Contents

Structured Query Language – SQL

- "Programming" language for DBMS
  - Data definition language: creation of data models (table design)
  - Data manipulation language: insertion, modification, deletion of data
  - Data query language: queries
- It is run on a DBMS
- The most widely used standard
  - Created in 1974 (D. D. Chamberlin & R. F. Boyce, IBM)
  - ANSI in 1986, ISO in 1987
  - Core (all DBMS) + packages (optional modules)
- Versions
  - SQL1 – SQL 86
  - SQL2 – SQL 92
  - SQL3 – SQL 1999, not fully supported by industry (recursion, programming, objects…)
- Limitations
  - It is not purely relational (e.g. views are *multisets* of tuples)
  - Important divergences between implementations (not directly portable in general, incompleteness, extensions) –one ends up learning SQL variants

Donald D. Chamberlin

Elements of an SQL database

- Database = set of tables
- Table (relation, entity, schema…) =
  - Fixed field structure (schema)
  - Set of records with field values
- Field (attribute, property, “column”), has a data type
- Record (tuple, “row”)
- Primary key
- Foreign keys

Lexical structure of the language

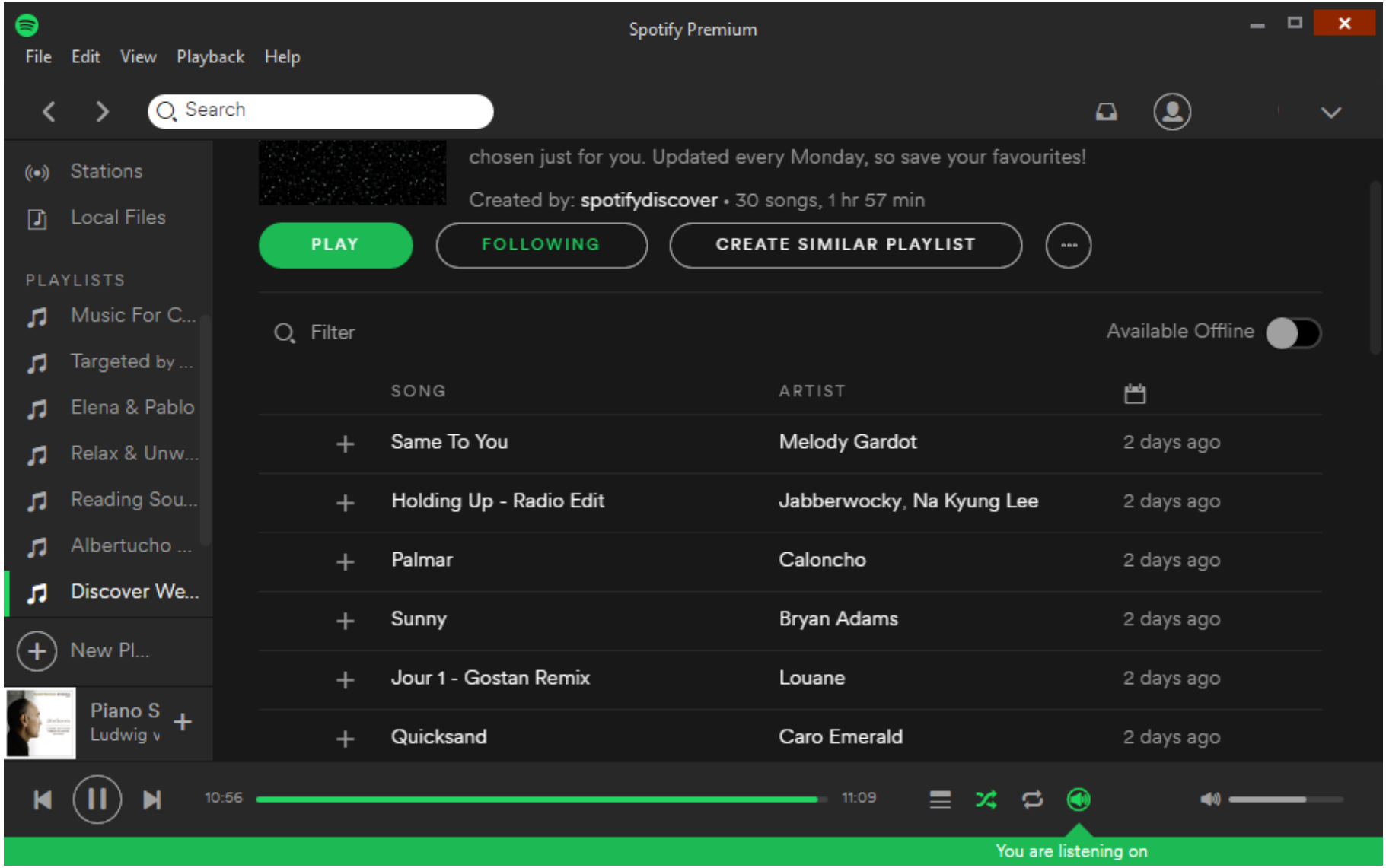
SQL operations

- DDL – Table creation, design, modification
- DML – Record insertion, modification, removal
- DQL – Query

Lexical structure of SQL

- Case-insensitive, insignificant whitespace
- Sentences, expressions, values, data types
- References
  - Elmasri cap. 8
  - PostgreSQL SQL ref: http://www.postgresql.org/docs/9.4

Example: DB for music application



Example: informal description

Music streaming with social network

- Data types: users, songs, albums, artists…
- Structures:
  - Users have a name, username, email…
  - Songs have a title, style, duration, date…
  - Artists have name, nationality…
- Relations:
  - Songs have authors, albums have songs, users have friends, favorite artists, users play songs…
- Functionalities:
  - Search for a song, play it, browse song data…
  - View / add friends…

Example: table view

Artist

id	name	nationality
0	The Beatles	UK
1	The Rolling Stones	UK
2	David Bowie	UK

Song

id	title	genre	duration	release_date	author
0	Norwegian Wood	Pop	125	1965-03-12	0
1	Here, there and everywhere	Pop	145	1966-08-05	0
2	Jumping jack flash	Pop	225	1968-04-20	1

User

username	name	email
amy	Amelia	amy@gmail.com
jim	James	jim@gmail.com
nick	Nicholas	nick@gmail.com
cate	Catherine	cate@gmail.com

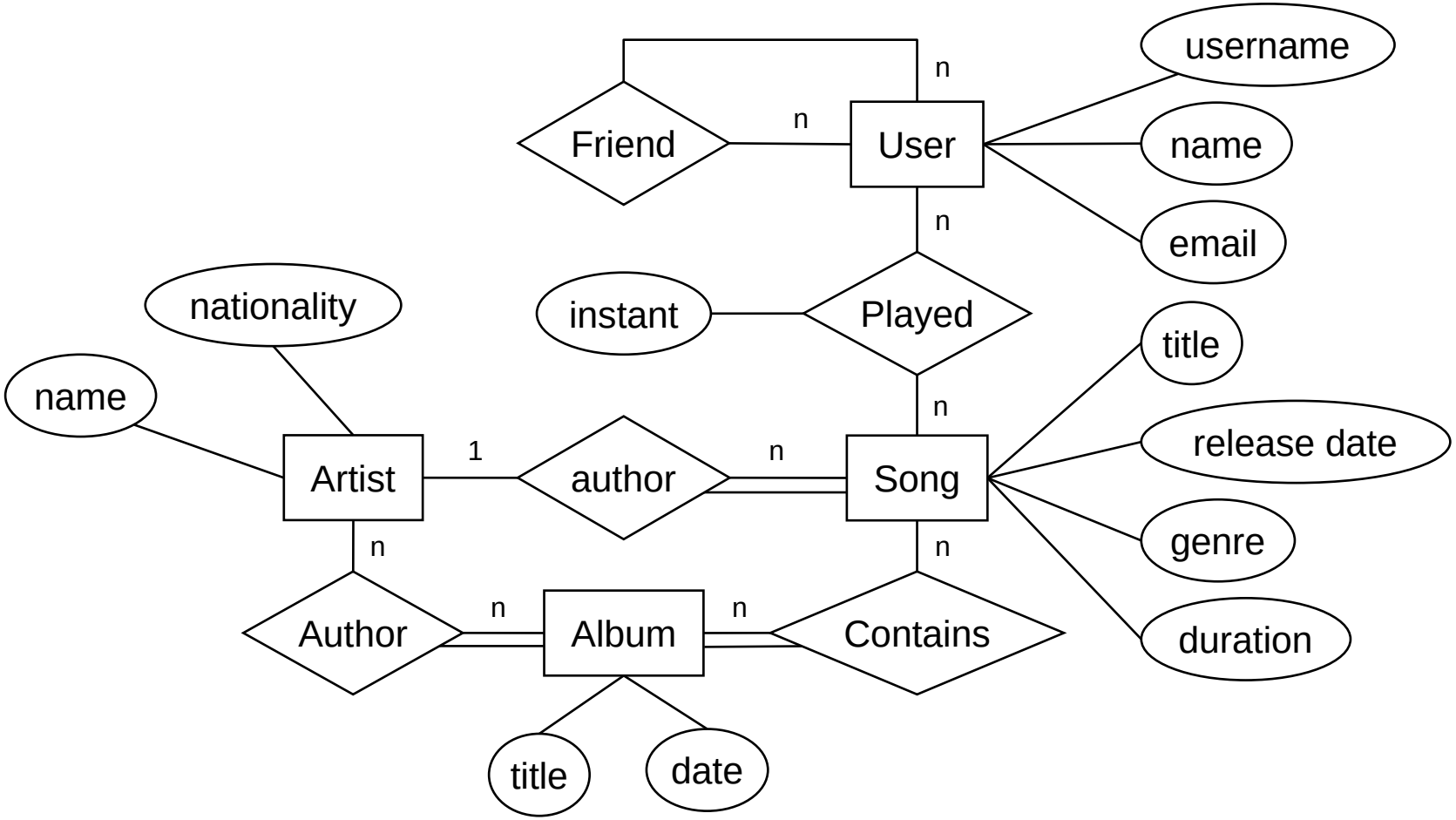
Friend

user1	user2
jim	amy
cate	jim
nick	cate

Played

user	played_song	instant
cate	1	2019-09-09 16:57:54
jim	2	2019-09-12 21:15:30

Example – ER diagram



Motivation

Database for music artist, songs, users, play records, social network

Example queries

- *Songs of the 60's* (*show title and genre*)
- *Set of all nationalities of artists in the DB*
- *Songs of artists from the UK*
- *Titles of songs that a user ever listened to*
- *All the friends of a given user*
- *Users with the same name*
- *All users at distance 2 of a given user in the social network*
- *Common friends to two given users*
- *How many times a song has been played*
- *Artists sorted from most to least played*
- *Users with more than two friends*
- *Users sorted by their number of friends*
- *User with most friends*

Data definition

CREATE TABLE *name* (

field1 type1 [*constraints1*],

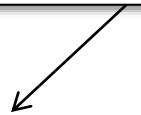
field2 type2 [*constraints2*],

…,

[*constraints*]

);

ALTER commands are useful for slight design modifications of non-empty tables



ALTER TABLE *name* ADD COLUMN *field type* [*constraints*];

ALTER TABLE *name* ADD *constraint*;

ALTER TABLE *name* DROP COLUMN *field*;

DROP TABLE *name*;

DROP CONSTRAINT *constraint-name*;

Example

Artist

id	name	nationality

```
CREATE TABLE Artist (
        id              int,
        name            text,
        nationality     text
);
```

Example

Artist

id	name	nationality

```
CREATE TABLE Artist (
        id              int     PRIMARY KEY,
        name            text    NOT NULL,
        nationality     text
);
```

Example

Song

id	title	genre	duration	release_date	author

```
CREATE TABLE Song (
        id              int         PRIMARY KEY,
        title           text        NOT NULL,
        genre           text,
        duration        int,
        release_date    date,
        author          int         NOT NULL REFERENCES Artist (id)
);
```

```
CREATE TABLE User (
        username        varchar(30)   PRIMARY KEY,
        name            text          NOT NULL,
        email           text          NOT NULL UNIQUE
);

CREATE TABLE Friend (
        user1           varchar(30)   REFERENCES User (username),
        user2           varchar(30)   REFERENCES User (username),
        PRIMARY KEY (user1, user2)
);

CREATE TABLE Played (
        user                varchar(30),
        played_song         int REFERENCES Song (id),
        instant             timestamp,
        PRIMARY KEY (user, played_song, instant)
);
```

Example: change tables after created

```
ALTER TABLE Played DROP COLUMN instant;

ALTER TABLE Played ADD instant timestamp; /* NULL's */


ALTER TABLE Played ADD    FOREIGN KEY (user)
                          REFERENCES User (username);

ALTER TABLE User ADD PRIMARY KEY (username);
```

Summary

- **CREATE** TABLE *name* (…);
- **ALTER** TABLE *name*
  - ADD
    - COLUMN *field type* [*constraints*];
    - *constraint*;
  - DROP COLUMN *field*;
- **DROP**
  - TABLE *name*;
  - CONSTRAINT *constraint-name*;

- Did you say “constraints”?
- Yes, I did.
- What is that?
- Mainly primary and foreign keys.
  We’ll see that in a few moments,
  let us se how to enter data first

Resulting DB state: a set of empty tables

Artist

id	name	nationality

Song

id	title	genre	duration	release_date	author

User

username	name	email

Friend

user1	user2

Played

user	played_song	instant

Let’s fill this in!

Data manipulation

INSERT INTO *table* [(*field1*, *field2*, …)]
VALUES
(*value11*, *value12*, …),
(*value21*, *value22*, …),
…
;

A query can also be nested here: the returned tuples are inserted

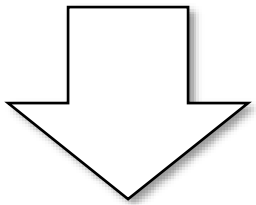
UPDATE *table* SET *field1* = *value1*, *field2* = *value2*, …
/* PostgreSQL: [FROM …] to form conditions with other tables */
[WHERE …];

DELETE FROM *table*
/* PostgreSQL: [USING *table*1, *table*2, …] for conditions with other tables */
[WHERE …];

TRUNCATE TABLE *table*;

Example: insert rows

```
INSERT INTO Artist VALUES (0, 'The Beatles', 'UK');
```

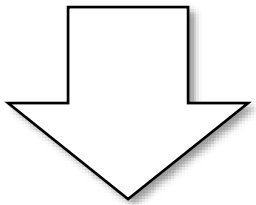


Artist

id	name	nationality
0	The Beatles	UK

Example: insert rows

```
INSERT INTO Artist VALUES (0, 'The Beatles', 'UK');
INSERT INTO Artist VALUES (1, 'The Rolling Stones', 'UK');
```

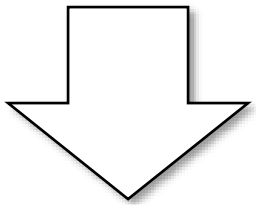


Artist

id	name	nationality
0	The Beatles	UK
1	The Rolling Stones	UK

Example: insert **incomplete** rows

```
INSERT INTO Artist VALUES (0, 'The Beatles', 'UK');
INSERT INTO Artist VALUES (1, 'The Rolling Stones', 'UK');
INSERT INTO Artist (id, name) VALUES (2, 'David Bowie');
```



Artist

id	name	nationality
0	The Beatles	UK
1	The Rolling Stones	UK
2	David Bowie	**NULL**

Example: insert in the other tables

```
INSERT INTO Song VALUES
    (0, 'Norwegian wood', 'Pop', 125, '1965-03-12', 0),
    (1, 'Here, there and everywhere', 'Pop', 145, '1966-08-05', 0),
    (2, 'Jumping jack flash', 'Pop', 225, '1968-04-20', 1);

INSERT INTO User VALUES
    ('amy', 'Amelia', 'amy@gmail.com'),
    ('jim', 'James', 'jim@gmail.com'),
    ('nick', 'Nicholas', 'nick@gmail.com'),
    ('cate', ‘Catherine', 'cate@gmail.com’);

INSERT INTO Friend VALUES
    ('jim', 'amy'),
    ('cate', 'jim'),
    ('nick', 'cate');

INSERT INTO Played VALUES
    ('cate', 2, '2019-09-09 16:57:54'),
    ('jim', 3, '2019-09-12 21:15:30');
```

Resulting DB state

Artist

id	name	nationality
0	The Beatles	UK
1	The Rolling Stones	UK
2	David Bowie	NULL

Song

id	title	genre	duration	release_date	author
0	Norwegian Wood	Pop	125	1965-03-12	0
1	Here, there and everywhere	Pop	145	1966-08-05	0
2	Jumping jack flash	Pop	225	1968-04-20	1

User

username	name	email
amy	Amelia	amy@gmail.com
jim	James	jim@gmail.com
nick	Nicholas	nick@gmail.com
cate	Catherine	cate@gmail.com

Friend

user1	user2
jim	amy
cate	jim
nick	cate

Played

user	played_song	instant
cate	1	2019-09-09 16:57:54
jim	2	2019-09-12 21:15:30

Example: change cell values

UPDATE Artist **SET** nationality = 'UK’
WHERE name = 'David Bowie';

UPDATE Album **SET** price = price * 1.2;
/* Fictional field, for illustrative purpose */

DELETE FROM Played **WHERE** instant < '2000-01-01 00:00:00';

- What literal values can I use?
- What data types to I have?
- How do I write conditions?
- How do I write expressions?

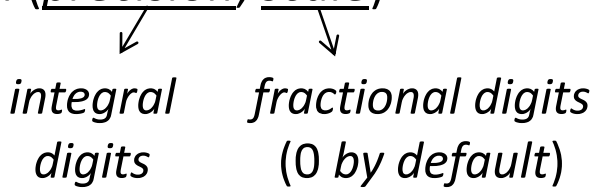
- What literal values can I use?
- What data types to I have?
- How do I write conditions?
- How do I write expressions?

These are common questions
for any “programming language”

Types and expressions

SQL types

- character(*n*) ≡ char(*n*), varchar(*n*), text
- integer ≡ int, smallint
- float, real, double precision
- numeric (*precision*, *scale*) ≡ decimal (*precision*, *scale*)
- date, time, timestamp



Literal values

- Strings in between '...'
- Numeric values similar e.g. to C
- date 'YYYY-MM-DD', time 'HH:MM:SS'

Expressions

- Can be used in WHERE, SELECT, SET, DEFAULT, CHECK…

Operators

- + – * / % ^
- AND OR NOT
- = < > <= >= LIKE ISNULL
- string operations: concatenation, like, regular expressions ('%' '_')

Comments

- – –
- /* … */

Now, about constraints…

Constraints

In a field

NOT NULL

UNIQUE

PRIMARY KEY

If omitted, will take the primary key

REFERENCES *table* (*key*) [(ON DELETE | ON UPDATE) (NO ACTION | RESTRICT | CASCADE | SET NULL | SET DEFAULT)]

DEFAULT *value*

With name

CONSTRAINT *name* *constraint*

In a table

PRIMARY KEY (*field1*, *field2*, …)

FOREIGN KEY (*field1*, *field2*, …) REFERENCES *table* (*key1*, *key2*, …)

UNIQUE (*field1*, *field2*, …)

CHECK (*expression*)

Primary keys

- They designate a unique identifier for the rows of a table
- There can be only one primary key per table, though it can comprise several fields
- It is very advisable that all tables have their primary key
- Technically equivalent to UNIQUE plus NOT NULL
- But they play a different role in indexing (we shall see this later on)
- Design choice: selection of primary key among several options
  - "Natural" primary key (email, web domain, SSN, ISBN, license plate, etc.)
  - "Artificial" primary key: usually an integer ID, typically self-incremented (serial)

Foreign keys

- Conceptually comparable to pointers
- They reference unique fields of another table
- The reference field is commonly a primary key
  - At least it has to be ‘unique’
- Technically they are not indispensable
- But they are greatly helpful in enforcing consistency in references!
  1. They throw an error when an attempt is made at inserting or setting a FK field to a value that does not exist in the referenced table
  2. And they automate what should happen when a referenced row is deleted
- In general it is preferable (more efficient) that FKs have integer type

Keys: in summary…

- The **primary key** of a table acts as a row **identifier**
  - Plays a similar role to the memory address of data in C
  - Only one per table (albeit not mandatory, always designate one)
  - Can be a natural field or (better) an artificial dedicated field (an integer)
  - Can be composed of several fields
- **Foreign keys** play the part of **pointers** between rows of different (or the same) tables
  - They typically point to a primary key
  - But they can also just point to a unique field
  - Foreign keys can also be composite
- What is the **advantage** of declaring a **primary key**?
  - Forbid the repetition of values in different rows (by raising an error)
  - Difference to unique not null: a matter of low-level implementation (indexes)
- What is the **advantage** of declaring **foreign keys**?
  - Enforce that its value appears in the pointed table (raising an error otherwise)
  - React automatically to changes in the pointed primary key

Example: primary and foreign keys

PK

FK

Follows

user1	user2
6	24
73	6
81	73

User

id	name	email
24	Amelia	amy@gmail.com
6	James	jim@gmail.com
81	Nicholas	nick@gmail.com
73	Catherine	cate@gmail.com

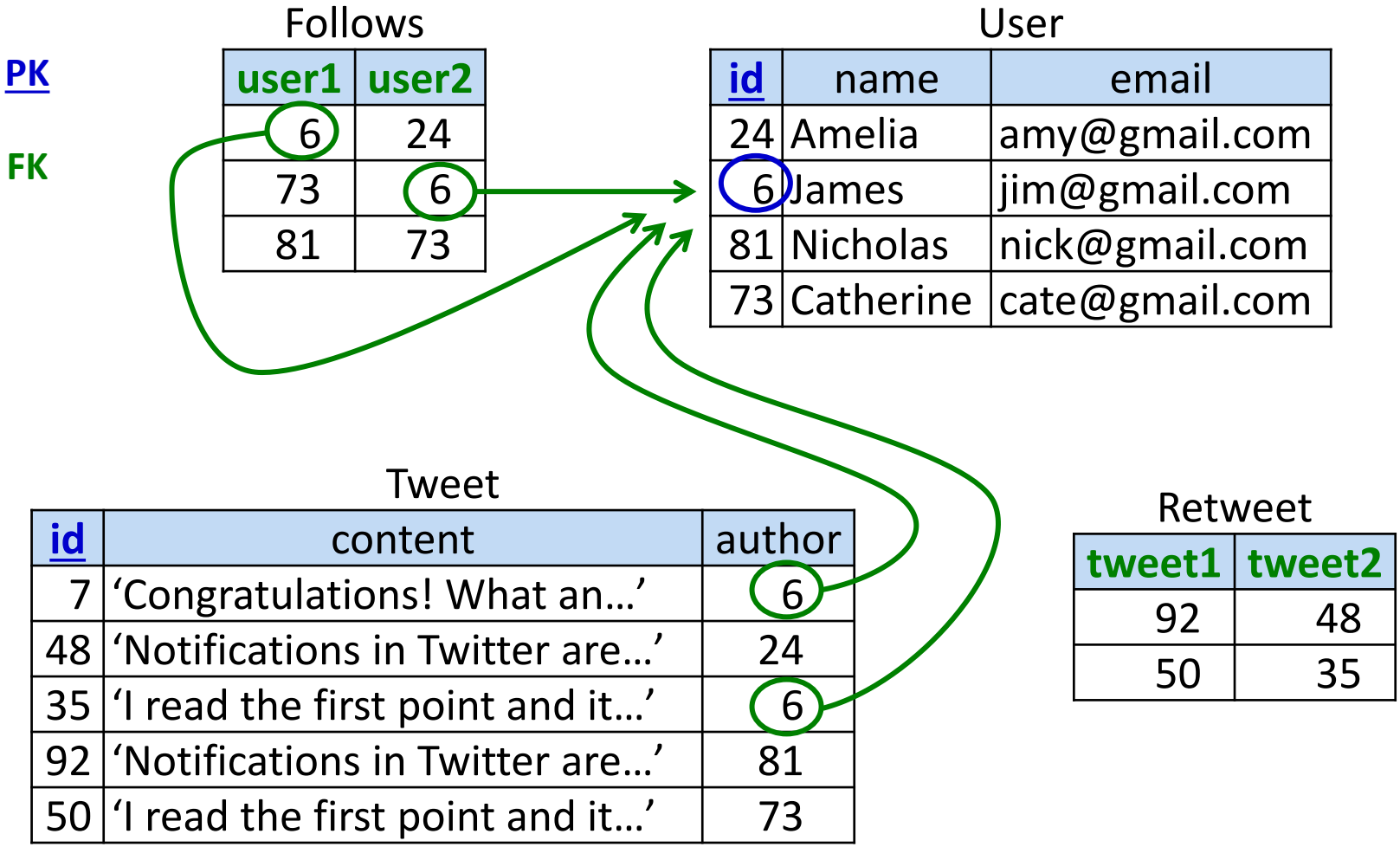
Tweet

id	content	author
7	‘Congratulations! What an…’	6
48	‘Notifications in Twitter are…’	24
35	‘I read the first point and it…’	6
92	‘Notifications in Twitter are…’	81
50	‘I read the first point and it…’	73

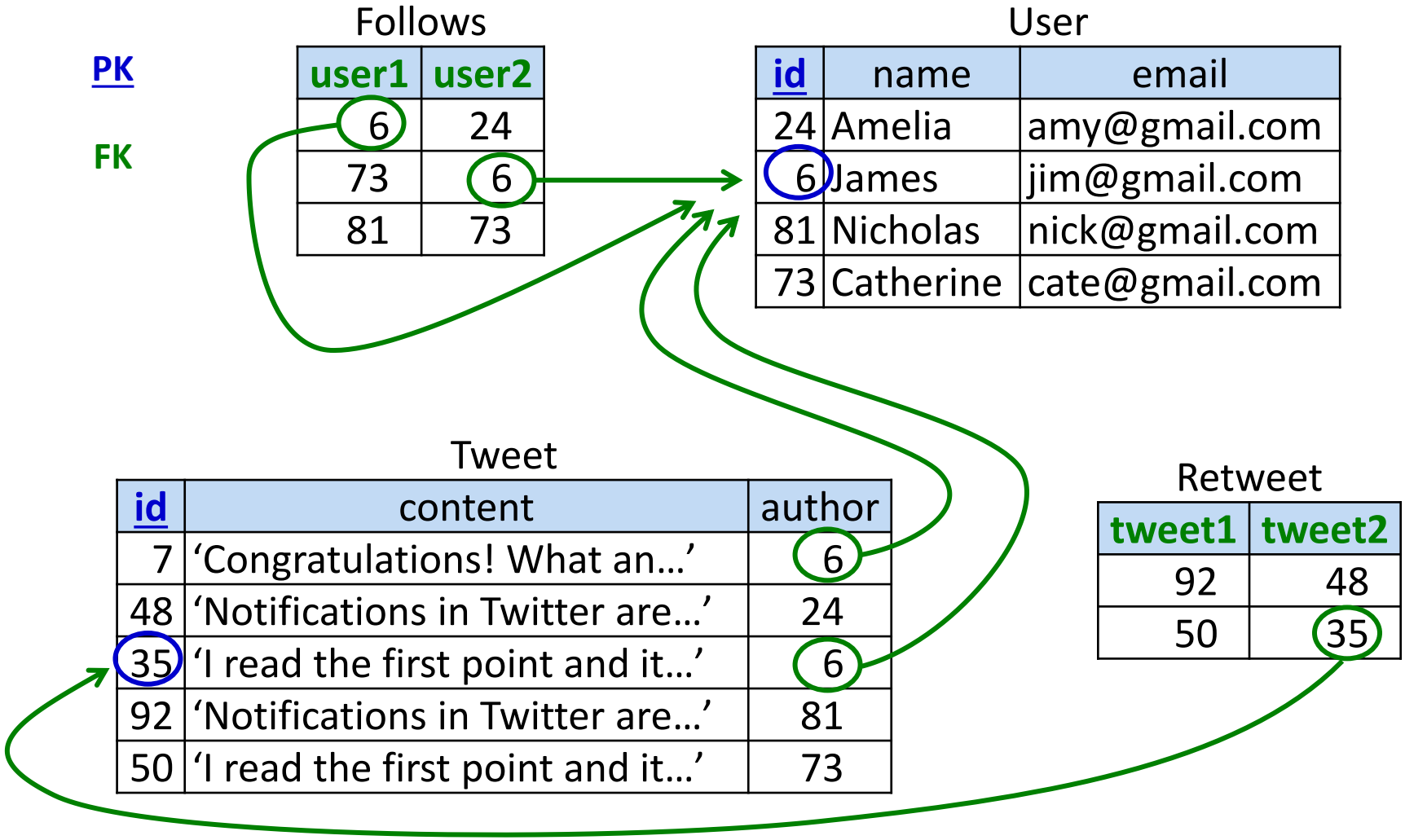
Retweet

tweet1	tweet2
92	48
50	35

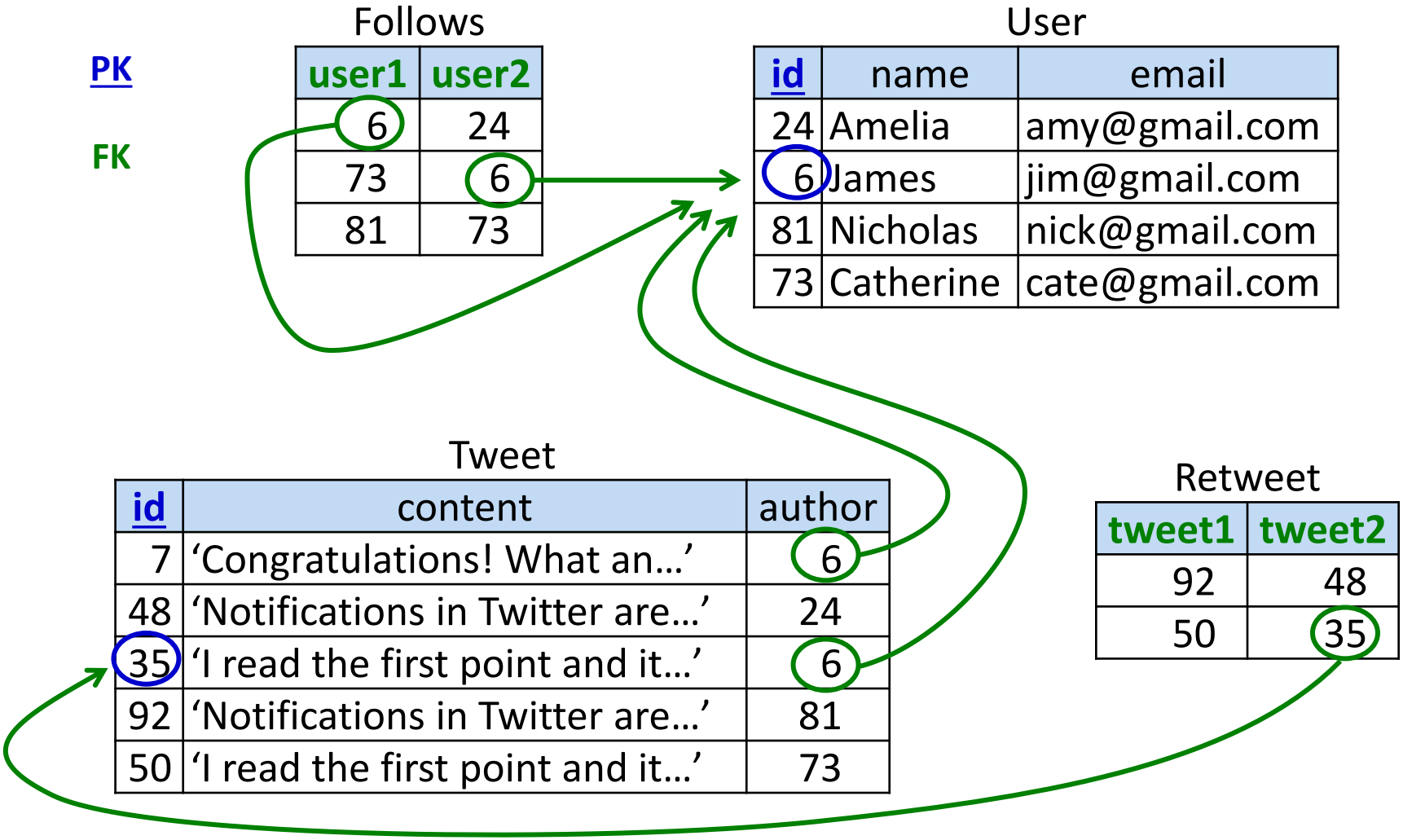
Example: primary and foreign keys



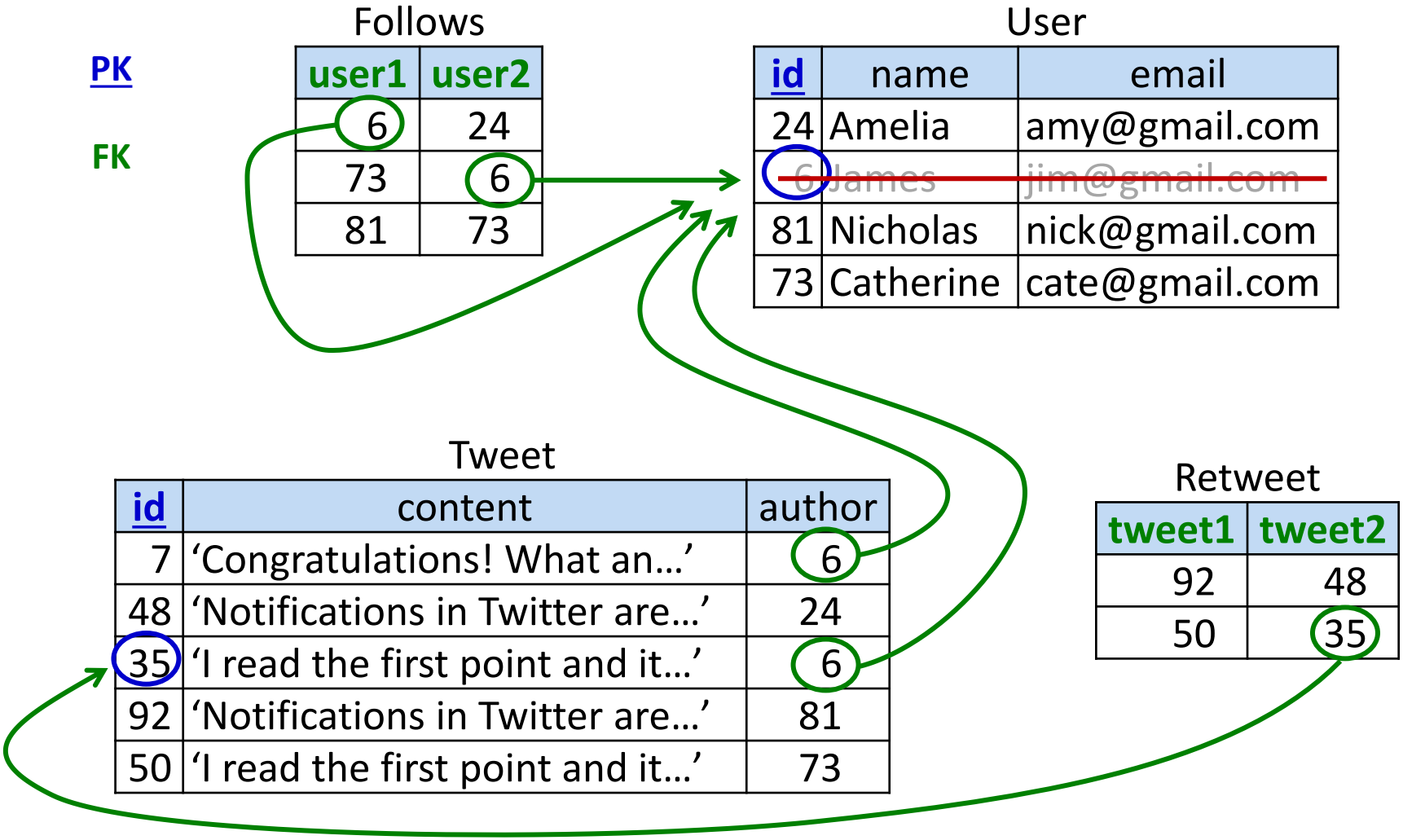
Example: primary and foreign keys



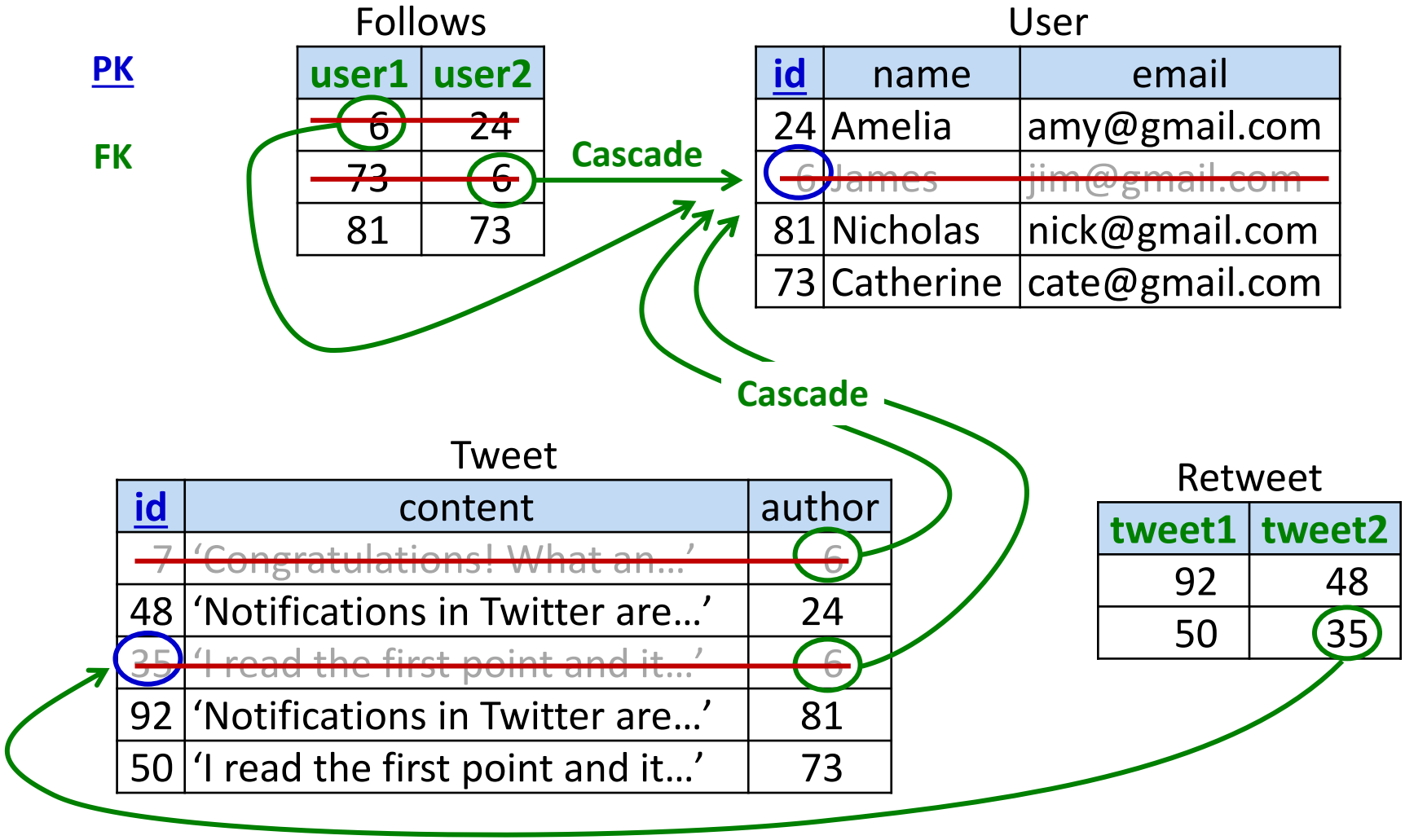
Example: primary and foreign keys



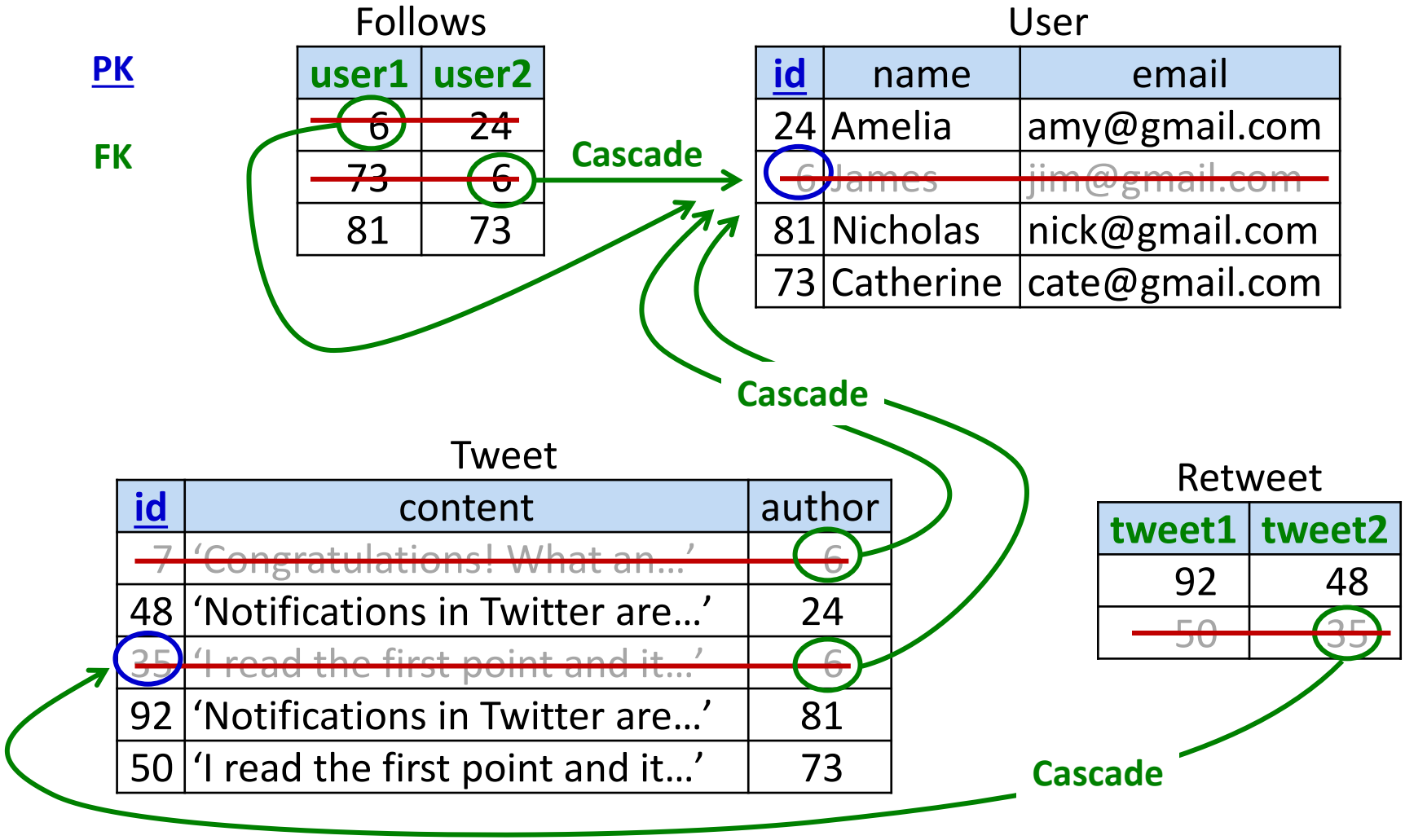
Example: on delete



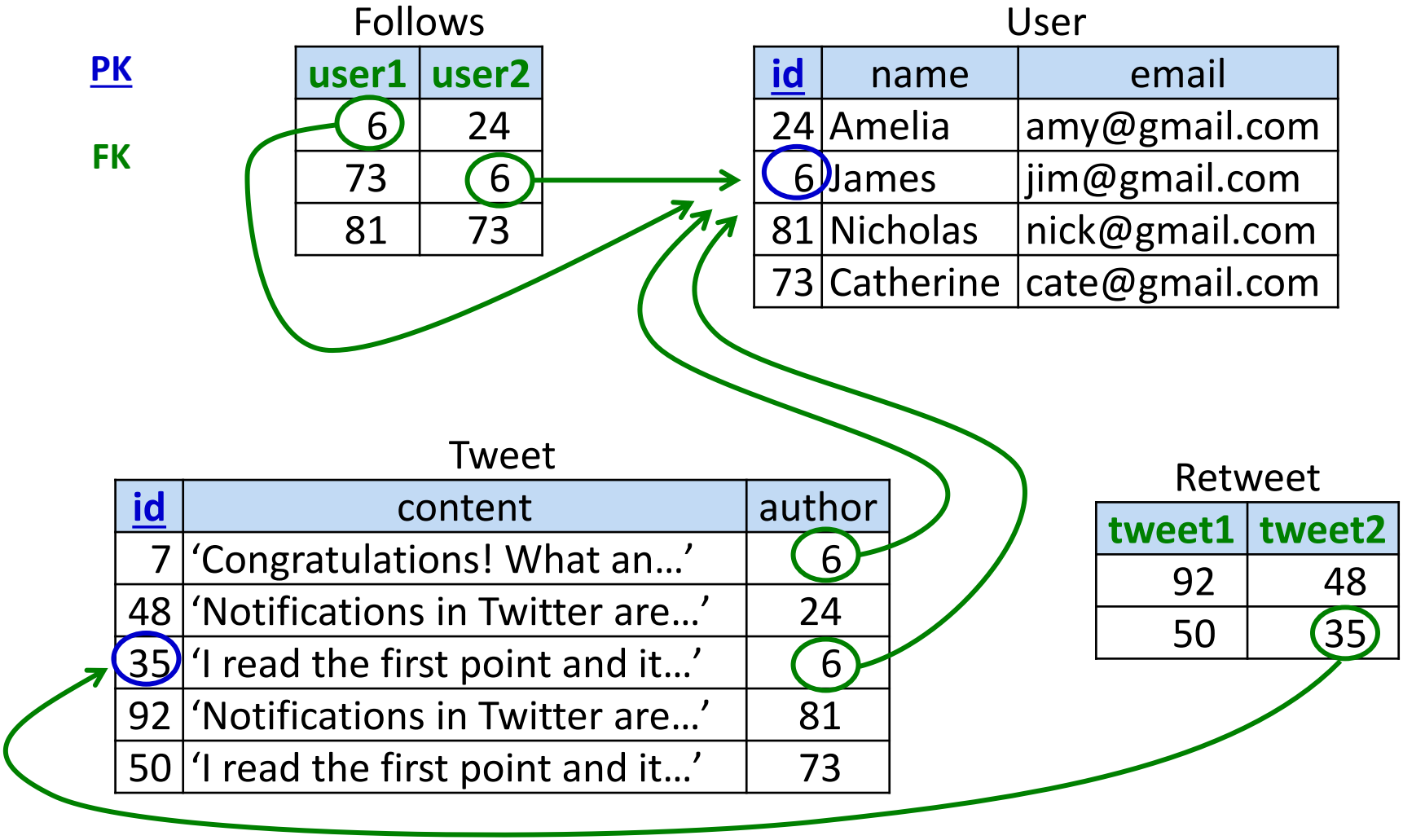
Example: on delete cascade



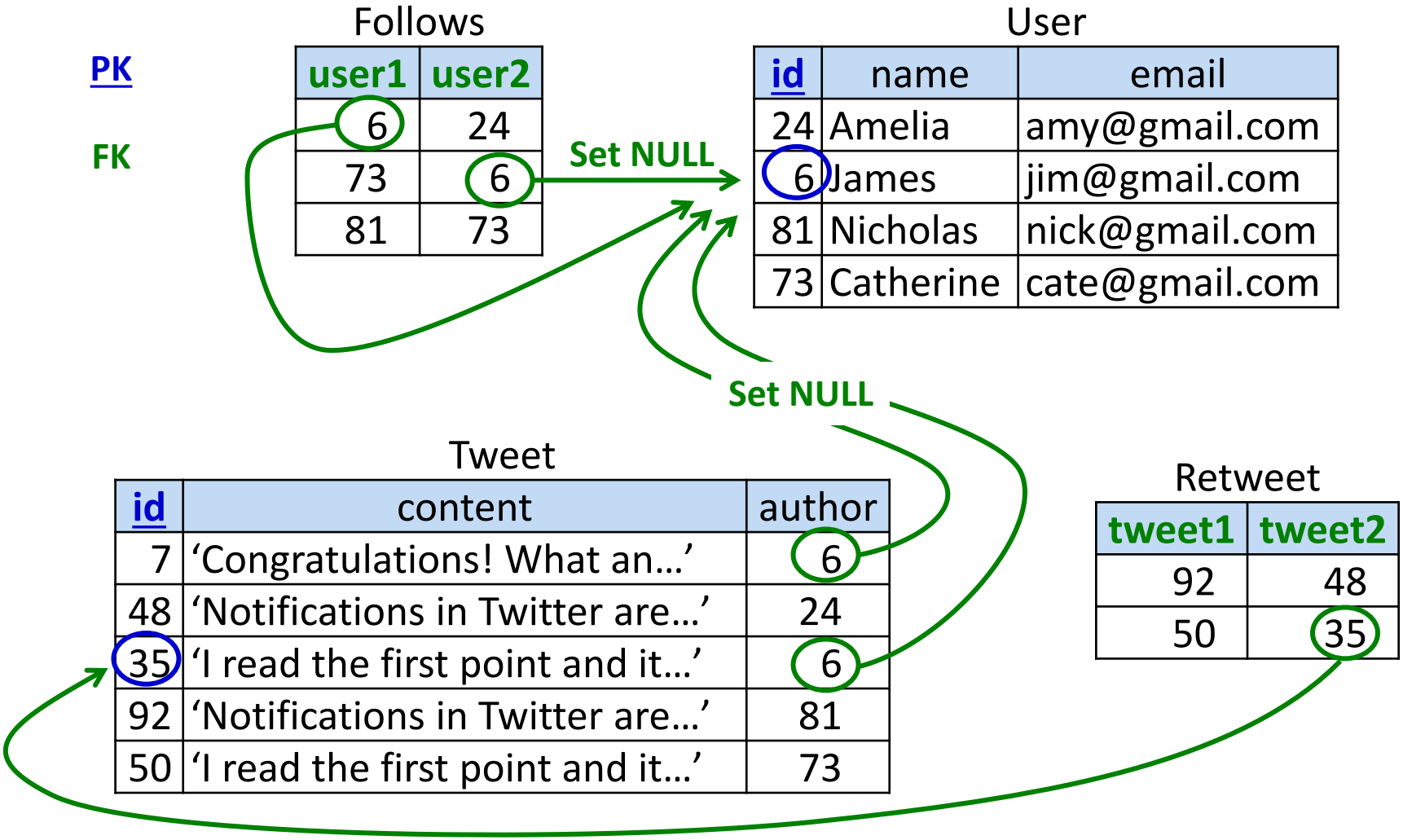
Example: on delete cascade



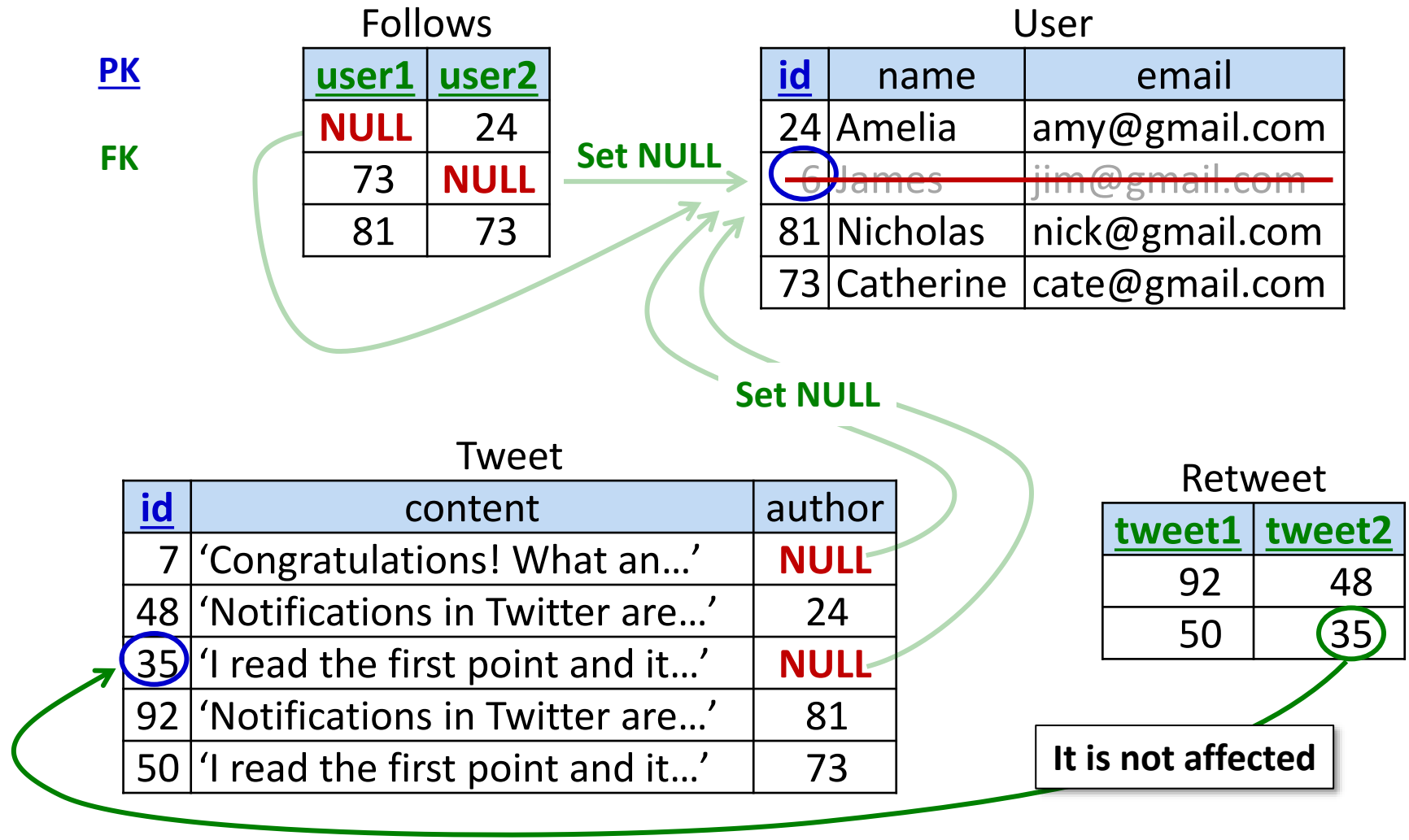
Example: on delete



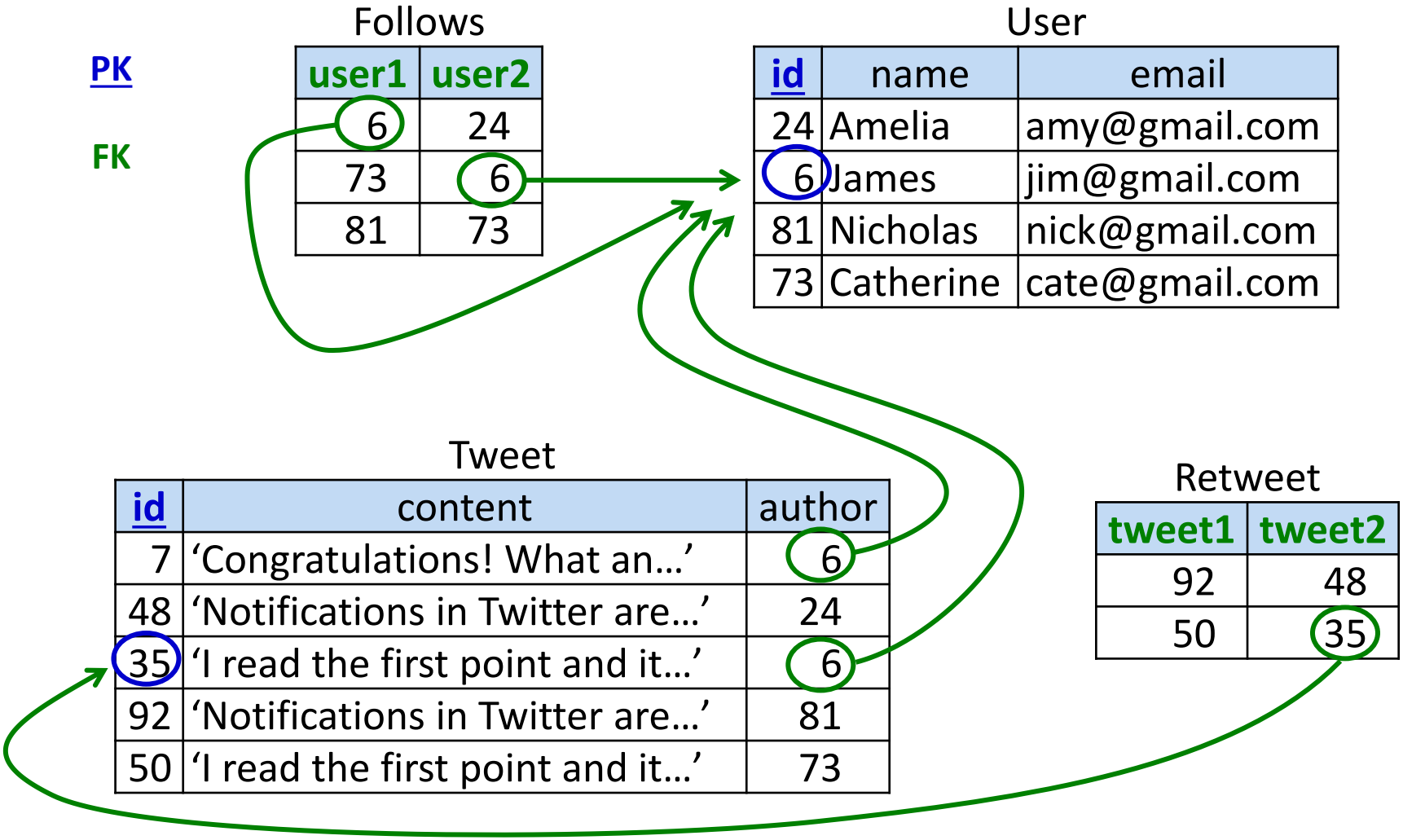
Example: on delete set null



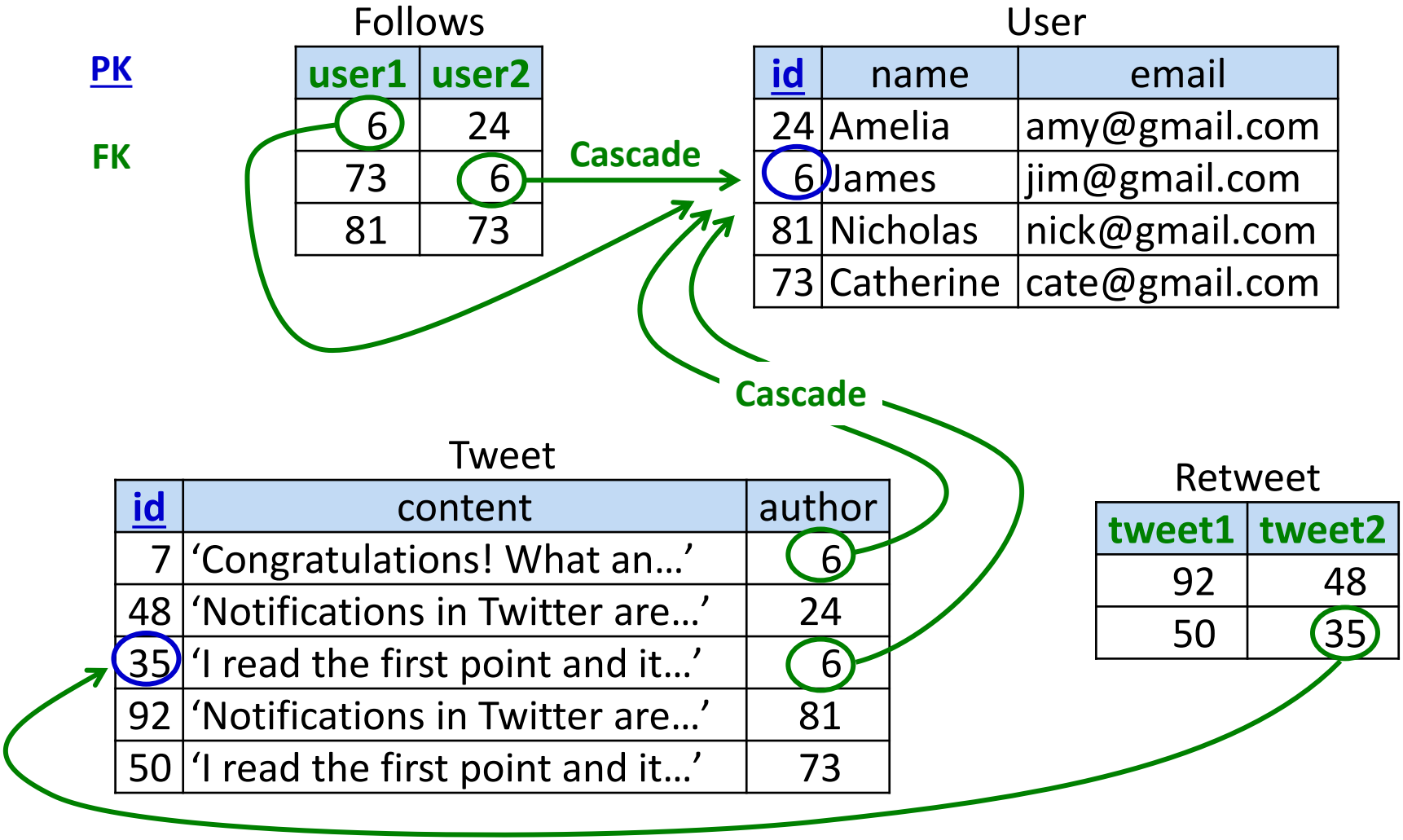
Example: on delete set null



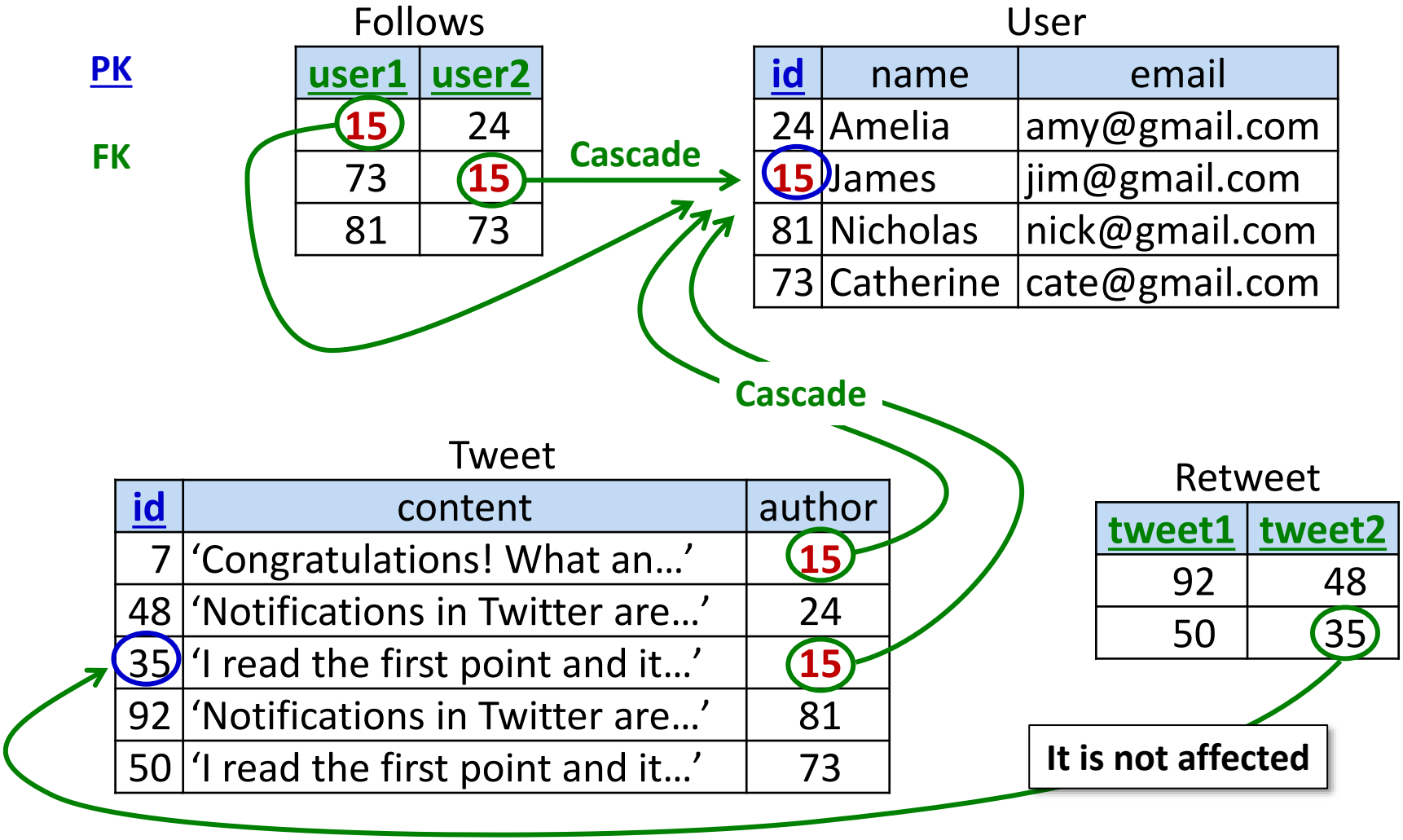
Example: on update



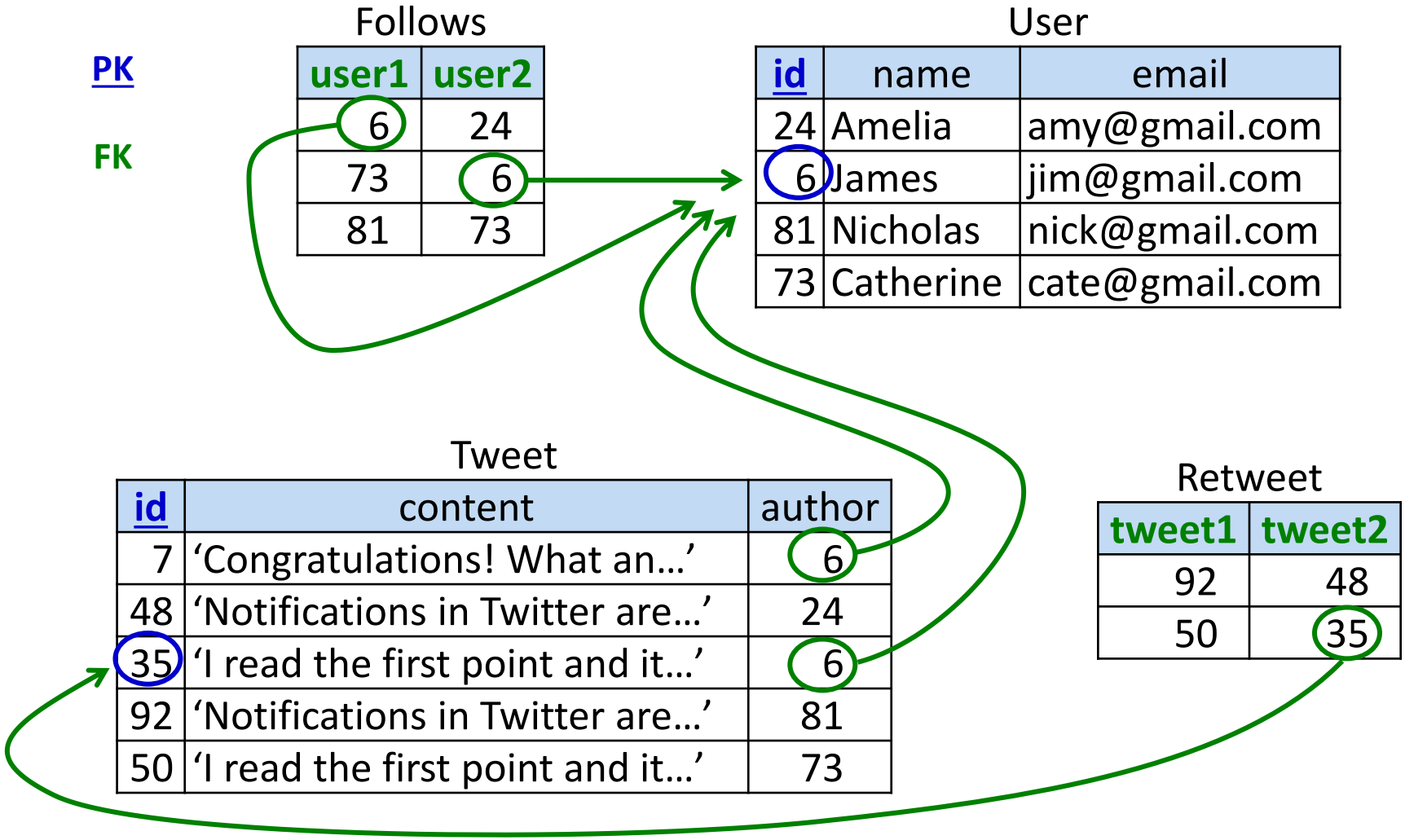
Example: on update cascade



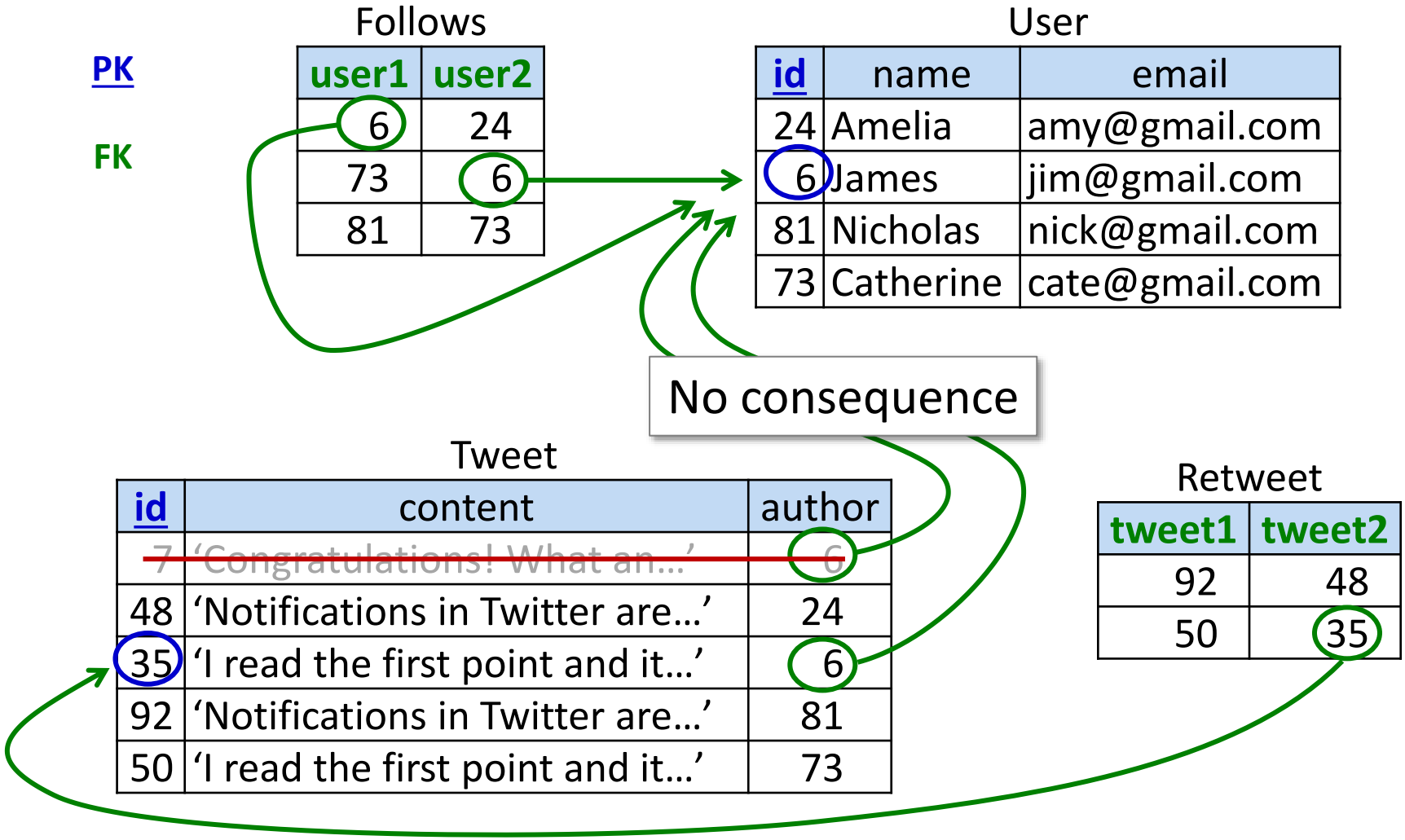
Example: on update cascade



What if references are deleted?



What if references are deleted?



Motivation

Database for music artist, songs, users, play records, social network

Example queries

Songs of the 60's (*show title and genre*)

Set of all nationalities of artists in the DB

Songs of artists from the UK

Titles of songs that a user ever listened to

All the friends of a given user

Users with the same name

All users at distance 2 of a given user in the social network

Common friends to two given users

How many times a song has been played

Artists sorted from most to least played

Users with more than two friends

Users sorted by their number of friends

User with most friends

Data query

SELECT [DISTINCT] *fields* FROM *tables*

[WHERE *condition*];

Can be expressions involving fields

Examples:

```
SELECT title, genre FROM Song
WHERE release_date > '1959-12-31' AND release_date < '1970-01-01';

SELECT DISTINCT nationality FROM Artist;

/* Several tables */
SELECT * FROM Song, Artist
WHERE Song.author = Artist.id AND Artist.nationality = 'UK';

/* Expressions */
SELECT sid, theory * 0.6 + labs * 0.4 FROM Grades;
```

Motivation

Database for music artist, songs, users, play records, social network

Example queries

- *Songs of the 60's* (*show title and genre*)
- *Set of all nationalities of artists in the DB*
- *Songs of artists from the UK*
- *Titles of songs that a user ever listened to*
- *All the friends of a given user*
- *Users with the same name*
- *All users at distance 2 of a given user in the social network*
- *Common friends to two given users*
- *How many times a song has been played*
- *Artists sorted from most to least played*
- *Users with more than two friends*
- *Users sorted by their number of friends*
- *User with most friends*

Join

SELECT *fields*

FROM *table1* **JOIN** *table2* **ON** *condition*

[WHERE *condition*];

Typically (though not only) with foreign keys: ON *foreign* = *primary*
More efficient (?) than cartesian product (i.e. just aggregating tables)

Examples:

Semantically equivalent

```
SELECT title FROM Song, Played
WHERE user = 'amy' AND played_song = id;
```

```
SELECT title FROM Song JOIN Played ON played_song = id
WHERE user = 'amy';
```

```
SELECT * FROM Friend JOIN User
ON (user1 = username OR user2 = username)
WHERE name = 'Catherine';
```

Types of join

◆ INNER	By default (no need to indicate it)
◆ LEFT | RIGHT | FULL	Rows that do not fulfil the condition are also included (incompatible with INNER), especially useful in certain queries with aggregative operations
◆ NATURAL	The condition is of equality between fields with the same name

table1 JOIN *table2* USING (*fields*) is equivalent to a natural join restricted to the fields indicated by 'using'.

Types of join (cont)

Example

```
CREATE TABLE Student (
    sid VARCHAR(12) PRIMARY KEY, name text);
CREATE TABLE Course (
    cid NUMERIC PRIMARY KEY, name text);
CREATE TABLE Grades (
    sid VARCHAR(12) REFERENCES Student(sid),
    cid NUMERIC REFERENCES Course(cid),
    theory NUMERIC (4,2),  labs NUMERIC (4,2),
    PRIMARY KEY (sid, cid));

SELECT name, theory FROM Grades NATURAL JOIN Course;

SELECT name, theory FROM Grades JOIN Course
ON Grades.cid = Course.cid;
```

Motivation

Database for music artist, songs, users, play records, social network

Example queries

- *Songs of the 60's* (*show title and genre*)
- *Set of all nationalities of artists in the DB*
- *Songs of artists from the UK*
- *Titles of songs that a user ever listened to*
- *All the friends of a given user*
- *Users with the same name*
- *All users at distance 2 of a given user in the social network*
- *Common friends to two given users*
- *How many times a song has been played*
- *Artists sorted from most to least played*
- *Users with more than two friends*
- *Users sorted by their number of friends*
- *User with most friends*

Alias

SELECT *fields* FROM *table* **AS** *alias* [(*alias-field1*, *alias-field2*, …)]

[WHERE *condition*];

SELECT *field* **AS** *alias* FROM …

Example:

SELECT u1.name FROM User **AS** u1, User **AS** u2

WHERE u1.name = u2.name AND u1.username < > u2.username;

SELECT sid, theory * 0.6 + labs * 0.4 **AS** average FROM Grades;

Motivation

Database for music artist, songs, users, play records, social network

Example queries

- *Songs of the 60's* (*show title and genre*)
- *Set of all nationalities of artists in the DB*
- *Songs of artists from the UK*
- *Titles of songs that a user ever listened to*
- *All the friends of a given user*
- *Users with the same name*
- *All users at distance 2 of a given user in the social network*
- *Common friends to two given users*
- *How many times a song has been played*
- *Artists sorted from most to least played*
- *Users with more than two friends*
- *Users sorted by their number of friends*
- *User with most friends*

Nested queries

SELECT *fields* FROM (SELECT …) AS *alias* WHERE ...;

SELECT *fields* FROM *table*

WHERE *field1*, *field2*, … **IN** (SELECT *field1*, *field2*, …);

SELECT *fields* FROM *table*

WHERE *field comparison* (**SOME** | **ALL**) (SELECT …);

SELECT *fields* FROM *table*

WHERE [NOT] **EXISTS** (SELECT …);

SELECT *fields* FROM *table*

WHERE (SELECT …) **CONTAINS** (SOME | ALL) (SELECT …);

Nested queries (cont)

Examples:

```
SELECT c1.user1
FROM Friend AS c1 JOIN
(SELECT * FROM Friend WHERE Friend.user2 = 'amy')
AS c2          /* In FROM always with AS */
ON c1.user2 = c2.user1
```

```
SELECT user1 FROM Friend
WHERE user2 IN (SELECT user1 FROM Friend WHERE user2 = 'amy')
```

Motivation

Database for music artist, songs, users, play records, social network

Example queries

- *Songs of the 60's* (*show title and genre*)
- *Set of all nationalities of artists in the DB*
- *Songs of artists from the UK*
- *Titles of songs that a user ever listened to*
- *All the friends of a given user*
- *Users with the same name*
- *All users at distance 2 of a given user in the social network*
- *Common friends to two given users*
- *How many times a song has been played*
- *Artists sorted from most to least played*
- *Users with more than two friends*
- *Users sorted by their number of friends*
- *User with most friends*

Set algebra

query1 UNION *query2*

query1 INTERSECT *query2*

query1 EXCEPT *query2*

- Homogeneous tuples: the sets of tuples need to have the same fields
- An implicit DISTINCT is applied (unless we indicate ALL)
- Can typically be reformulated as a simple query with AND, OR, NOT in the WHERE condition → the advantage could be of readability in some cases

Example:

```
(SELECT user2 FROM Friend  WHERE user1 = 'cate'
 UNION
 SELECT user1 FROM Friend WHERE user2 = 'cate')
INTERSECT
(SELECT user2 FROM Friend WHERE user1 = 'amy'
 UNION
 SELECT user1 FROM Friend WHERE user2 = 'amy')
```

Motivation

Database for music artist, songs, users, play records, social network

Example queries

Songs of the 60's (*show title and genre*)

Set of all nationalities of artists in the DB

Songs of artists from the UK

Titles of songs that a user ever listened to

All the friends of a given user

Users with the same name

All users at distance 2 of a given user in the social network

Common friends to two given users

How many times a song has been played

Artists sorted from most to least played

Users with more than two friends

Users sorted by their number of friends

User with most friends

Order and aggregation

SELECT COUNT ([DISTINCT] *field*) FROM *table* …

[**GROUP BY** *field1*, *field2*, … [**HAVING** *condition*]];

Post-aggregation filter

SELECT SUM | MAX | MIN | AVG (*field*) FROM *table* …

[**GROUP BY** *field1*, *field2*, …];

SELECT …

[**ORDER BY** *field1*, *field2*, … [DESC]];

Why?

Useful combined with LIMIT *n*

In general when GROUP BY is used, only those fields can be used in the SELECT clause (though some DBMS tolerate it)

Order and aggregation: examples

SELECT **COUNT** (*) FROM Played JOIN Song ON played_song = id
WHERE title = 'Norwegian Wood';

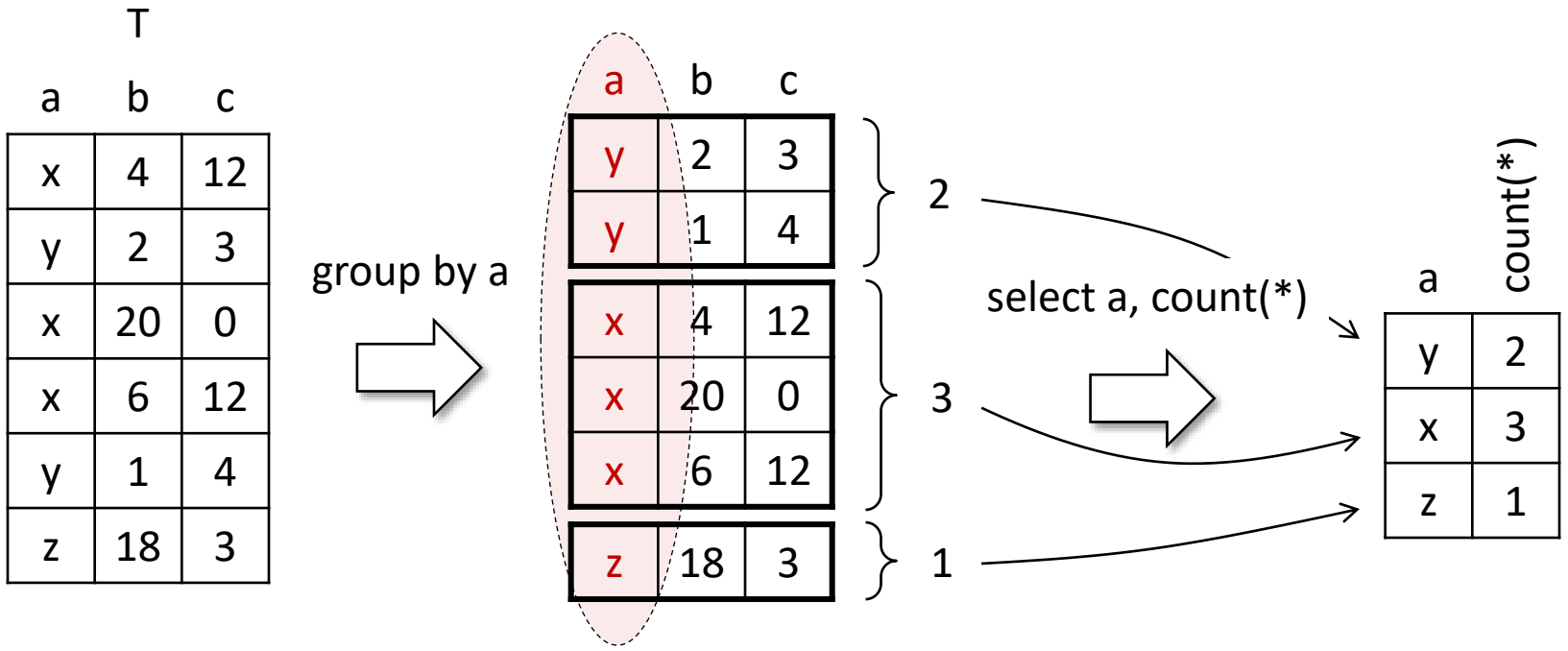
SELECT author, **COUNT** (*) FROM Played JOIN Song ON played_song = id
GROUP BY author;

SELECT author, **COUNT** (*) AS n FROM Played JOIN Song ON played_song = id
GROUP BY author
ORDER BY n
DESC
LIMIT 1

SELECT author FROM Played JOIN Song ON played_song = id
GROUP BY author
HAVING COUNT (*) > 100

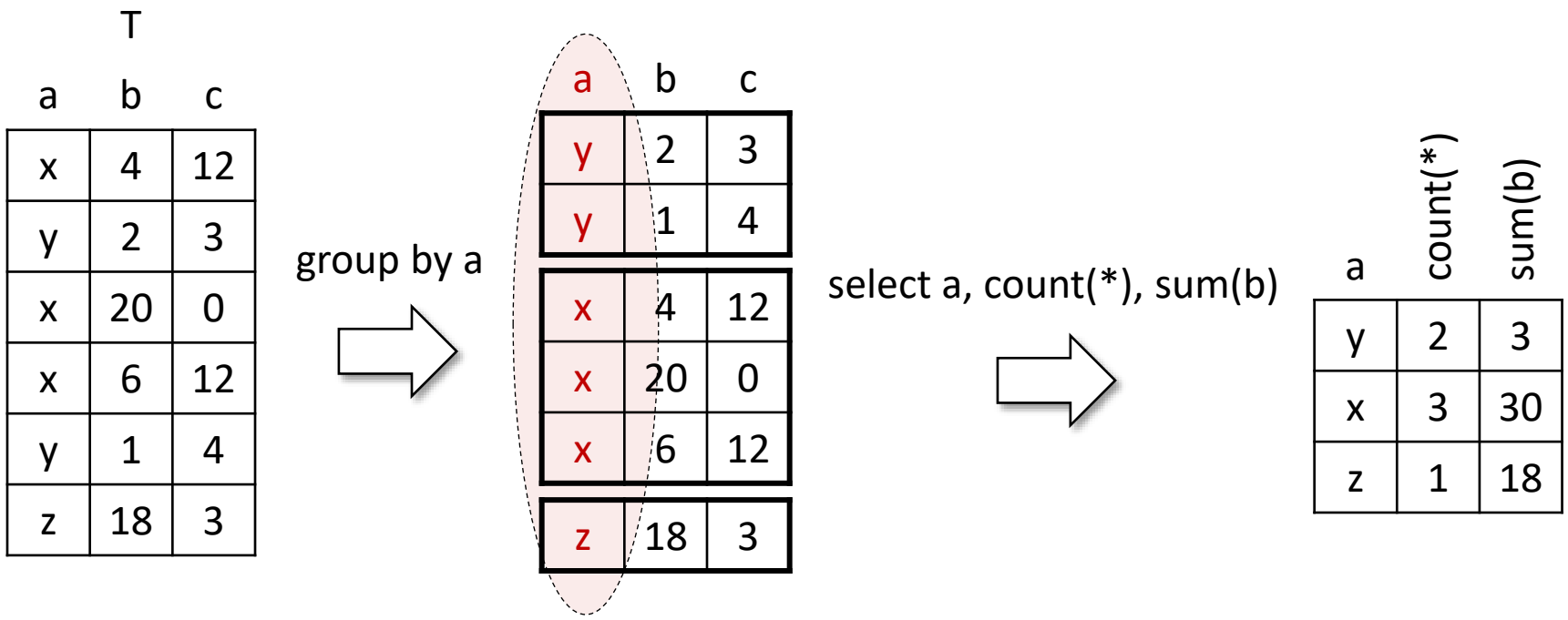
Order and aggregation: example

SELECT a, count(*) FROM T GROUP BY a



Order and aggregation: example

SELECT a, count(*), sum(b) FROM T GROUP BY a



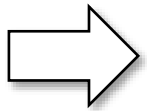
Order and aggregation: example

SELECT a, count(*), sum(b) FROM T GROUP BY a, c

T

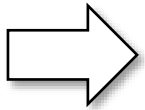
a	b	c
x	4	12
y	2	3
x	20	0
x	6	12
y	1	4
z	18	3

group by a, c



a	b	c
y	2	3
y	1	4
x	4	12
x	6	12
x	20	0
z	18	3

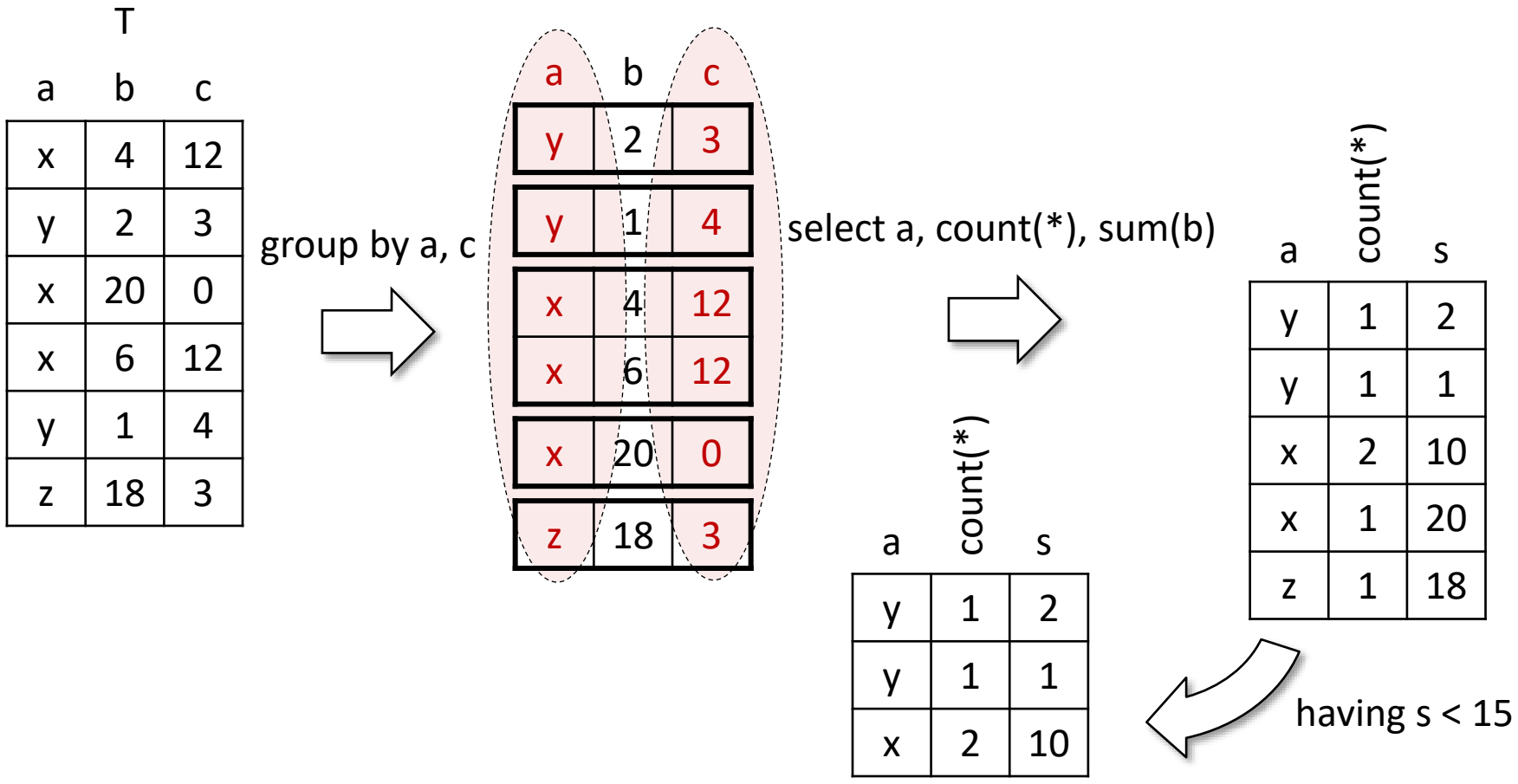
select a, count(*), sum(b)



a	count(*)	sum(b)
y	1	2
y	1	1
x	2	10
x	1	20
z	1	18

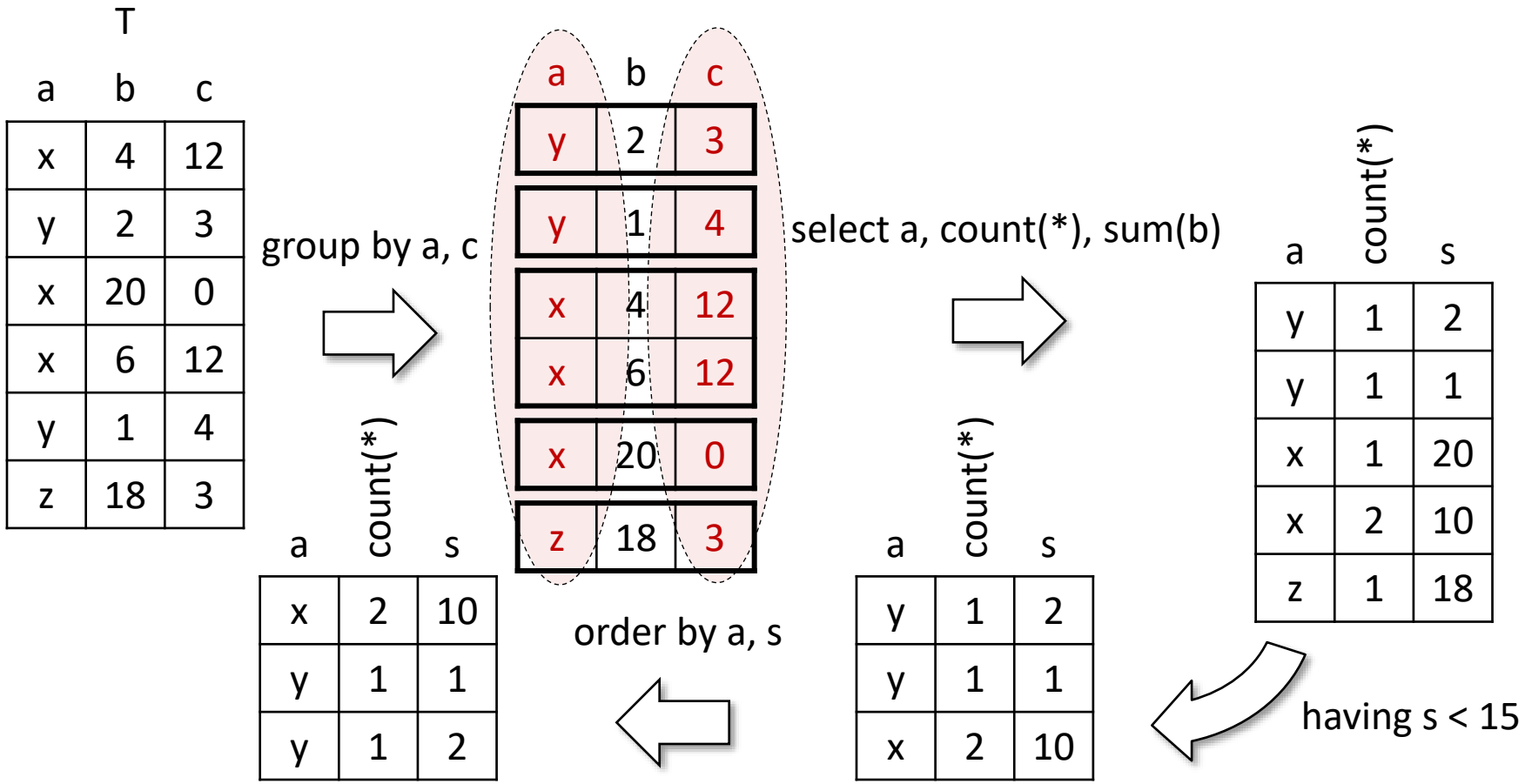
Order and aggregation: example

SELECT a, count(*), sum(b) as s FROM T GROUP BY a, c
HAVING s < 15



Order and aggregation: example

SELECT a, count(*), sum(b) as s FROM T GROUP BY a, c
HAVING s < 15
ORDER BY a, s



Views

CREATE VIEW *name* **AS** SELECT…;

A way to assign a name to a query

Equivalent to nested query, but…

- Useful to reuse nested queries and improve readability of complex queries
- Views are always up to date when the data change
- Can be configured to be stored on disk (to save re-execution cost)

Examples:

```
CREATE VIEW UserFriends AS
SELECT u1.username, u2.name FROM User AS u1, User AS u2
WHERE (u1.username, u2.username) IN
        ((SELECT user1, user2 FROM Friend)
         UNION (SELECT user2, user1 FROM Friend));

SELECT name FROM UserFriends WHERE username = 'jim';
```

Other elements of SQL…

- Schemas
  - To define namespaces for tables, similar to e.g. Java packages
- Domains
  - Data types defined by properties and conditions over a primitive data type (e.g. a string with a certain format defined by a regular expression)
- Triggers
  - Run a procedure when certain update actions (insert, update, delete) are produced in a table
- Asserts
  - Taylor-made checks over several rows and/or several tables
- Transactions
  - Establish sequences of actions and queries that must always be completed or cancelled together as a block
  - Also allow to synchronize (block) concurrent operations
- And many more basic functionalities supported by each DBMS, extending the SQL standard…

Practical informal comments

- When **several tables** are included in **in the from clause**, two types of conditions will be usually defined in the where clause:
  - The ones expressed in the 'natural' query
  - The ones that connect tables to each other (not expressed in the query)
    - Typically foreign key = primary key
    - Each table is connected to some other, in such a way that all tables are connected
    - This is not mandatory, but it is usual for a query to make sense
- The **alias** are used mainly…
  - On tables: when the same table is involved several times in the same query
  - On fields: to use fields defined by operations (typically in nested queries, or with aggregation)
- In an initial version of a query it may not be worthwhile to use **join**
  - Let the SQL engine optimize the order of operations
  - If (or when) we have a crisp idea of the optimal order, then go for **joins**
  - External joins can also be useful in certain queries with aggregation

Practical informal comments (cont)

- **Nested queries** are typically useful…
  - To facilitate complex operations using e.g. not exists, not in
  - To concatenate aggregations, e.g. “Average [avg] number [count] of followers”
  - As views, to ease up complex queries and/or reuse subqueries
  - In other cases it is very often possible to write queries without nesting
- Queries can generally be written in different equivalent ways
  - Some ways are usually more efficient than others
  - In particular, queries with IN can be generally rewritten with EXISTS and vice-versa, with EXCEPT when they are negated, etc.
  - And any query with OR, AND and NOT in the where clause can be rewritten as a UNION, INTERSECT, EXCEPT
- The order in which the parts of a query are executed is:
  - FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY