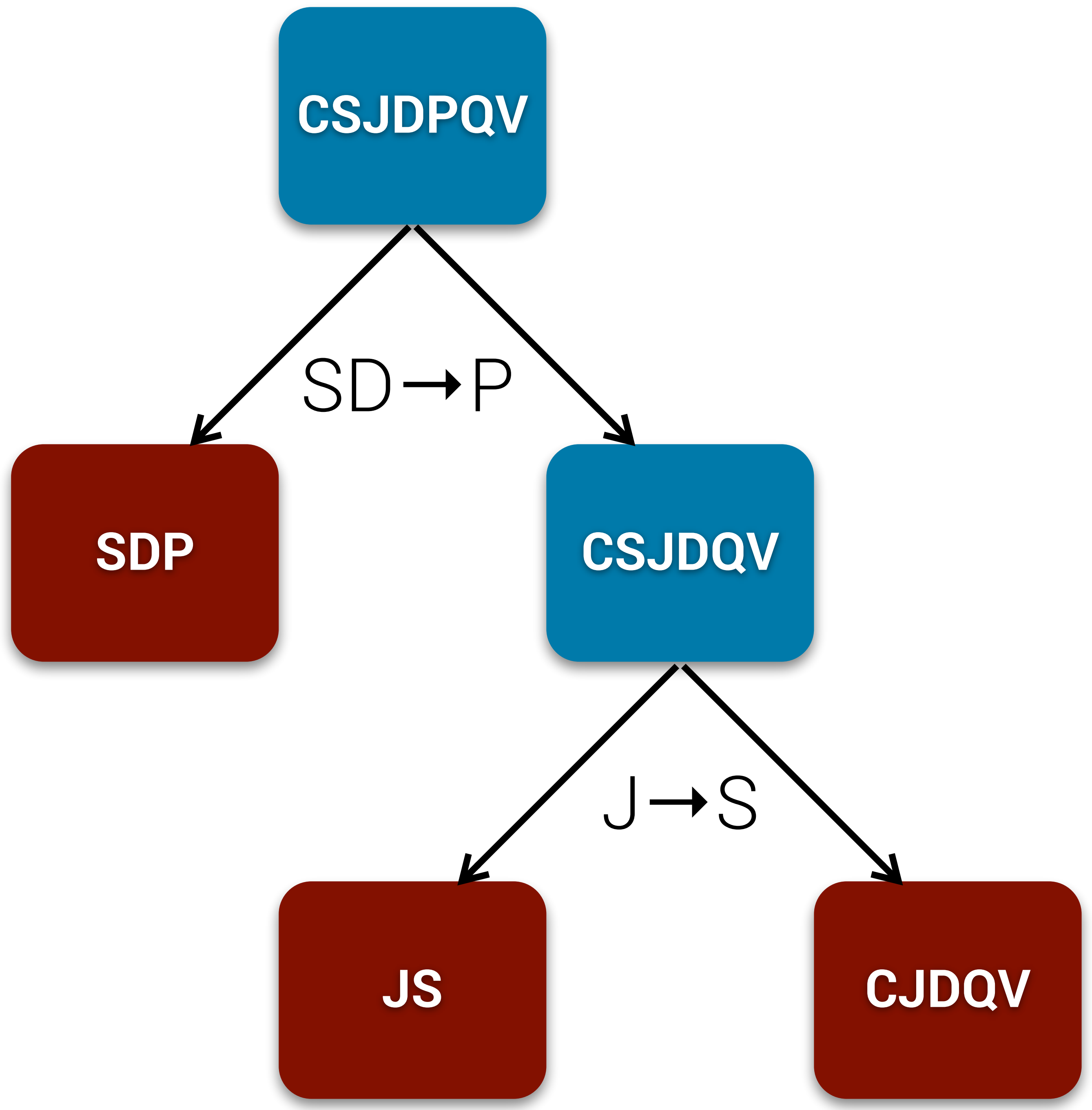
Decomposing into Lossless-Join BCNF Schemas

- Suppose that R is not in BCNF.
- Let $X \subset R$, A be a single attribute in R and $X \rightarrow A$ be an FD that leads to a violation of BCNF.
- If so, decompose R into $R \setminus A$ and XA.
- If either $R \setminus A$ or XA is not in BCNF, decompose them further recursively.

An Example Decomposition into BCNF

- Consider again Contracts with attributes CSJDPQV and key C.
- Suppose further that each project deals with a single supplier, so the set of FDs is {JP→C, SD→P, J→S}.
- Because P is dependent on non-prime attributes, CSJDPQV is not in BCNF.
- We therefore use SD→P to guide the decomposition and thereby obtain SDP and CSJDQV, where SDP is in BCNF.
- Because S is dependent on a non-prime attribute, CSJDQV is not in BCNF.
- We therefore use J→S to guide the decomposition and thereby obtain JS and CJDQV.
- The FD C→JDQV holds over CJDQV, and the only other FDs that hold are obtained from that FD by augmentation, so all FDs that hold contain a key in the left-hand side.
- Thus, SDP, JS and CJDQV are in BCNF, and this collection is a lossless-join decomposition of CSJDPQV over the given FDs.

The Example Decomposition, Graphically

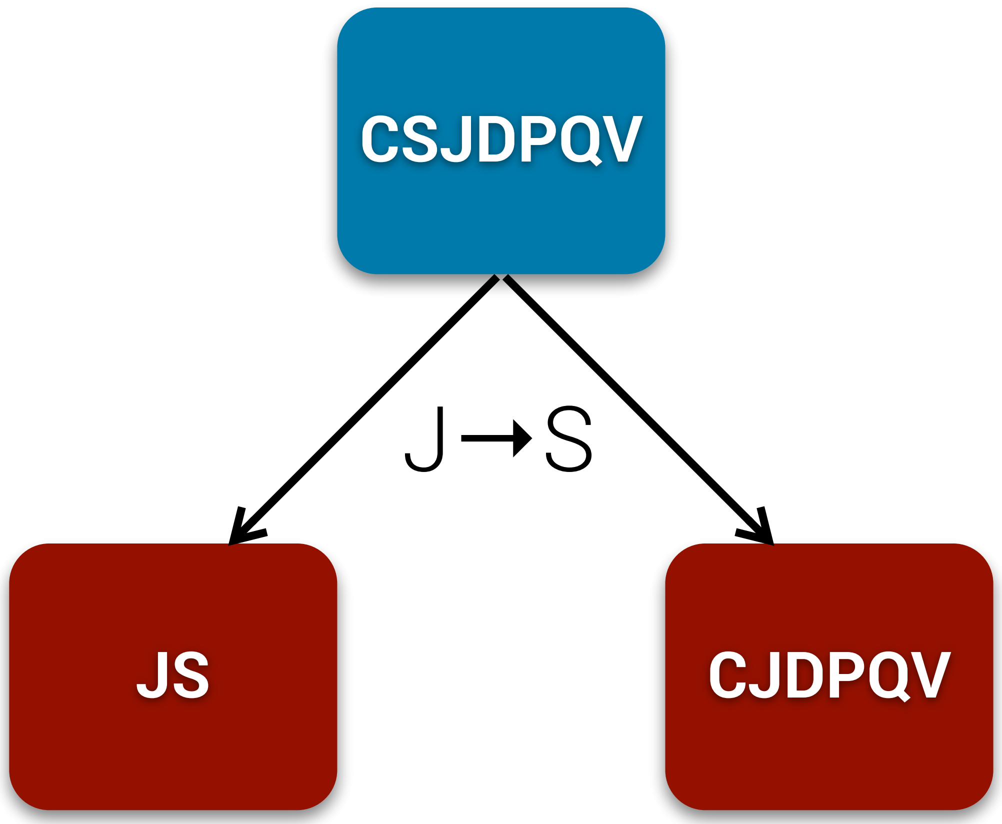


Revisiting Redundancy in BCNF

- The example decomposition is not dependency-preserving.
- Intuitively, J→P cannot be enforced without a join.
- So, there is the dilemma between eliminating all the redundancy and incurring the cost of performing relatively-expensive joins in order to enforce FDs at insertion and update times.
- For example, if we add a relation CJP, we avoid the need for joins at the price of more redundancy.
- Which of the two is the best design will depend on an analysis of how typical workloads will lead to a decision as to which is more onerous: the need for joins or the management of redundancy?

Alternatives When Decomposing into BCNF

- If several dependencies cause BCNF violation, different choices on which to use as guide will lead to different final outcome for the algorithm.
- Suppose that rather than using SD→P as guide, we use J→S.
- We obtain JS and CJDPQV.
- The FDs that hold over CJDPQV are JP→C and the key dependency C→CJDPQV, and since JP is a key, CJDPQV is in BCNF.
- This is a different outcome from the one we had before.



Decomposition into 3NF

- It is possible to algorithmically achieve a decomposition into a set of lossless-join, dependency-preserving relation schemas in 3NF.
- For details, see Chapter 19 in R. Ramakrishnan, J. Gehrke, *Database Management Systems*, 3rd ed., McGraw-Hill, 2002, 978-0071230575

Guidelines for Schema Refinement

- If a relation is in BCNF, it is free of redundancies that can be detected using FDs.
- Thus, trying to ensure that all relations are in BCNF is a good heuristic.
- If a relation is not in BCNF, we can try to decompose it into a collection of BCNF relations.
- We must consider whether all FDs are preserved.
- If a lossless-join, dependency-preserving decomposition into BCNF is not possible (or unsuitable, given typical queries), we should consider decomposition into 3NF.
- Decompositions should be carried out and/or re-examined while keeping performance requirements in mind.

Revision: 1NF : Ensure all attributes are atomic

A relation schema R is in first normal form (1NF) if every attribute A in R is atomic.

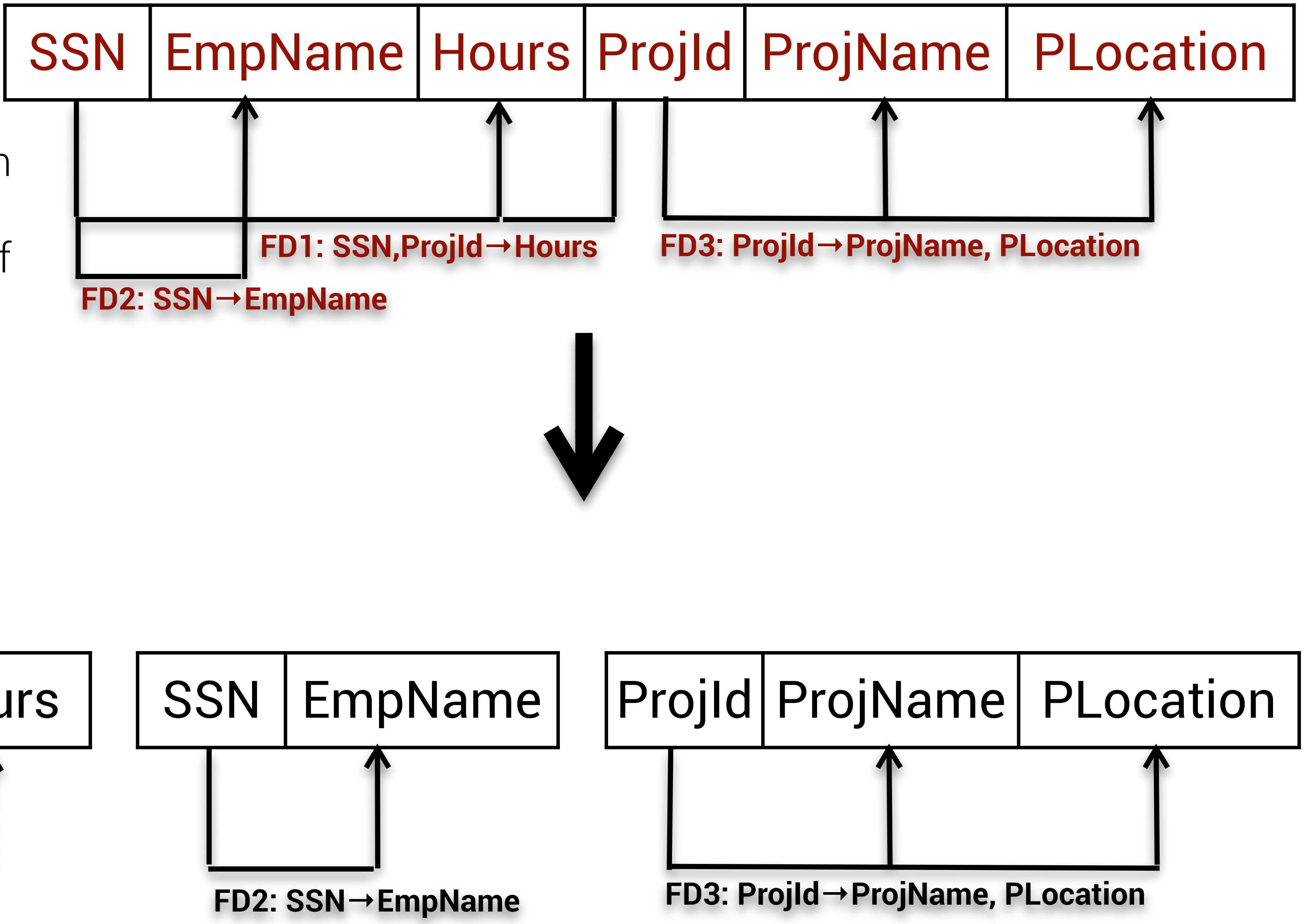
DeptName	DeptId	DeptMgr	DeptLocation
Research	5	287348	{York, Leeds, Bath}
Administration	2	456765	{Newcastle}
Headquarters	1	759849	{London}

DeptName	DeptId	DeptMgr
Research	5	287348
Administration	2	456765
Headquarters	1	759849

DeptId	DeptLocation
5	York
5	Leeds
5	Bath
2	Newcastle
1	London

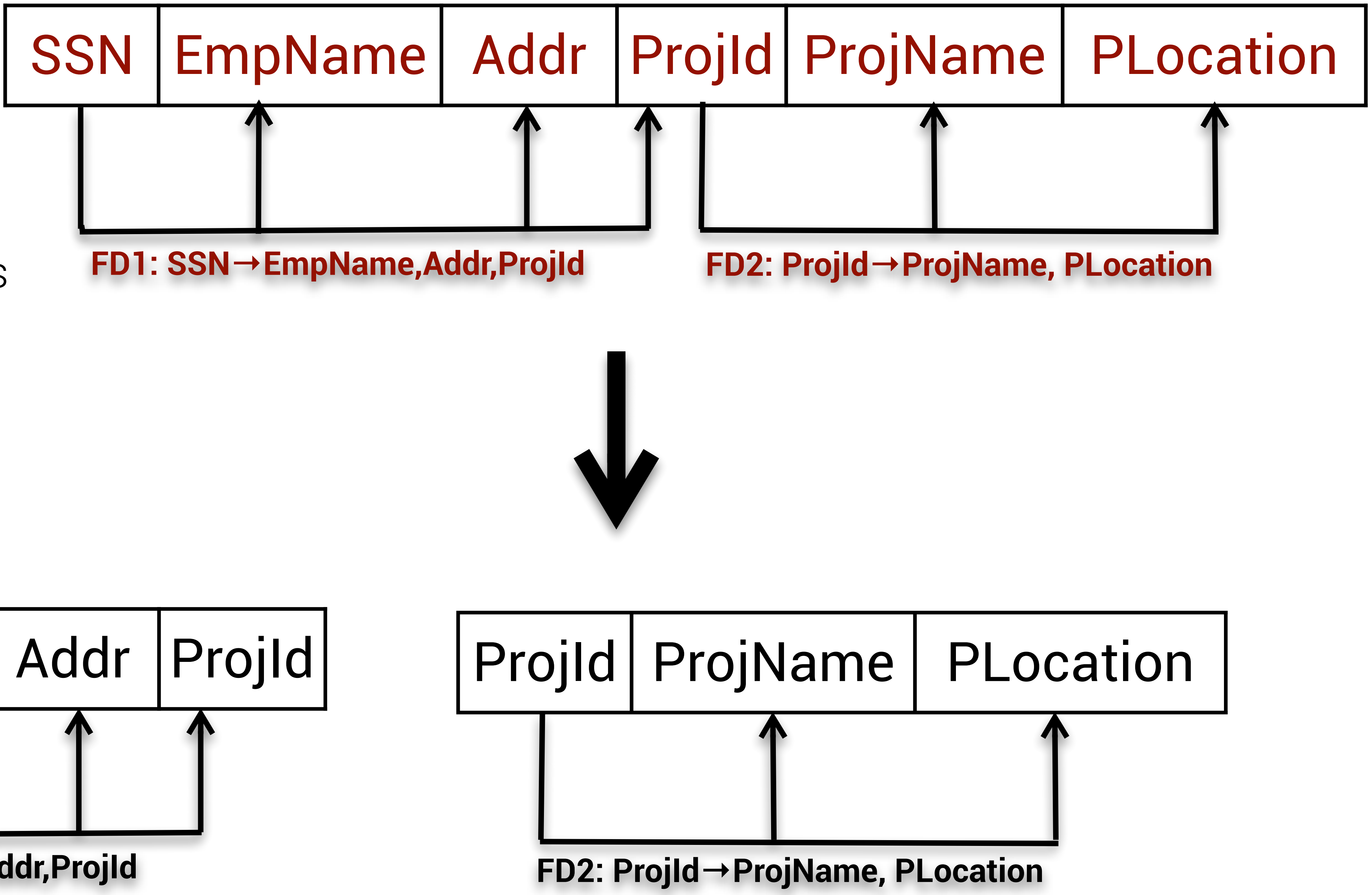
Revision: 2NF : Ensure non-prime attributes are dependent on the primary key

A relation schema R is in second normal form (2NF) if it is in 1NF and if every non-prime attribute A in R is fully functionally dependent on the primary key.



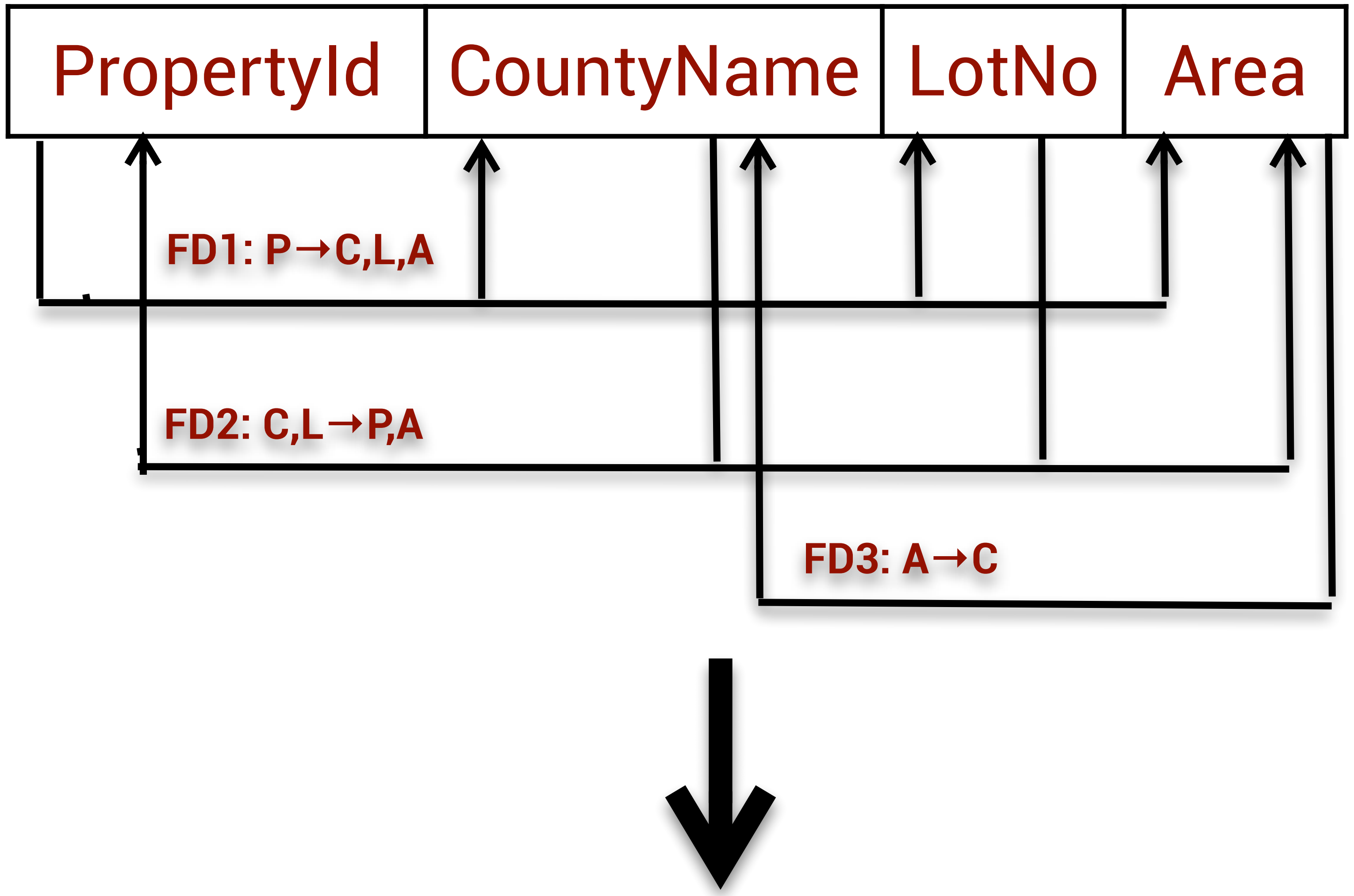
Revision: 3NF : Ensure non-prime attributes are dependent on the primary key

A relation relation schema R is in third normal form (3NF) if it is in 2NF and no non-prime attribute A in R is transitively dependent on the primary key.


SSN
EmpName
Addr
ProjId
ProjName
PLocation
FD1: SSN→EmpName,Addr,ProjId
FD2: ProjId→ProjName, PLocation
SSN
EmpName
Addr
ProjId
FD1: SSN→EmpName,Addr,ProjId
ProjId
ProjName
PLocation
FD2: ProjId→ProjName, PLocation

Revision: BCNF : Ensure non-prime attributes are dependent on the primary key, the whole primary key and nothing but the primary key

A relation schema R is in Boyce-Codd Normal Form (BCNF) if it is in 3NF and if whenever an FD X➞A holds in R, then X is a superkey of R.



PropertyId	LotNo	Area

Area	CountyName

In The Next Handout

- We'll learn some advanced features that are available in SQL and how they can be used to capture more application semantics in a DBMS environment.