Databases and Info Systems

Normalisation

Dr. Seán Russell
sean.russell@ucd.ie,

School of Computer Science,
University College Dublin

March 27, 2020

Table of Contents

- **Normalisation** is the process of transforming data from a problem into relations, ensuring **data integrity** and eliminating **data redundancy**.
  - Data Integrity: Database is consistent and satisfies all constraint rules.
  - Data Redundancy: If data can be found in two places in a single database (direct redundancy) or calculated using data from different parts of the database (indirect redundancy) then redundancy exists.
- Normalisation should remove redundancy, but not at the expense of data integrity.

- If redundancy exists then this can cause problems during normal database operations:
    - When data is inserted into the database, the data must be duplicated wherever redundant versions of that data exists.
    - When data is updated, all redundant data must be updated at the same time.

- An **integrity constraint** is a rule that restricts the values that may be present in the database.
- **Entity integrity**: the rows (or tuples) in a relation represent entities, and each one must be uniquely identified.
  - We must have a **primary key** that must have a unique non-null value for every row
- **Referential integrity**: Involves the foreign keys
  - Foreign keys tie the relations together, so it is important that the links are correct.
  - Every foreign key must either be null, or its value must be the actual value of a key in another relation

- There are many benefits to normalised database
  - Saves space
  - Prevents anomalies
  - Avoid NULL values

- While many modern computers have a lot of disk space, this can still be of benefit
- If we can make our database small enough to fit in memory, queries will be much quicker (no disk reads)
- This is also important for apps on devices with less memory, like phones

Example

employees

Eid	name	Bdate	salary	*Dnum*
1234	Sean Russell	1985-07-21	50000	20
4567	Jamie Heaslip	1982-05-04	47000	10
6542	Leo Cullen	1978-01-07	45000	10
1238	Brendan Macken	1990-12-03	25000	20
1555	Sean O'Brien	1986-04-09	50000	30

departments

Dnum	Dname	*Dmgr_id*
10	Training	4567
20	Design	1238
30	Implementation	1555

Example - Alternate

employees_departments

Eid	name	Bdate	salary	Dnum	Dname	Dmgr_id
1234	Sean Russell	1985-07-21	50000	20	Design	1238
4567	Jamie Heaslip	1982-05-04	47000	10	Training	4567
6542	Leo Cullen	1978-01-07	45000	10	Training	4567
1238	Brendan Macken	1990-12-03	25000	20	Design	1238
1555	Sean O'Brien	1986-04-09	50000	30	Implementation	1555

- This configuration of the table contains the same information as the previous
- But we are recording the same pieces of information in several places

- One of the biggest concerns with a database is integrity
- Anomalies in that database are where our data is not correct
- This can happen in a number of ways
  - When inserting data
  - When deleting data
  - When modifying data

Insertion Anomalies

- Every time we insert new data, we must make sure it is consistent with the data already in the table
- For example, when inserting a new employee into department 10, we must make sure that the department name and manager are correct

employees_departments

Eid	name	Bdate	salary	Dnum	Dname	Dmgr_id
1234	Sean Russell	1985-07-21	50000	10	Design	1238
4444	Paul Heaslip	1983-05-04	44000	10	Training	4567

- With this data it is impossible to know what the name of department 10 is and who is the manager

Deletion Anomalies

- Because we have represented the employees and departments together deleting data can cause problems
- If we delete employee 1555, we will lose all records of department number 30

employees_departments

Eid	name	Bdate	salary	Dnum	Dname	Dmgr_id
1234	Sean Russell	1985-07-21	50000	20	Design	1238
4567	Jamie Heaslip	1982-05-04	47000	10	Training	4567
6542	Leo Cullen	1978-01-07	45000	10	Training	4567
1238	Brendan Macken	1990-12-03	25000	20	Design	1238
1555	Sean O'Brien	1986-04-09	50000	30	Implementation	1555

Modification Anomalies

- Because we have represented the employees and departments together changing data can is more complicated
- If we want to rename one of the departments, we must rename it in every row where that department is use
- If we miss some rows, our data will be inconsistent

- In some database designs, we may create tables that have many attributes that do not apply to all rows
- This means that we end up with many rows with NULL values in those attributes
- This can cause problems with wasted space and may make some join operations more difficult to understand

employees

Eid	name	Bdate	salary	Dnum	typing_speed	eng_type
1234	Sean Russell	1985-07-21	50000	20	140	NULL
4567	Jamie Heaslip	1982-05-04	47000	10	NULL	Civil
6542	Leo Cullen	1978-01-07	45000	10	NULL	NULL
1238	Brendan Macken	1990-12-03	25000	20	NULL	NULL
1555	Sean O'Brien	1986-04-09	50000	30	NULL	Demolition

Table of Contents

- If $R = \{A_1, A_2, ..., A_n\}$ is a relation that describes all of the tables in our database
- Then a functional dependency, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R.
- The constraint is that, for any two tuples t_1 and t_2 in r that have $t_1[X] = t_2[X]$, they must also have $t_1[Y] = t_2[Y]$

Explanation

- A functional dependency means that the values for the attributes on the right depend on the attributes on the left
- So if we have the following functional dependency: $Eid \rightarrow name$
    - Then whenever we see particular id, we will know what name is on that row
    - The opposite is not true (two employees may have the same name)
- Functional dependencies always involve keys on the left (candidate and primary keys)

Example

employees

Eid	name	Bdate	salary	*Dnum*
1234	Sean Russell	1985-07-21	50000	20
4567	Jamie Heaslip	1982-05-04	47000	10
6542	Leo Cullen	1978-01-07	45000	10
1238	Brendan Macken	1990-12-03	25000	20
1555	Sean O'Brien	1986-04-09	50000	30

departments

Dnum	Dname	*Dmgr_id*
10	Training	4567
20	Design	1238
30	Implementation	1555

- $Eid \rightarrow \{name, Bdate, salary, Dnum\}$
- $Dnum \rightarrow \{Dname, Dmgr\_id\}$

Understanding the Data

- You cannot find the functional dependencies in a database by looking at the data
- You need to understand the meaning of the data
- You can prove that something is not a functional dependency by finding a single example that contradicts it

Keys

- Identifying functional dependencies comes down to keys
- We describe keys using several terms
  - superkey
  - candidate key
  - primary key
  - surrogate key
- Lets look at an example using the following relation and assumptions:
  - Table: `students(UCD_id, name, major , Bdate)`
  - Assumptions: UCD_id is unique and the combination of name and Bdate is also unique

Superkeys

- A **superkey** is a **set** of attributes that uniquely identify every row in a database
- From the example, we can identify the following superkeys:
  - $\{UCD\_id, name, major, Bdate\}$
  - $\{UCD\_id, name, major\}$
  - $\{UCD\_id, name, Bdate\}$
  - $\{UCD\_id, major, Bdate\}$
  - $\{UCD\_id, name\}$
  - $\{UCD\_id, major\}$
  - $\{UCD\_id, Bdate\}$
  - $\{UCD\_id\}$
  - $\{name, major, Bdate\}$
  - $\{name, Bdate\}$

Keys

- A **key**, is a superkey with one additional property, removing any attribute from the set will make it no longer a superkey
- This means if we remove any of the attributes, the set is no longer unique and cannot be used to identify each row
- The main difference between keys and super keys is that keys are minimal

Keys and Superkeys

- From out previous example:
  1. $\{UCD\_id, name, major, Bdate\}$ superkey
  2. $\{UCD\_id, name, major\}$ superkey
  3. $\{UCD\_id, name, Bdate\}$ superkey
  4. $\{UCD\_id, major, Bdate\}$ superkey
  5. $\{UCD\_id, name\}$ superkey
  6. $\{UCD\_id, major\}$ superkey
  7. $\{UCD\_id, Bdate\}$ superkey
  8. $\{UCD\_id\}$ **key**
  9. $\{name, major, Bdate\}$ superkey
  10. $\{name, Bdate\}$ **key**
- If we remove an attribute from 8 it is empty
- If we remove any attribute from 10, it can no longer identify the rows

Candidate and Primary Keys

- If a table in our database has more than on key, each key is called a **candidate key**
- One of the keys is chosen to be the **primary key**
  - How you choose does not matter
  - In the previous example, I would choose UCD_id, because it is only made up of a single attribute
  - This could make join operations easier
- Every table **must** have a primary key

Surrogate Key

- Every table **must** have a primary key
  - If you can't identify any candidate keys, we can use all of the attributes as the primary key
- Alternatively, we can make up a primary key by adding a new attribute
  - This is called a **surrogate key**
  - Usually, automatically generated integers are used
  - This is how you get your student numbers

Functional Dependencies and Keys

- There is always a functional dependency from each candidate key to all other information in the table
- `students(UCD_id, name, major , Bdate)`
  - $\{UCD\_id\} \rightarrow \{name, major, Bdate\}$
  - $\{name, Bdate\} \rightarrow \{UCD\_id, major\}$

Functional Dependency Types

- For the some normal forms, we must have an understanding of the different types of functional dependencies
- There are two types of functional dependency
  - Full functional dependency
  - Partial functional dependency
- Partial functional dependency is something we need to check for when our primary key contains multiple attributes
- A primary key with only a single attribute is automatically a full functional dependency

Full Functional Dependency

- A functional dependency $X \rightarrow Y$ is a **full** functional dependency if we **cannot** remove any attributes from X and have it stay as a functional dependency
- Consider the relation: $project\_hours(\underline{E\_id}, \underline{Pnum}, hours)$
- We have the functional dependency $\{E\_id, Pnum\} \rightarrow \{hours\}$
- This is a full functional dependency because neither $\{E\_id\} \rightarrow \{hours\}$ or $\{Pnum\} \rightarrow \{hours\}$ hold as functional dependencies

Partial Functional Dependency

- A functional dependency $X \rightarrow Y$ is a **partial** functional dependency if we **can** remove an attribute from X and have it stay as a functional dependency
- Consider the relation:
  $emp\_proj(\underline{E\_id}, \underline{Pnum}, Ename, Pname, Plocation)$
- We have the functional dependencies $\{E\_id\} \rightarrow \{Ename\}$, $\{E\_id, Pnum\} \rightarrow \{hours\}$ and $\{Pnum\} \rightarrow \{Pname, Ploaction\}$
- Lets look at the dependencies on the key we have $\{E\_id, Pnum\} \rightarrow$ $\{hours, Ename, Pname, Plocation\}$

Partial Functional Dependency

- So we are checking if $\{E\_id, Pnum\} \rightarrow \{hours, Ename, Pname, Plocation\}$ is a partial functional dependency
  - $\{E\_id, Pnum\} \rightarrow \{hours\}$ - Cannot remove part of key
  - $\{E\_id, Pnum\} \rightarrow \{Ename\}$ - Can remove Pnum
  - $\{E\_id, Pnum\} \rightarrow \{Pname\}$ - Can remove E_id
  - $\{E\_id, Pnum\} \rightarrow \{Plocation\}$ - Can remove E_id
- This is a **partial** functional dependency because $\{E\_id\} \rightarrow \{Ename\}$ and $\{Pnum\} \rightarrow \{Pname, Plocation\}$ both hold as functional dependencies

Table of Contents

- The data in a database can be considered to be in one of a number of **normal forms**.
- Basically, the normal form of the data indicates how much redundancy is in the data.
- The normal forms have a strict ordering
  - 1st Normal Form
  - 2nd Normal Form
  - 3rd Normal Form
  - Boyce-Codd Normal Form (BCNF)
  - 4th Normal Form
  - 5th Normal Form

- First Normal Form (1NF) deals with the shape of the record type.
- A relation is in 1NF if, and only if, it contains no repeating attributes or groups of attributes
- $\{UCD\_id, name, Bdate, (subject, grade)\}$
    - For this example, we have a single row for each student, but can have multiple pairs of subject and grade
    - This is not usually a feature of modern database systems

students

UCD_id	name	Bdate	subject	grade
12345	Smith.J	1985-07-21	Java	B
			Soft Eng	C
			Databases	A
23456	White.A	1990-12-03	Java	D
			Soft Eng	B
34567	Moore.T	1986-04-09	Databases	A
			Soft Eng	B
			Networks	C
45678	Smith.J	1998-11-02	Soft Eng	C

- This table with the repeating group is not in 1NF.
- To remove the repeating group, one of two things can be done:
  - "Flatten" the relation (fill in the empty attribute spaces) and extend the key, or
  - "Decompose" the relation (divide into multiple relations)

Flatten

students

<u>UCD_id</u>	name	Bdate	<u>subject</u>	grade
12345	Smith.J	1985-07-21	Java	B
12345	Smith.J	1985-07-21	Soft Eng	C
12345	Smith.J	1985-07-21	Databases	A
23456	White.A	1990-12-03	Java	D
23456	White.A	1990-12-03	Soft Eng	B
34567	Moore.T	1986-04-09	Databases	A
34567	Moore.T	1986-04-09	Soft Eng	B
34567	Moore.T	1986-04-09	Networks	C
45678	Smith.J	1998-11-02	Soft Eng	C

- The empty spaces are filled in with the data from the row
  - But this would mean we have multiple rows with the same primary key
- To correct this, the key is changed to $\{UCD\_id, subject\}$
- This is now in **1NF**

Problems with Flattened Tables

- Redundant Data
  - We are wasting storage space by storing the name and date of birth of each student many times
- Anomalies

Insertion : Because the key is now UCD_id and subject, we cannot insert a new student until they have completed a course

Update : To change the name of a student, we must update it in every row for that student in the table

Deletion : If we delete all rows containing data about a subject, we will also delete any student who has only completed that subject

Decompose

- The alternative approach is to split the table into two relations: one for the repeating groups and one for the non-repeating groups.
- The primary key for the original relation is included in both of the new relations.
- We can return to the original table by using a JOIN operation on these relations

Decomposed

students

UCD_id	name	Bdate
12345	Smith.J	1985-07-21
23456	White.A	1990-12-03
34567	Moore.T	1986-04-09
45678	Smith.J	1998-11-02

grades

UCD_id	subject	grade
12345	Java	B
12345	Soft Eng	C
12345	Databases	A
23456	Java	D
23456	Soft Eng	B
34567	Databases	A
34567	Soft Eng	B
34567	Networks	C
45678	Soft Eng	C

Results with Decomposed Tables

- There is no redundancy
- Anomalies

Insertion : We can insert a new student before they have completed a course

Update : We only need to update a single row to change the name of a student

Deletion : We can delete all of the rows for a subjects grade, and we will not delete any students

- A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is **fully** functionally dependent on the **whole** key
- The relation must be in 1NF and all non-key attributes must depend on the whole key, not just part of it.
  - In other words: there must be no partial key dependencies.
- The problem arises when there is a compound key, e.g. in the Grades relation: UCD_id, subject
- In this case it is possible for non-key attributes to depend on only part of the key (i.e. on only one of the key attributes).

2NF Example 1

- Consider the flattened student relation: $students(\underline{UCD\_id}, name, Bdate, \underline{subject}, grade)$
- There are no repeating groups: already in 1NF.
- However, there is a compound primary key, so we must check that the non-key attributes depend on the whole key.
- There are three non-key attributes: name, Bdate and grade

2NF Example 1

- So we are checking if $\{UCD\_id, subject\} \rightarrow \{name, Bdate, grade\}$ is a partial functional dependency
  - $\{UCD\_id, subject\} \rightarrow \{name\}$ - Can remove subject
  - $\{UCD\_id, subject\} \rightarrow \{Bdate\}$ - Can remove subject
  - $\{UCD\_id, subject\} \rightarrow \{grade\}$ - Cannot remove part of key
- This relation is not in 2NF. It appears to be two tables squashed into one.
- Solution: decompose the relation

Heath's Theorem

- Heath's Theorem allows us to perform non-loss decomposition.
- If R is a relation made up of sets of attributes $\{A, B, C\}$.
- If $A \rightarrow B$, then R can be non-loss decomposed into:
    - $\{A, B\}$
    - $\{A, C\}$
- R can be created again using $\{A, B\} JOIN \{A, C\}$

Heath's Theorem Steps

1. Create a new relation that contains all of the attributes that are solely dependent on UCD_id (UCD_id is the primary key of the new relation).
2. Create a new relation that contains all the attributes that are solely dependent on subject (subject is the primary key of the new relation).
   - In this example, there are no attributes that depend only on subject.
3. Create a new relation with all the attributes that depend on both UCD_id and subject.
   - The primary key is UCD_id, Subject

Result

$students(\underline{UCD\_id}, name, Bdate)$
$grades(\underline{UCD\_id}, \underline{subject}, grade)$

- All attributes in each relation are fully functionally dependent on the primary key.
  - $\{UCD\_id\} \rightarrow \{name, Bdate\}$
  - $\{UCD\_id, subject\} \rightarrow \{grade\}$
- Both relations are now in 2NF

- 3NF is an even stricter normal form that removes almost all redundant data.
- A relation is in 3NF if, and only if, it is in 2NF and there are no transitive functional dependencies.
- Transitive functional dependencies are when one non-key attribute is functionally dependent on another non-key attribute.
- By definition, a transitive functional dependency can only happen if there is more than one non-key attribute

3NF Example

- Consider the following table *projects*(<u>*Pnum*</u>, *manager*, *address*)

projects

<u>Pnum</u>	manager	address
P1	Black. B	32 High St
P2	Smith.J	11 New St
P3	Black. B	32 High St
P4	Black. B	32 High St

- Projects has more than one non-key field, so we must check for transitive dependency

3NF Example

- In this example, we are told that address depends on the value of manager.
  - If we know a project's manager, we can find the address
- This gives us a functional dependency of $\{manager\} \rightarrow \{address\}$
- In this case, address is transitively dependent on manager.
- The primary key is Pnum, but the functional dependency makes no reference to this key

3NF Example

- Data redundancy can come from this:
  - We duplicate the address if a manager is in charge of more than one project.
  - Causes problems if we have to change the address, because it must be changed in several places.
- Solution: decompose:
  - Create two relations: one with the transitive dependency in it, and another for all of the remaining attributes.
  - Split projects into projects and managers
  - In the projects relation, we keep the same primary key: Pnum
  - In the managers relation we use the left side of the functional dependency as the primary key: manager

projects

Pnum	*manager*
P1	Black. B
P2	Smith.J
P3	Black. B
P4	Black. B

managers

manager	address
Black. B	32 High St
Smith.J	11 New St

- Now we store the address of each manager only once
- If we need to know a manager's address, we can look it up in the managers table
- The manager attribute in the Projects relation it is now a foreign key.
- These relations are now in 3NF

- Boyce-Codd Normal Form (BCNF) is named after Raymond Boyce and Edgar Codd who developed it in 1974 to fix some anomalies not addressed in 3NF
    - It is sometimes called 3.5NF
- A relation in 3NF is usually also in BCNF
- Only if a relation in 3NF has **overlapping candidate keys** can it not be in BCNF

Overlapping Candidate Keys

- Overlapping candidate keys means that we have composite candidate keys with at least one attribute in common
  - E.G If we have relation containing prices for products: $products(Pid, start\_date, end\_date, price)$
  - Then we would have candidate keys of $\{Pid, start\_date\}$ and $\{Pid, end\_date\}$
- Here there is an overlap, because Pid is in both candidate keys

BCNF

- BCNF is based on the concept of a **determinant**.
  - A determinant is any attribute (or group of attributes) that some other attribute is fully functionally dependent on
  - I.e. the left side of a functional dependency
- A relation is in BCNF if, and only if, every determinant is a candidate key.

BCNF: Theory

- Consider the following relation and dependencies:
- R(a, b, c, d)
  - $\{a, c\} \rightarrow \{b, d\}$
  - $\{a, d\} \rightarrow \{b\}$
- To be in BCNF, all determinants must be candidate keys. Let's look at each in turn
- $\{a, c\} \rightarrow \{b, d\}$
  - a and c together can determine both b and d
  - Therefore $\{a, c\}$ is a candidate key for this relation
- $\{a, d\} \rightarrow \{b\}$
  - $\{a, d\}$ does not determine c, so it can't be a candidate key.
  - Therefore R is not in BCNF.

BCNF: Hospital Example

PatNum	PatName	AppSlot	Time	Doctor
1	Eamonn	0	09:00	Octopus
2	Eoin	0	09:00	Evil
3	Arnold	1	10:00	Octopus
4	Stephen	0	13:00	Evil
5	Patricia	1	14:00	Octopus

- Extra information:
  - Every patient has a unique patient number.
  - Patients with names beginning with a letter before ‘P’ get morning appointments.
  - The Appointment slots start at 0 for the first appointment of the morning or afternoon, 1 for the second and so on.

BCNF: Hospital Example

- $Appointments(PatNum, PatName, AppSlot, Time, Doctor)$
- We are given some functional dependencies (mostly based on the extra information):
  - $\{PatNum\} \rightarrow \{PatName\}$
  - $\{PatNum, AppSlot\} \rightarrow \{Time, Doctor\}$
  - $\{Time\} \rightarrow \{AppSlot\}$
- We have two options for selecting a primary key:
  - Appointments(<u>PatNum</u>, PatName, <u>AppSlot</u>, Time, Doctor): example A
  - Appointments(<u>PatNum</u>, PatName, AppSlot, <u>Time</u>, Doctor): example B

BCNF: Example A

- *Appointments*(*<u>PatNum</u>*, *PatName*, *<u>AppSlot</u>*, *Time*, *Doctor*)
- No repeating groups, so in 1NF.
- 2NF – eliminate partial key dependencies:
  - *Appointments*(*<u>PatNum</u>*, *<u>AppSlot</u>*, *time*, *Doctor*)
  - *Patients*(*<u>PatNum</u>*, *PatName*)
- 3NF – no transient dependencies so it's already in 3NF
- Now try BCNF

BCNF: Example A

- BCNF: Every determinant must be a candidate key
  - *Appointments*(<u>*PatNum*</u>, <u>*AppSlot*</u>, *Time*, *Doctor*)
  - *Patients*(<u>*PatNum*</u>, *PatName*)
- For each functional dependency, we will look at each table and ask the following questions:
  1. Are the attributes present in the table?
  2. Is the determinant the key of the table?

BCNF: Example A

- BCNF: Every determinant must be a candidate key
  - $Appointments(\underline{PatNum}, \underline{AppSlot}, Time, Doctor)$
  - $Patients(\underline{PatNum}, PatName)$
- We have 3 functional dependencies:
  1. $\{PatNum\} \rightarrow \{PatName\}$
  2. $\{PatNum, AppSlot\} \rightarrow \{Time, Doctor\}$
  3. $\{Time\} \rightarrow \{AppSlot\}$

1. $\{PatNum\} \rightarrow \{PatName\}$

- $Patients(\underline{PatNum}, PatName)$
  1. All attributes are present, so this is relevant
  2. $\{PatNum\}$ is the key of the table (and also a candidate key), so this is OK
- $Appointments(\underline{PatNum}, \underline{AppSlot}, Time, Doctor)$
  1. PatNum is present in Appointments, but not PatName, so this is not relevant

2. $\{PatNum, AppSlot\} \rightarrow \{Time, Doctor\}$

- $Patients(\underline{PatNum}, PatName)$
  1. Not all attributes are present, so this is not relevant
- $Appointments(\underline{PatNum}, \underline{AppSlot}, Time, Doctor)$
  1. All attributes are present, so this is relevant
  2. $\{PatNum, AppSlot\}$ is the key of the table (and also a candidate key), so this is OK

3. $\{Time\} \rightarrow \{AppSlot\}$

- $Patients(\underline{PatNum}, PatName)$
  1. No attributes are present, so this is not relevant
- $Appointments(\underline{PatNum}, \underline{AppSlot}, Time, Doctor)$
  1. All attributes are present, so this is relevant
  2. $\{Time\}$ is the not a candidate key of the table,
- This is not in BCNF

Rewrite to BCNF

- Original 3NF version:
  - *Appointments*(*<u>PatNum</u>*, *<u>AppSlot</u>*, *Time*, *Doctor*)
  - *Patients*(*<u>PatNum</u>*, *PatName*)
- Rewriten to BCNF
  - *Appointments*(*<u>PatNum</u>*, *<u>Time</u>*, *Doctor*)
  - *Patients*(*<u>PatNum</u>*, *PatName*)
  - *Slots*(*<u>Time</u>*, *AppSlot*)
- "Time" is enough to work out the appointment slot of a patient
- Now BCNF is satisfied, and the final relations shown are in BCNF

BCNF: Example B

- *Appointments*(<u>*PatNum*</u>, *PatName*, *AppSlot*, <u>*Time*</u>, *Doctor*)
- No repeating groups, so in 1NF.
- 2NF – eliminate partial key dependencies:
    - *Appointments*(<u>*PatNum*</u>, <u>*Time*</u>, *Doctor*)
    - *Patients*(<u>*PatNum*</u>, *PatName*)
    - *Slots*(<u>*Time*</u>, *AppSlot*)
- 3NF – no transient dependencies so it's already in 3NF
- Now try BCNF

BCNF: Example B

- BCNF: Every determinant must be a candidate key
  - $Appointments(\underline{PatNum}, \underline{Time}, Doctor)$
  - $Patients(\underline{PatNum}, PatName)$
  - $Slots(\underline{Time}, AppSlot)$
- We have 3 functional dependencies:
  1. $\{PatNum\} \rightarrow \{PatName\}$
  2. $\{PatNo, AppSlot\} \rightarrow \{Time, Doctor\}$
  3. $\{Time\} \rightarrow \{AppSlot\}$

1. $\{PatNum\} \rightarrow \{PatName\}$

- $Appointments(\underline{PatNum}, \underline{Time}, Doctor)$
  1. PatNum is present in Appointments, but not PatName, so this is not relevant
- $Patients(\underline{PatNum}, PatName)$
  1. All attributes are present, so this is relevant
  2. $\{PatNum\}$ is the key of the table (and also a candidate key), so this is OK
- $Slots(\underline{Time}, AppSlot)$
  1. No attributes are present, so this is not relevant

2. $\{PatNum, AppSlot\} \rightarrow \{Time, Doctor\}$

- $Appointments(\underline{PatNum}, \underline{Time}, Doctor)$
  1. PatNum, Time and Doctor are present, but not AppSlot, so this is not relevant
- $Patients(\underline{PatNum}, PatName)$
  1. PatNum is present in, but none of the others, so this is not relevant
- $Slots(\underline{Time}, AppSlot)$
  1. Time and AppSlot are present, but not the others, so this is not relevant

3. $\{Time\} \rightarrow \{AppSlot\}$

- $Appointments(\underline{PatNum}, \underline{Time}, Doctor)$
  1. Time is present, but not AppSlot, so this is not relevant
- $Patients(\underline{PatNum}, PatName)$
  1. No attributes are present, so this is not relevant
- $Slots(\underline{Time}, AppSlot)$
  1. Time and AppSlot are present, so this is relevant
  2. $\{Time\}$ is the key of the table (and also a candidate key), so this is OK
- Relations are in BCNF

- It can take a bit of time and effort to get a database design into BCNF
- So what is the benefit of having done so?
- We will look at an example that shows some of the problems that BCNF overcomes

students

Snum	major	supervisor
123	Physics	Einstein
123	Music	Mozart
456	Biology	Darwin
789	Physics	Bohr
999	Physics	Einstein

- No repeating groups, so it's in 1NF
- No partial key dependencies, so it's in 2NF
- There's only one non-key attribute (Supervisor) so it must be in 3NF

students

Snum	major	supervisor
123	Physics	Einstein
123	Music	Mozart
456	Biology	Darwin
789	Physics	Bohr
999	Physics	Einstein

- We have the following functional dependencies:
  - $\{Snum, major\} \rightarrow \{supervisor\}$
  - $\{supervisor\} \rightarrow \{Major\}$

students

<u>Snum</u>	<u>major</u>	supervisor
123	Physics	Einstein
123	Music	Mozart
456	Biology	Darwin
789	Physics	Bohr
999	Physics	Einstein

- If the record for student 456 is deleted we lose not only information about that student, but also the fact that Darwin advises in Biology.
- We cannot record the fact that Watson can advise on Computing until we have a student doing a project on Computing that has Watson as an advisor.

- In BCNF we have two tables, and these problems are eliminated:

supervisors

Snum	supervisor
123	Einstein
123	Mozart
456	Darwin
789	Bohr
999	Einstein

majors

supervisor	major
Einstein	Physics
Mozart	Music
Darwin	Biology
Bohr	Physics

Summary

- A relation is in 1NF if, and only if, it contains no repeating groups.
- A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is fully functionally dependent on the whole key.
- A relation is in 3NF if, and only if, it is in 2NF and has no transitive functional dependencies.
- A relation is in BCNF if, and only if, it is in 3NF and every determinant is a candidate key.