Databases

Lecture 6

I. Normalization. 3NF, NFBC

Functional dependency

Definition of functional dependence (according to J. Date):

- Let r be a relation, and X and Y be arbitrary subsets of the set of attributes of relation r. Then Y is functionally dependent on X, which is written symbolically as X → Y (read either "X functionally determines Y" or "X is an arrow of Y")
- if and only if every value of the set X of the relation r is associated with exactly one value of the set Y of the relation r.

Functional dependency

Definition of functional dependence (according to J. Date):

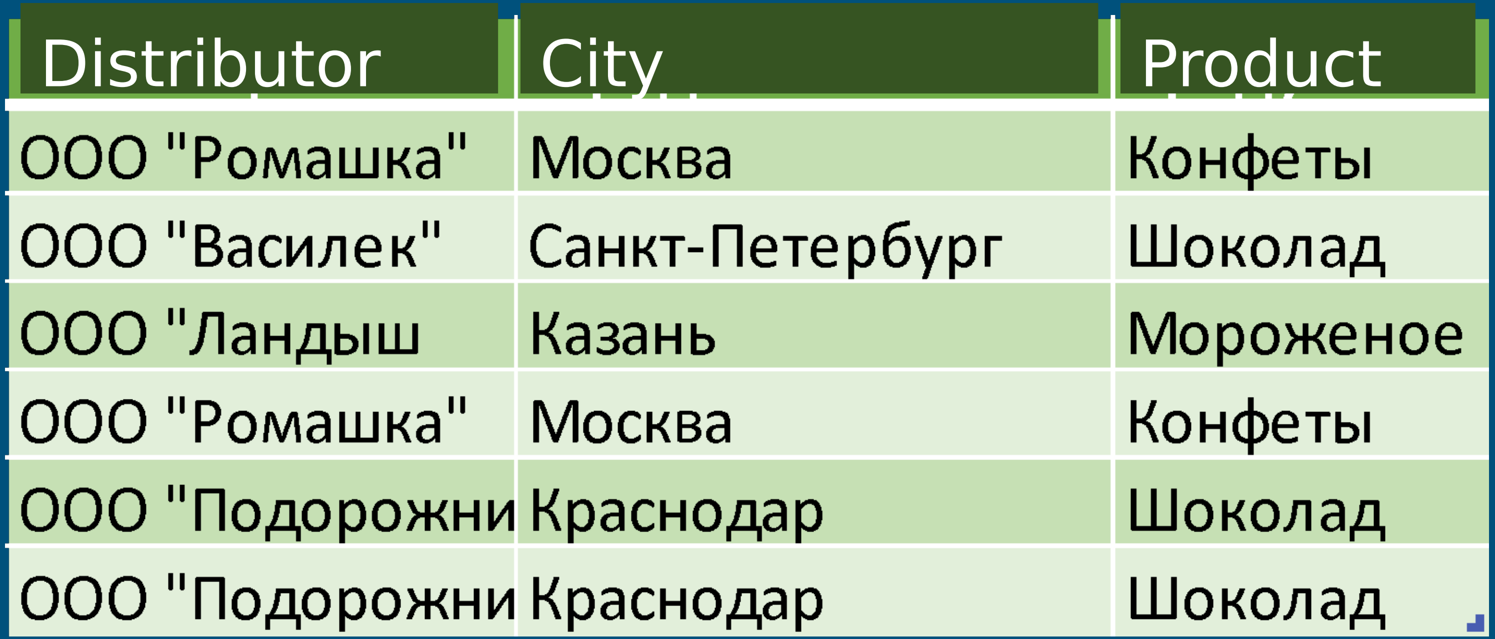
- In other words, if two tuples of a relation r coincide in the value X, they also coincide in the value Y (⇔ there are no two different tuples that have the same values in the X attribute and different values in the Y attribute).
- The left part of the functional dependence record is called the determinant, the right part is the dependent part.

Relation between keys and functional dependencies

Heath’s theorem.

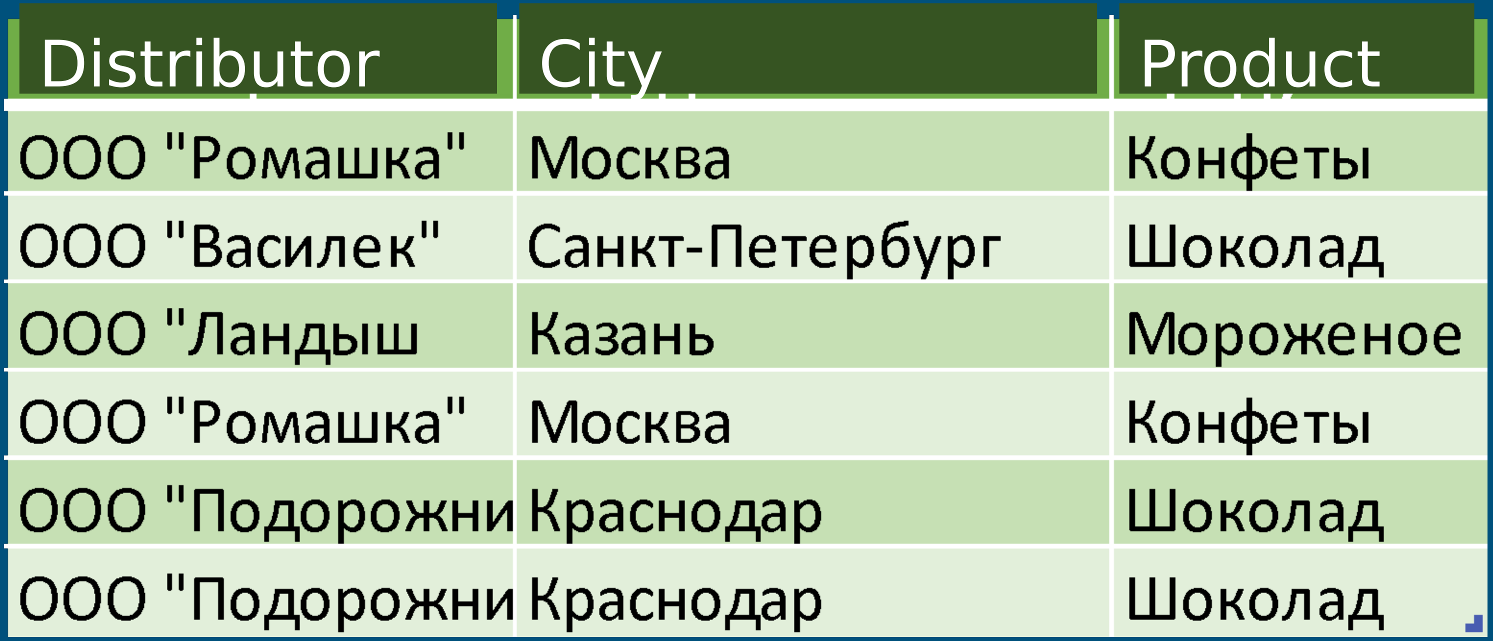
Let’s consider a relation R(A,B,C), where A,B,C are attributes. If R satisfies the functional dependency A -> B, then R is a union of R’s projections on attributes (A,B) and (A,C)

Functional dependency: Heath’s theorem



Distributor	City	Product
ООО "Ромашка"	Москва	Конфеты
ООО "Василек"	Санкт-Петербург	Шоколад
ООО "Ландыш	Казань	Мороженое
ООО "Ромашка"	Москва	Конфеты
ООО "Подорожни	Краснодар	Шоколад
ООО "Подорожни	Краснодар	Шоколад

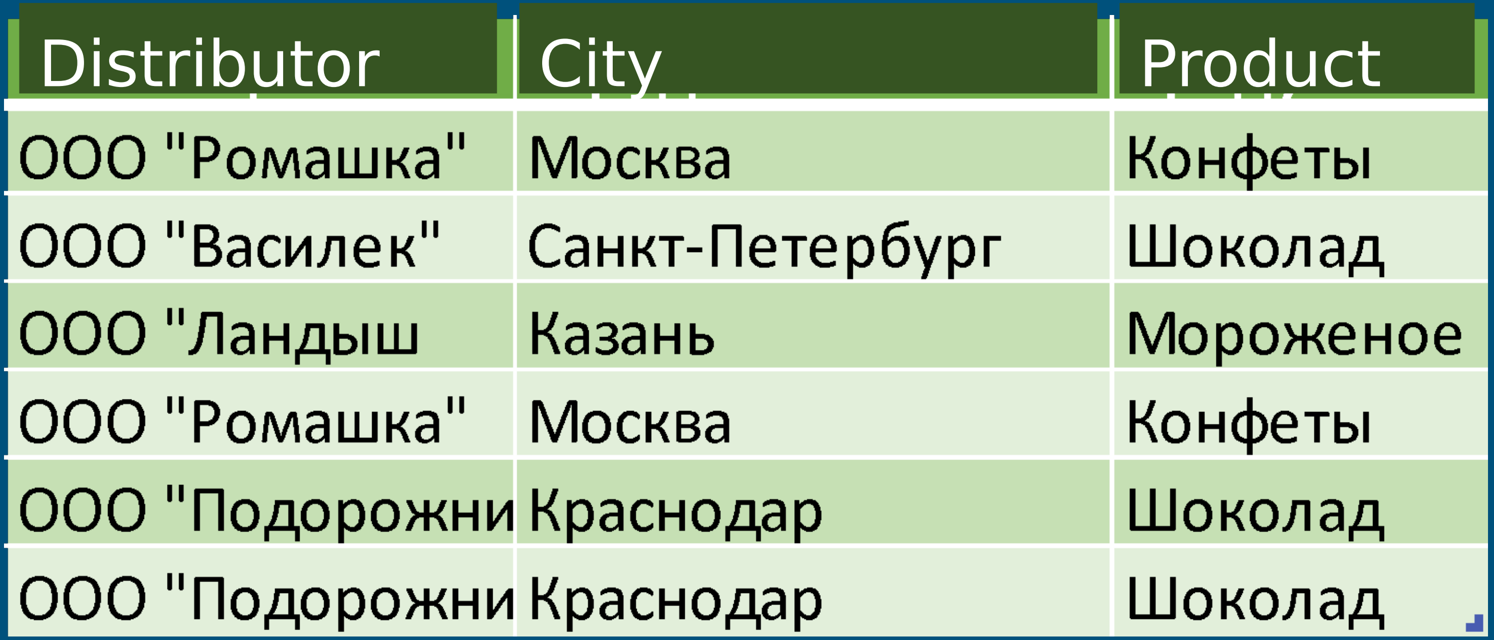
Functional dependency: Heath’s theorem



Distributor	City	Product
ООО "Ромашка"	Москва	Конфеты
ООО "Василек"	Санкт-Петербург	Шоколад
ООО "Ландыш	Казань	Мороженое
ООО "Ромашка"	Москва	Конфеты
ООО "Подорожни	Краснодар	Шоколад
ООО "Подорожни	Краснодар	Шоколад

It is obvious that the data in the table above is redundant - information about suppliers, cities and products is duplicated in different rows

Functional dependency: Heath’s theorem



Distributor	City	Product
ООО "Ромашка"	Москва	Конфеты
ООО "Василек"	Санкт-Петербург	Шоколад
ООО "Ландыш	Казань	Мороженое
ООО "Ромашка"	Москва	Конфеты
ООО "Подорожни	Краснодар	Шоколад
ООО "Подорожни	Краснодар	Шоколад

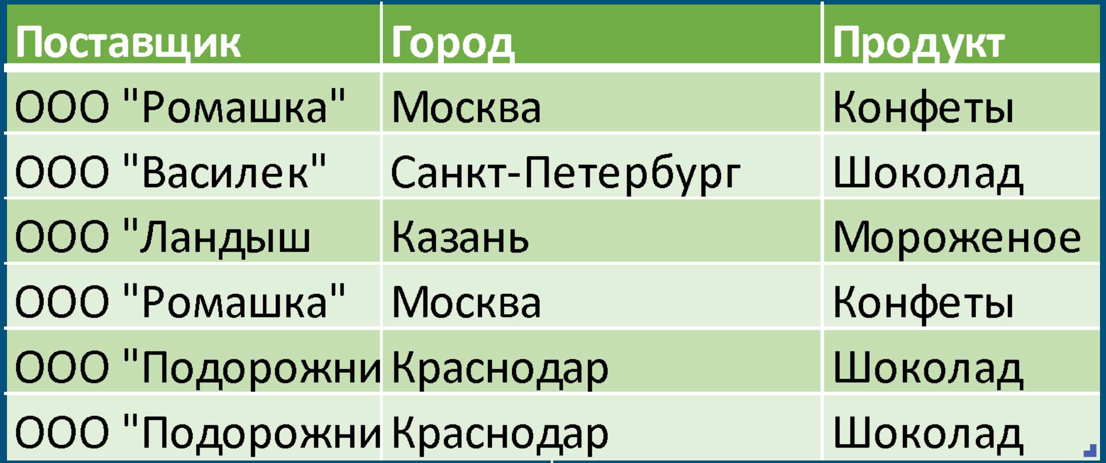
Let’s use Heath’s theorem

Functional dependency: Heath’s theorem

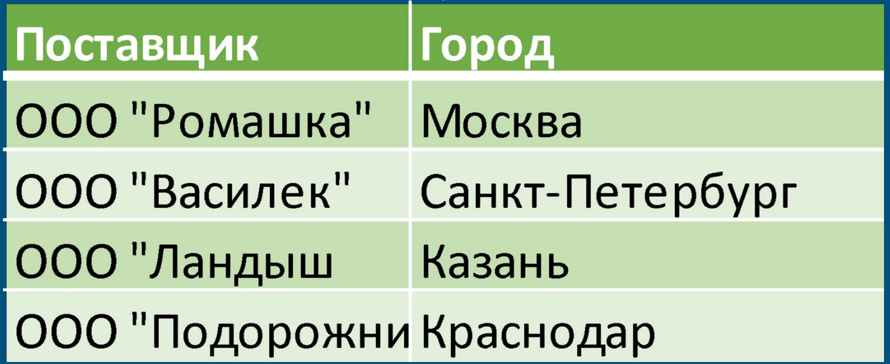
A = {Distributor}
B = {City}
C = {Product}

If A -> B, then R = {AB}$\bowtie${AC}

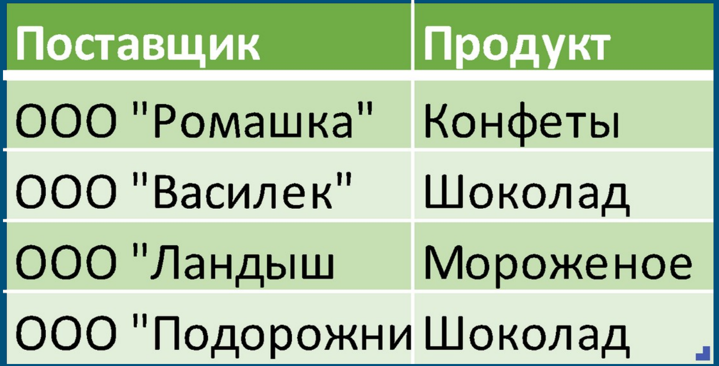
FD: Heath’s theorem



Поставщик	Город	Продукт
ООО "Ромашка"	Москва	Конфеты
ООО "Василек"	Санкт-Петербург	Шоколад
ООО "Ландыш	Казань	Мороженое
ООО "Ромашка"	Москва	Конфеты
ООО "Подорожни	Краснодар	Шоколад
ООО "Подорожни	Краснодар	Шоколад



Поставщик	Город
ООО "Ромашка"	Москва
ООО "Василек"	Санкт-Петербург
ООО "Ландыш	Казань
ООО "Подорожни	Краснодар



Поставщик	Продукт
ООО "Ромашка"	Конфеты
ООО "Василек"	Шоколад
ООО "Ландыш	Мороженое
ООО "Подорожни	Шоколад

3NF

- A relation is in the third normal form if and only if it is in second normal form and no non-key attribute is transitively dependent on its primary key (the definition assumes there is only one candidate key, which is also the relation's primary key)
- Informal definition: A relation is in 3NF if and only if each tuple consists of a primary key value, and a set of mutually independent attributes

3NF: transitive dependency

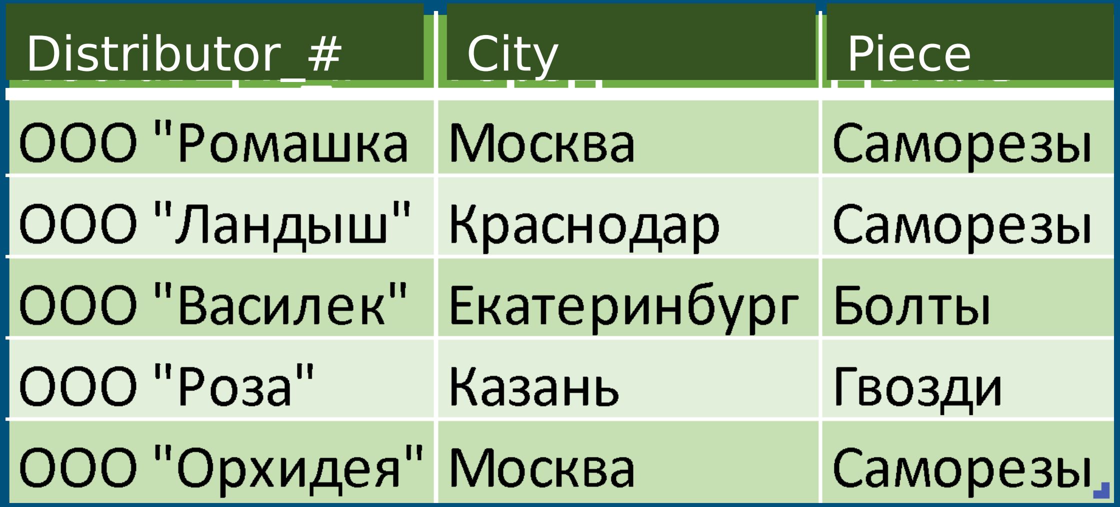
If functional dependencies X -> Y, Y -> Z exist for attributes X, Y, Z of the relation R, then Z has a transitive dependency on attribute X via attribute Y (and X is not functionally dependent on Y or Z).

3NF: example

X - {Distributor_#} - primary key
Y - {City}
Z - {Piece}

{Distributor_#} -> {City},
{City}->{Piece}

{Distributor_#} $\nleftarrow$ {City},
{City}$\nleftarrow${Piece}



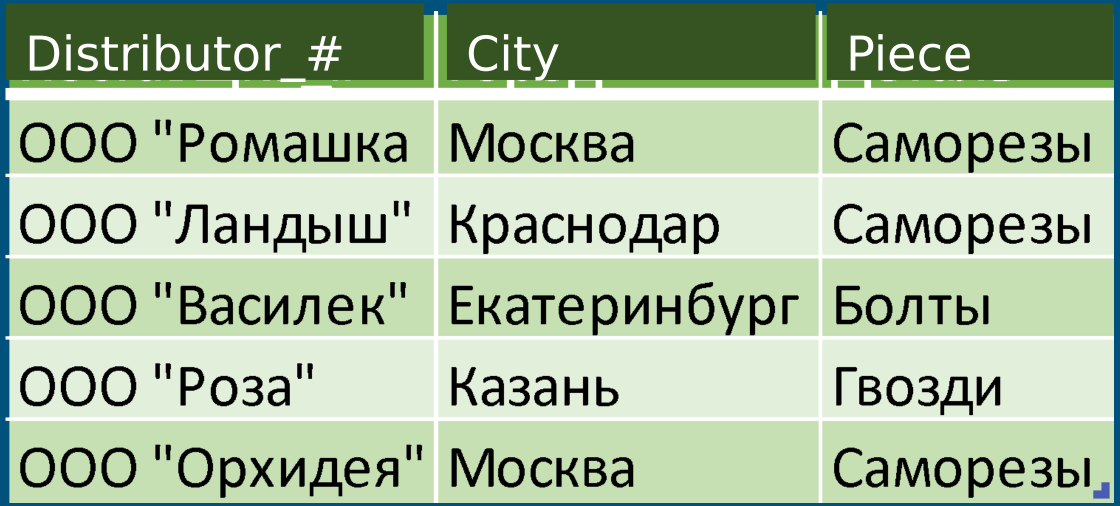
Distributor_#	City	Piece
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

3NF: example

X - {Distributor_#} - primary key
Y - {City}
Z - {Piece}
{Distributor_#} -> {City}, {City}->{Piece}
{Distributor_#} ↚ {City}, {City}↚{Piece}

Let’s try to reduce FD
{Distributor_#} -> {Piece}

R = {Distributor_#, Piece} ⋈{Distributor_#, City}



Distributor_#	City	Piece
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

3NF: example

X - {Distributor_#} - primary key
Y - {City}
Z - {Piece}
{Distributor_#} -> {City}, {City}->{Piece}
{Distributor_#} $\nleftarrow$ {City}, {City}$\nleftarrow${Piece}

Let’s try to reduce FD
{Distributor_#} -> {Piece}

Distributor_#	City	Piece
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

R = {Distributor_#, Piece} ⋈ {Distributor_#, City}
IN GENERAL, THIS IS INCORRECT!

3NF: example

Поставщик_#	Город	Деталь
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

Поставщик_#	Деталь
ООО "Ромашка	Саморезы
ООО "Ландыш"	Саморезы
ООО "Василек"	Болты
ООО "Роза"	Гвозди
ООО "Орхидея"	Саморезы

Поставщик_#	Город
ООО "Ромашка	Москва
ООО "Ландыш"	Краснодар
ООО "Василек"	Екатеринбург
ООО "Роза"	Казань
ООО "Орхидея"	Москва

We’ve lost FD {City} -> {Piece}

3NF: example

X - {Distributor_#} - primary key
Y - {City}
Z - {Piece}
{Distributor_#} -> {City}, {City}->{Piece}
{Distributor_#} ↚ {City}, {City}↚{Piece}

Distributor_#	City	Piece
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

Let's try to reduce FD
{Distributor_#} -> {Piece}

R = {Distributor_#, City} ⋈ {City, Piece}

3NF: example

Поставщик_#	Город	Деталь
ООО "Ромашка	Москва	Саморезы
ООО "Ландыш"	Краснодар	Саморезы
ООО "Василек"	Екатеринбург	Болты
ООО "Роза"	Казань	Гвозди
ООО "Орхидея"	Москва	Саморезы

Поставщик_#	Город
ООО "Ромашка	Москва
ООО "Ландыш"	Краснодар
ООО "Василек"	Екатеринбург
ООО "Роза"	Казань
ООО "Орхидея"	Москва

Город_#	Деталь
Москва	Саморезы
Краснодар	Саморезы
Екатеринбург	Болты
Казань	Гвозди

BCNF

BCNF is used under the following conditions (according to K. Date):

- a relation has two (or more) candidate keys
- these potential keys are composite
- two or more candidate keys overlap (i.e. have at least one attribute in common)

In practice, a set of such conditions does not occur often, so they are limited to 2NF

BCNF

Definition:

- A relation is in BCNF if and only if each of its nontrivial and left irreducible functional dependencies has some potential key as its determinant.
    - *Trivial FD - FD between composite primary key and its attributes*
    - *Left irreducibility = minimal FD*
- Informal definition:
    - A relation variable is in BCNF if and only if the determinants of all its RFs are potential keys

BCNF: example

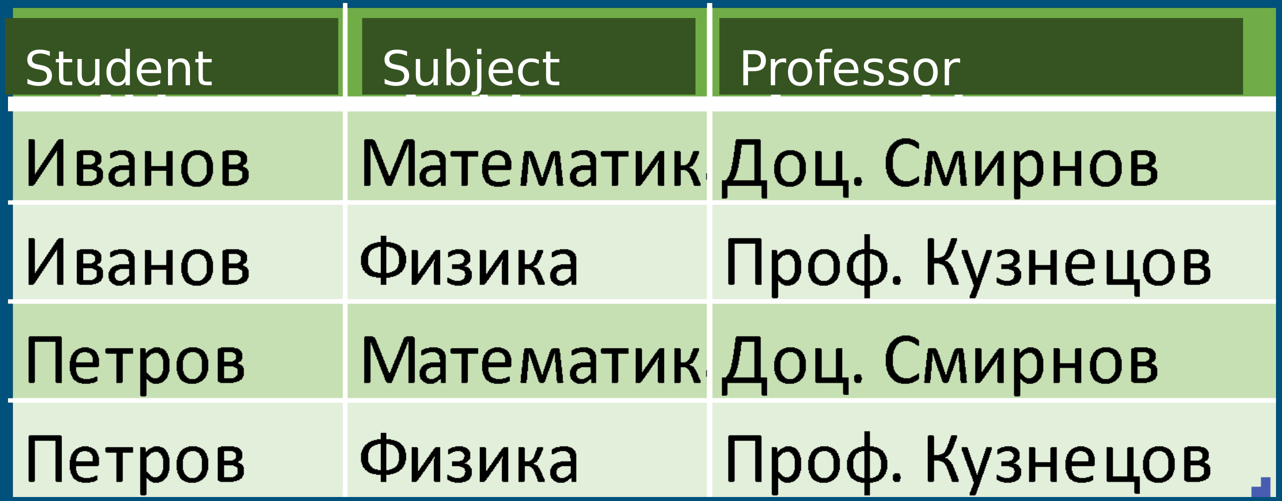
Let’s consider the relation {Student, Subject, Professor}
Let it satisfy the following restrictions

- Every student study the particular subject with only one professor
- Every professor teaches only one subject, but many professors teach the same subject

Then

- {Student, Subject} -> {Professor}
- {Professor} -> {Subject}

BCNF: example



Student	Subject	Professor
Иванов	Математик	Доц. Смирнов
Иванов	Физика	Проф. Кузнецов
Петров	Математик	Доц. Смирнов
Петров	Физика	Проф. Кузнецов

- Potential keys:
  - {Student, Subject}
  - {Student, Professor}

BCNF

Student	Subject	Professor
Иванов	Математик	Доц. Смирнов
Иванов	Физика	Проф. Кузнецов
Петров	Математик	Доц. Смирнов
Петров	Физика	Проф. Кузнецов

Professor	Stude nt
Доц. Смирнов	Иванов
Проф. Кузнецов	Иванов
Доц. Смирнов	Петров
Проф. Кузнецов	Петров

Professor	Subject
Доц. Смирнов	Математика
Проф. Кузнецов	Физика

BCNF: example

We still have a problem.
Decomposition of the original relation on projections ST and TJ let us to exclude one types of anomalies, but add another types.

Initial FD {Student, Subject} -> {Professor} cannot be derived from the single FD {Professor} -> {Subject} that still presents in the data

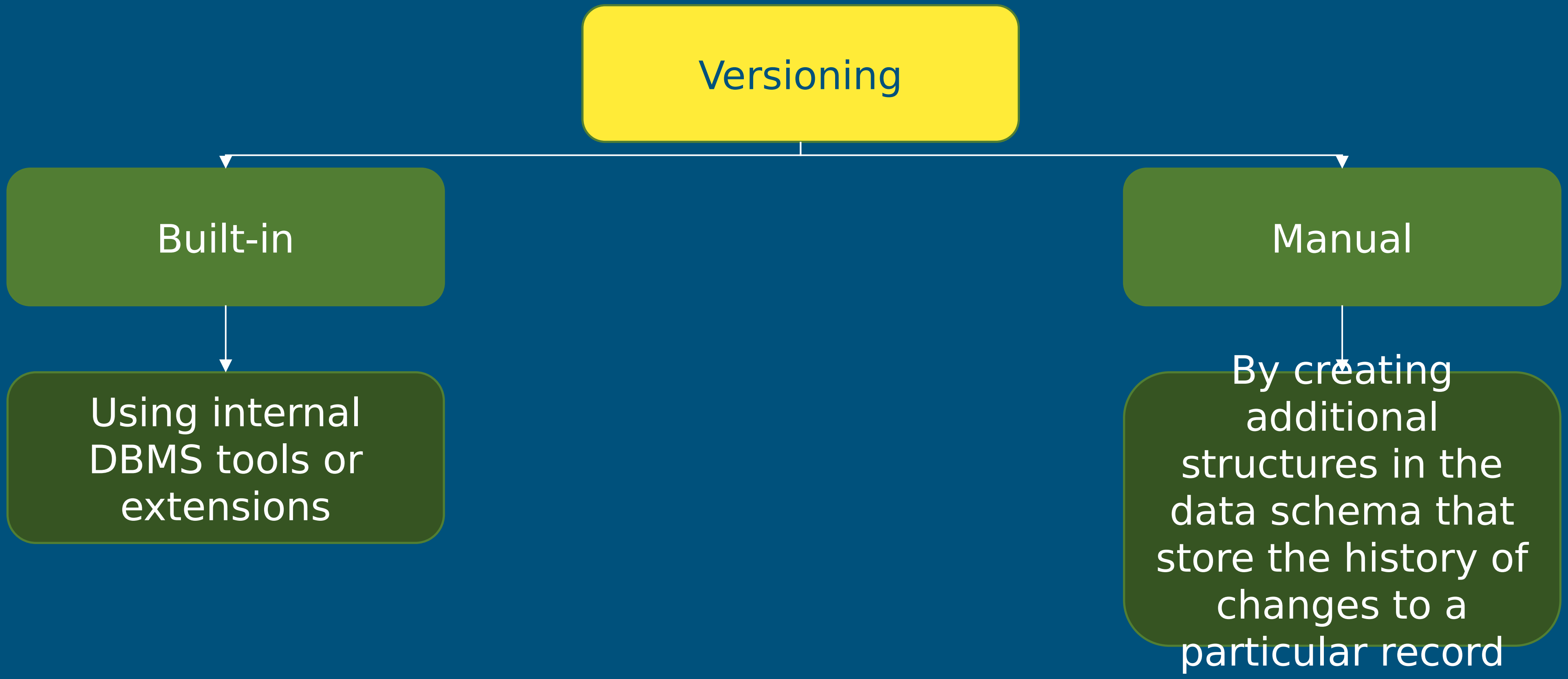
II. Versioning

Versioning

- Versioning is a change management technique that allows you to store, track and manage different versions of data over time
- Why do we need that:
    - Historical storage: Versioning allows you to store historical versions of records, allowing you to analyze trends and restore previous states of data
    - Audit and Reporting: Supports audit requirements by allowing you to track who made what changes to data, when and what
    - Conflict management: In a multi-user environment, helps resolve conflicts that arise when different users change the same data at the same time.

Versioning

- Conventionally, versioning can be divided into built-in and manual


Versioning
Built-in
Manual
Using internal DBMS tools or extensions
By creating additional structures in the data schema that store the history of changes to a particular record

«Manual» versioning

- One of the manual methods – Slowly Changing Dimensions (term by – *Ralph Kimball, Margy Ross, The Data Warehouse Toolkit*)
- The term comes from the concept of data warehouse design developed by Ralph Kimball (an alternative concept is William Inmon)

«Manual» versioning

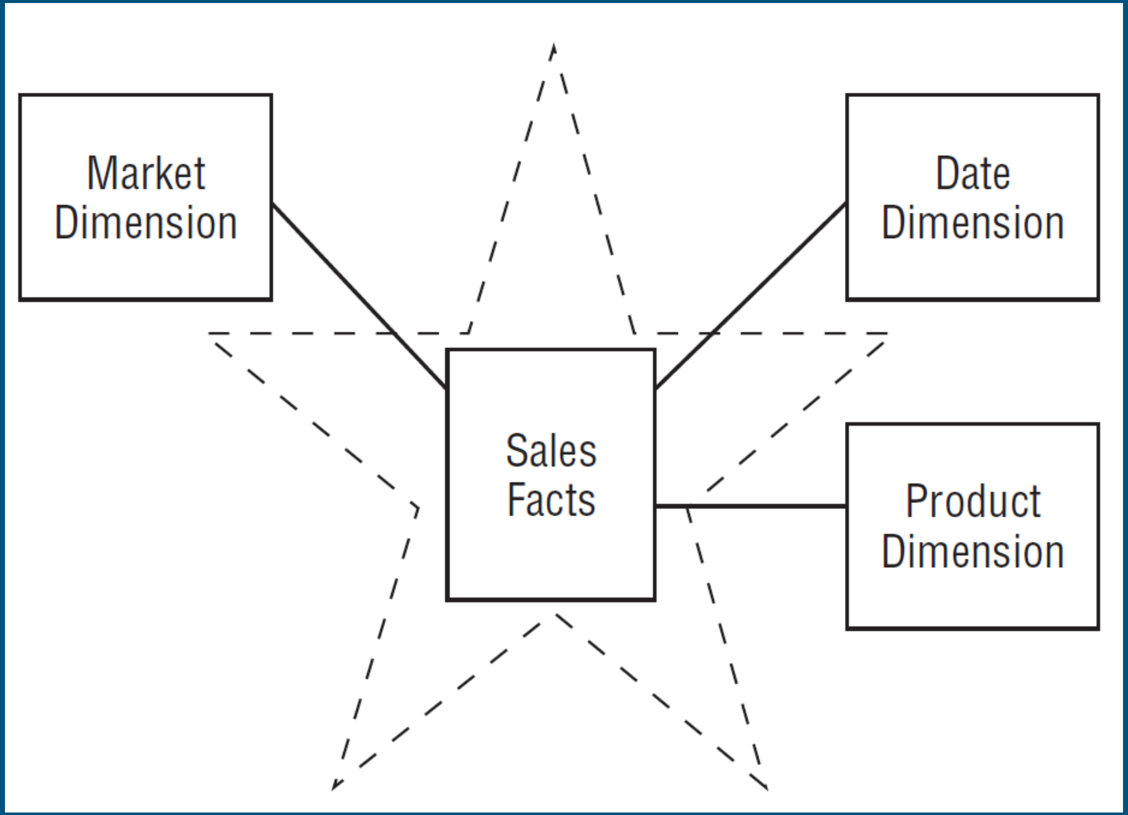
Relationships in the repository are conventionally divided into fact tables and dimension tables.

- Fact tables record data about the facts of the company’s “economic life” (sales, deliveries, acceptance of consignments, etc.)
- Dimension tables record descriptive information about objects or events. Each row in a dimension table represents one entity or one dimension variant. Measurement tables are designed to answer the questions of who, what, where, when, how and why “committed” or participated in a particular fact entered in the fact table)

«Manual» versioning

- Since dimension tables can change at different intervals, and there is also a need to store change history or early data, the technique/concept of Slowly Changing Dimensions was proposed

«Manual» versioning - dimensions



**Ralph Kimball, Margy Ross, The Data Warehouse Toolkit, 2013*

SCD – Slowly Changing Dimensions

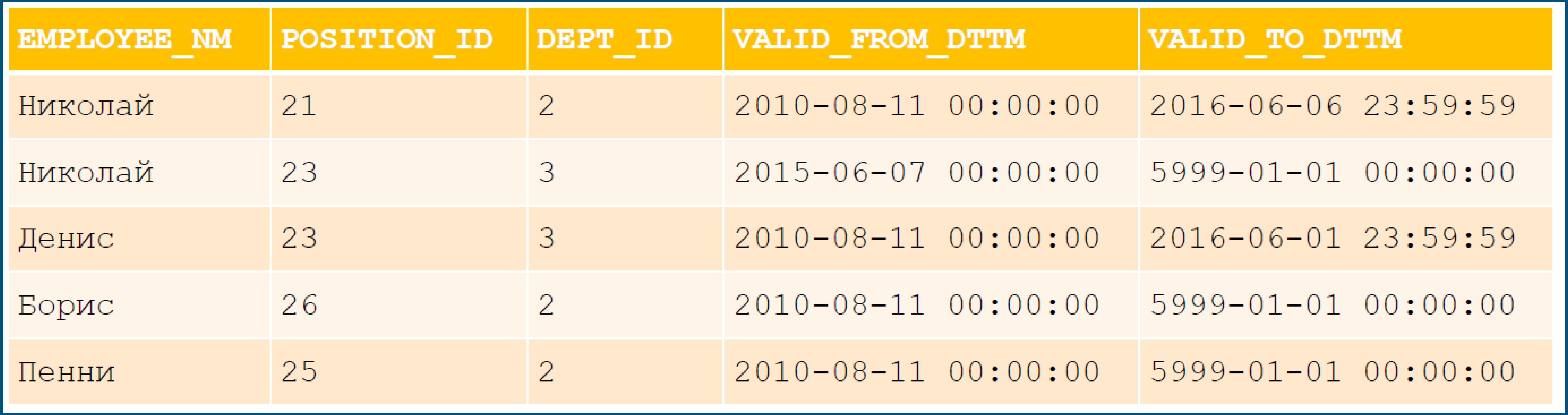
SCD

Type 0 – saving the original	Attribute values do not change
Type 1 - Overwrite	The data is completely replaced with new ones
Type 2 - Adding a new line	The old entry is saved, the new one is simply added in the next line
Type 3 – Adding a new attribute	The old record is saved, new attributes are added that store the previous value
Type 4 – Adding a new relationship	The old record is overwritten, changes are stored in a separate table

SCD – Slowly Changing Dimensions

- Kimball identifies two more types of SCD (5 and 6)
- These types are combinations of the previous ones, and also entail the need to perform additional operations when adding data
- For versioning data in a “classic” DBMS they are less important
- In practice, types 2-4 are most often used (type 2 is most often used)

SCD 2: example



EMPLOYEE_NM	POSITION_ID	DEPT_ID	VALID_FROM_DTTM	VALID_TO_DTTM
Николай	21	2	2010-08-11 00:00:00	2016-06-06 23:59:59
Николай	23	3	2015-06-07 00:00:00	5999-01-01 00:00:00
Денис	23	3	2010-08-11 00:00:00	2016-06-01 23:59:59
Борис	26	2	2010-08-11 00:00:00	5999-01-01 00:00:00
Пенни	25	2	2010-08-11 00:00:00	5999-01-01 00:00:00

Here: creating a new record in the table for each version of the data, adding fields for the start date and end date of the version's existence period

SCD 2: example

- The valid_from_dttm and valid_to_dttm fields generally do not use NULL values. Instead of NULL, some constant is used, for example, ‘59 99 01 01 00:00:00’ for valid_to_dttm , as in the example. This approach simplifies writing conditions:

```
WHERE day_dt BETWEEN valid_from_dttm AND valid_to_dttm
```

instead of

```
WHERE day_dt >= valid_from_dttm
        AND day_dt <= valid_to_dttm
        OR valid _to_dttm IS NULL)
```

SCD 2: characteristics

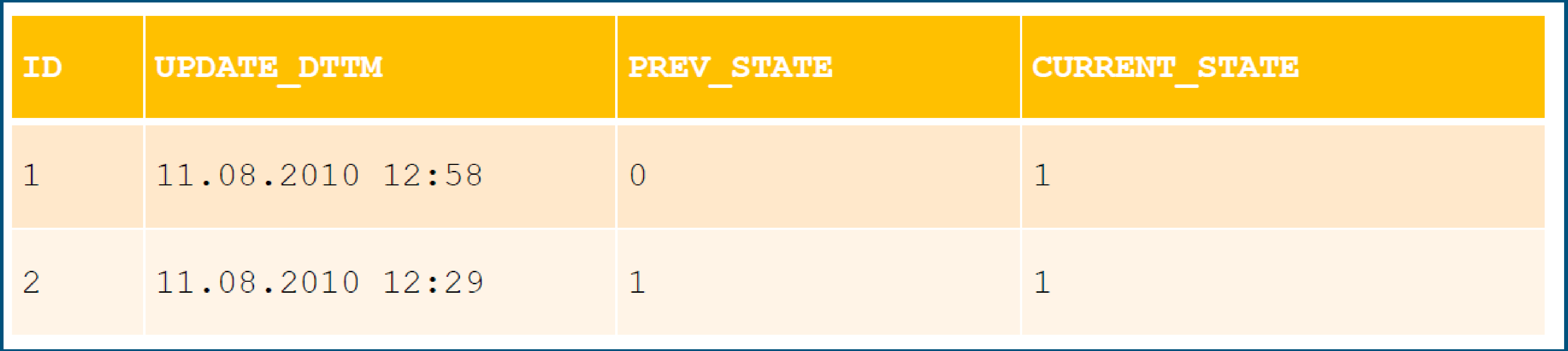
Advantages:

- Stores a complete and unlimited version history
- Convenient and easy access to data of the required period

Disadvantages:

- Provokes redundancy or the creation of additional tables to store changeable attributes

SCD 3: example



ID	UPDATE_DTTM	PREV_STATE	CURRENT_STATE
1	11.08.2010 12:58	0	1
2	11.08.2010 12:29	1	1

Here: The record itself contains additional fields for previous attribute values. When new data is received, the old data is overwritten with the current values.

SCD 3: characteristics

- Advantages:
  - Small amount of data
  - Easy and quick access to history
- Disadvantages:
  - Limited history

SCD 4: example

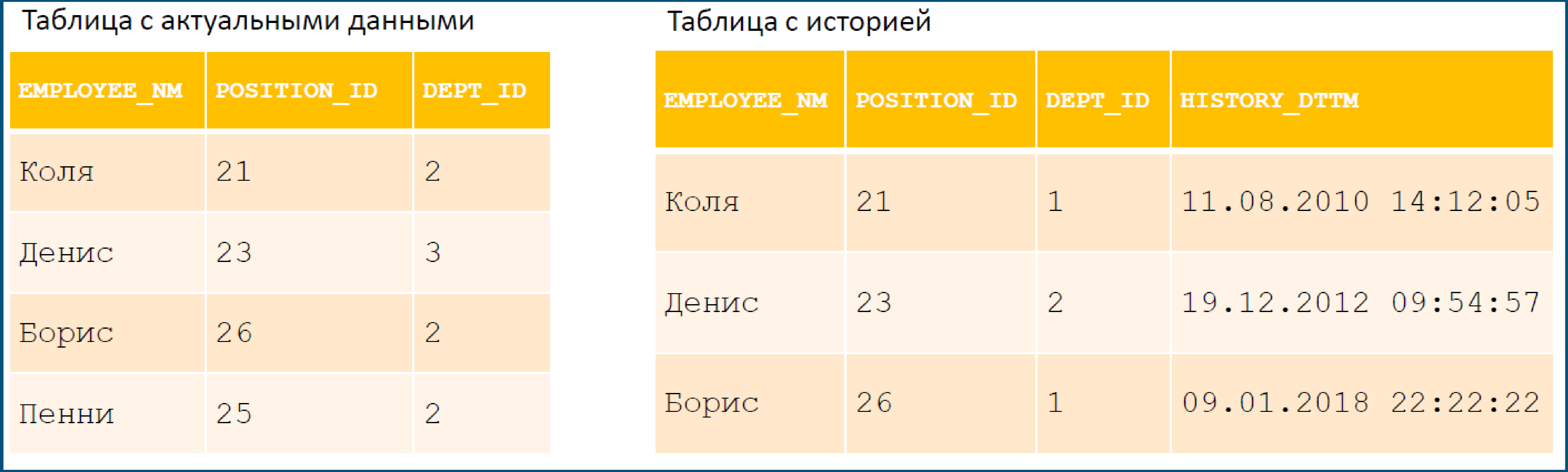


Таблица с актуальными данными

EMPLOYEE_NM	POSITION_ID	DEPT_ID
Коля	21	2
Денис	23	3
Борис	26	2
Пенни	25	2

Таблица с историей

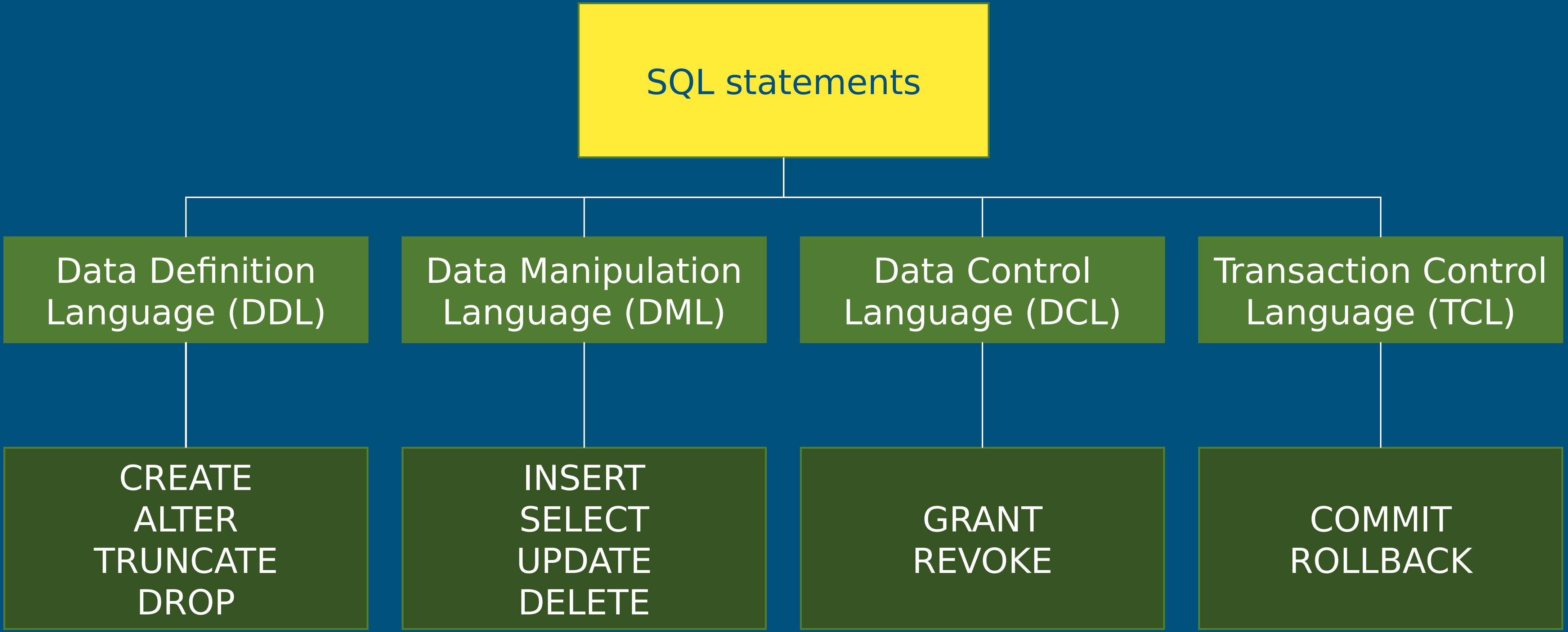
EMPLOYEE_NM	POSITION_ID	DEPT_ID	HISTORY_DTTM
Коля	21	1	11.08.2010 14:12:05
Денис	23	2	19.12.2012 09:54:57
Борис	26	1	09.01.2018 22:22:22

SCD 4: characteristics

- Advantages:
    - Quick work with current versions
- Disadvantages:
    - Splitting a single entity into different tables

III. TCL, ACID, isolation

Groups of SQL operators


SQL statements
Data Definition Language (DDL)
Data Manipulation Language (DML)
Data Control Language (DCL)
Transaction Control Language (TCL)
CREATE ALTER TRUNCATE DROP
INSERT SELECT UPDATE DELETE
GRANT REVOKE
COMMIT ROLLBACK

TRANSACTION

A transaction is a group of sequential operations with a database, which is a logical unit of work with data, guaranteed to transfer the database from one consistent state to another.

Transaction

Transactions ensure the integrity of the database under the following conditions:

- Parallel data processing
- Physical disk failures
- Emergency power failure
- And others...

- Relational DBMS transactions necessarily have four key properties. They are usually abbreviated to ACID.

CAP-theorem

It is impossible for any distributed computing implementation to provide more than two of the following properties:

- Data consistency (Consistency) – in all computing nodes at one point in time, data does not contradict each other;
- Data availability (Availability) – any request to a distributed system ends with a correct response, but without a guarantee that the responses of all system nodes match;
- Partition tolerance – splitting a distributed system into several isolated sections does not lead to incorrect response from each section.

*Maximizing Consistency and Availability generates ACID transaction properties for RDBMS

ACID

- Transactions must satisfy the following requirements:
  - Atomic
    - Either all sub-operations have been completed or none
  - Consistent
    - Each successful transaction records only valid results
  - Isolated
    - Parallel transactions do not affect each other's results
  - Durable
    - Regardless of system failures, the results of successful transactions will be stored in the system

ACID: Atomic

- No transaction will be partially recorded in the system
- Either all sub-operations have been performed, or none
- In practice, simultaneous and atomic execution of transactions is not possible
- In practice, "atomicity" is implemented using "rollback" (rollback)
- During the rollback process, all operations that have already been performed are canceled

ACID: Consistent

Once the transaction is complete, all data should remain in a consistent state. When executing a transaction, all rules of the relational DBMS must be followed:

- Constraint compliance checks (domains, unique indexes, foreign keys, checks, rules, etc.)
- Index update
- Executing Triggers
- And others

But: consistency is not required during the operation (due to atomicity, intermediate inconsistency remains hidden)

ACID: Isolated

Changes to data made within a transaction must be isolated from all changes made in other transactions until the transaction is committed. There are different isolation levels to achieve a compromise between the degree of parallelization of work with the database and the strictness of the implementation of the principle of consistency:

- The higher the isolation level, the higher the degree of data consistency;
- The higher the isolation level, the lower the degree of parallelization and the lower the degree of data availability.

In real databases, complete isolation is not supported!

ACID: Durable

- If a transaction has been made, its result must be stored in the system, despite the system failure. That is, if the user has received confirmation of the success of the transaction, the safety of the results is guaranteed
- If a transaction was not completed, its result may be completely canceled due to a system failure.

TCL (TRANSACTION CONTROL LANGUAGE)

COMMIT

- Applies the transaction, i.e. saves the changes made during the execution of the transaction

ROLLBACK

- Rolls back all changes made during the execution of the transaction

SAVEPOINT

- Creates a so-called breakpoint

BREAKPOINT

A breakpoint is an intermediate section in a transaction to which you can roll back if necessary.

- Allows you to split a transaction into parts
- Allows you to implement "nested" transactions

Syntax

- BEGIN TRANSACTION transaction_mode [,
  - ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ
  - COMMITTED | READ UNCOMMITTED }
  - READ WRITE | READ ONLY
  - [NOT] DEFERRABLE
- BEGIN / START
  - COMMIT
  - ROLLBACK
- SAVEPOINT name
  - ROLLBACK TO SAVEPOINT name
  - RELEASE SAVEPOINT name

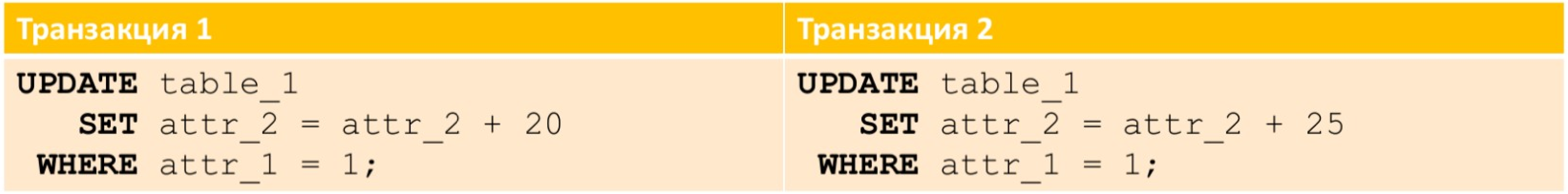
Example

```
BEGIN TRANSACTION ISOLATION LEVEL READ
COMMITTED;
        INSERT
        INTO table1 VALUES (1);
        SAVEPOINT my_savepoint
        INSERT INTO table1 VALUES (2);
        ROLLBACK TO SAVEPOINT my_savepoint
        INSERT INTO table1 VALUES (3);
COMMIT
```

Isolation problems

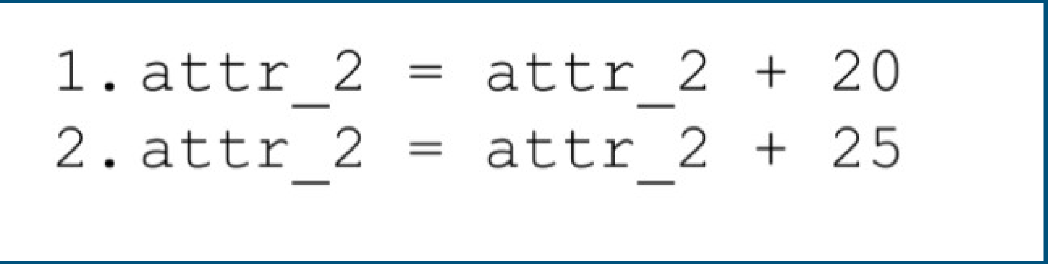
Lost update	Changing one block of data by multiple transactions
"Dirty" reading	Reading data modified by a rolled back transaction
Not repeating reading	Repeated reading of changed data by the same transaction
Reading phantoms	If a transaction repeats a query with some condition, and another transaction adds or removes rows that satisfy that condition, the resulting row set will be different

LOST UPDATE



Транзакция 1	Транзакция 2

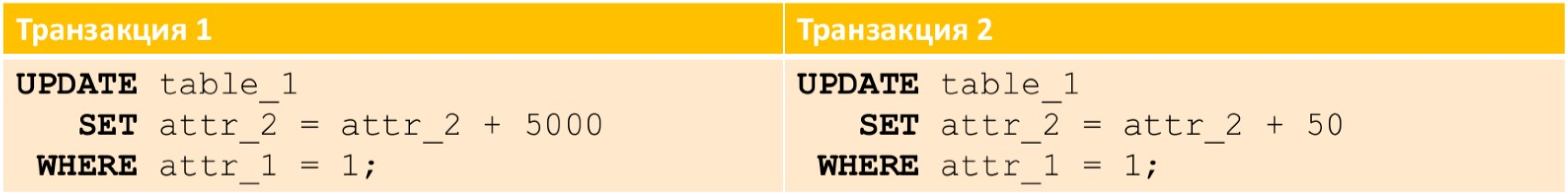
`UPDATE table_1`
`   SET attr_2 = attr_2 + 20`
` WHERE attr_1 = 1;`
	`UPDATE table_1`
	`   SET attr_2 = attr_2 + 25`
	` WHERE attr_1 = 1;`



```
1. attr_2 = attr_2 + 20
2. attr_2 = attr_2 + 25
```

The result is not unambiguously defined

LOST UPDATE



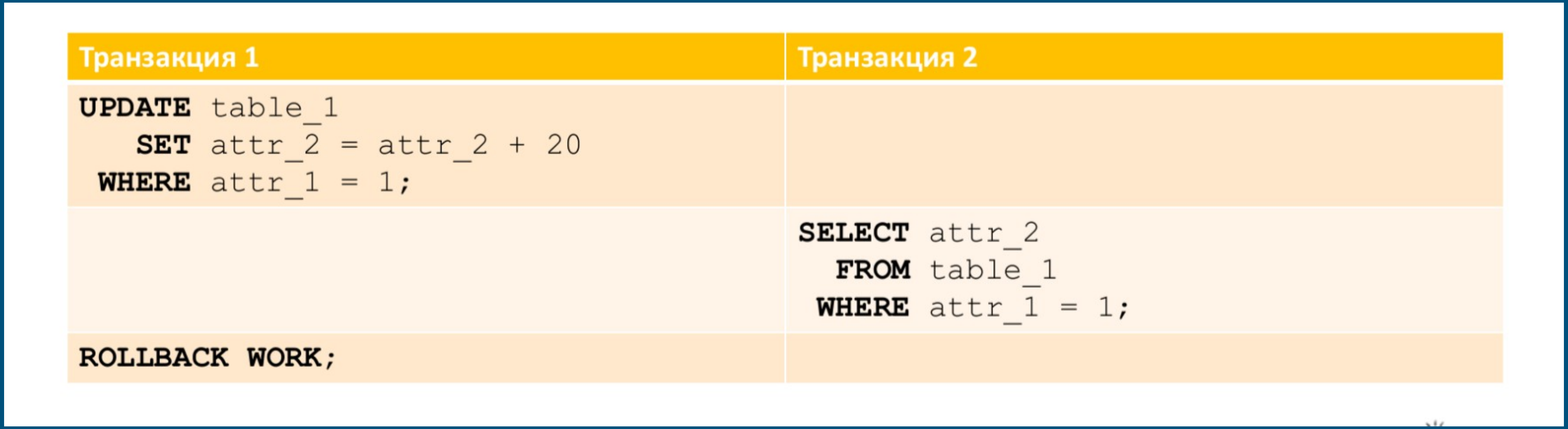
Транзакция 1	Транзакция 2

`UPDATE table_1`
`   SET attr_2 = attr_2 + 5000`
` WHERE attr_1 = 1;`
	`UPDATE table_1`
	`   SET attr_2 = attr_2 + 50`
	` WHERE attr_1 = 1;`

- You top up your card for 5000 rubles
- Someone transfers you 50 rubles

The total amount on your account is unknown

"DIRTY" READING



Транзакция 1	Транзакция 2

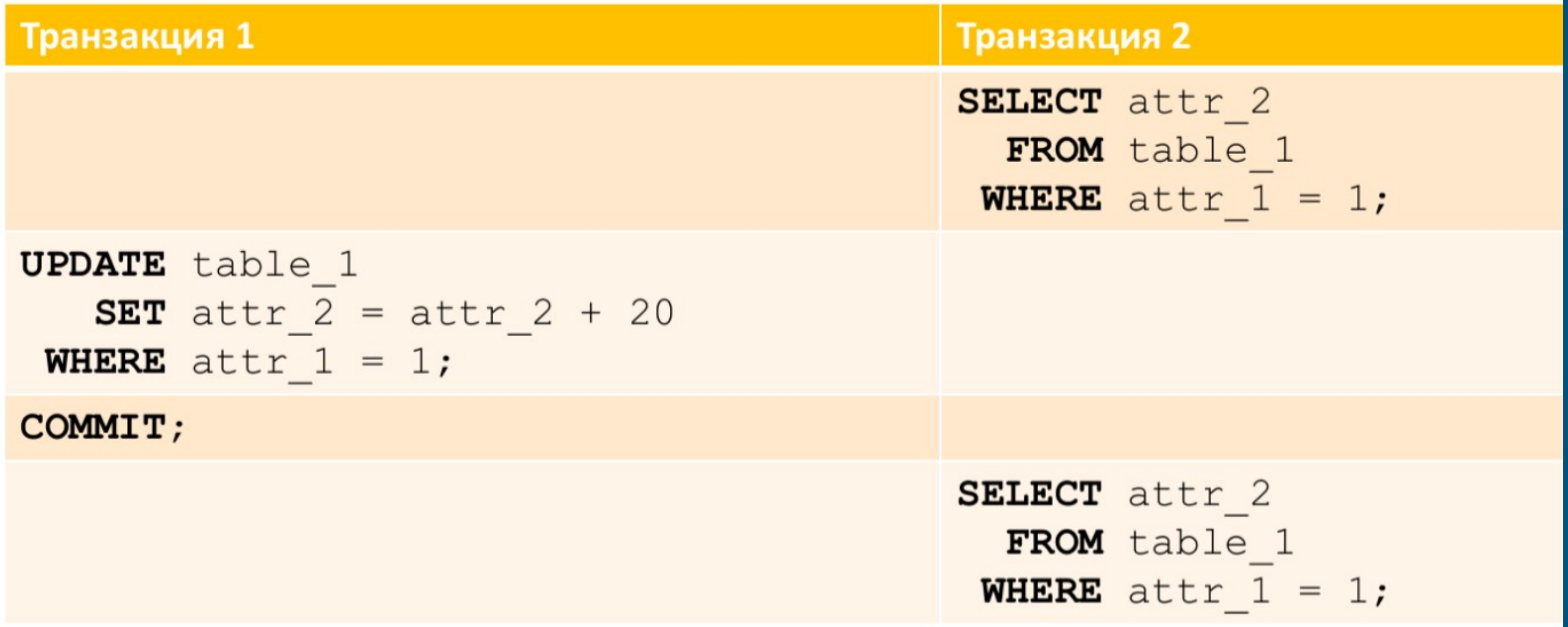
`UPDATE table_1`
`   SET attr_2 = attr_2 + 20`
` WHERE attr_1 = 1;`	

	`SELECT attr_2`
	`  FROM table_1`
	` WHERE attr_1 = 1;`

`ROLLBACK WORK;`	

You are paying for an expensive purchase from a relative's account
Your relative checks the remainder -> falls into a stupor
....
The write-off does not pass, the transaction is rolled back, but the relative is already in shock

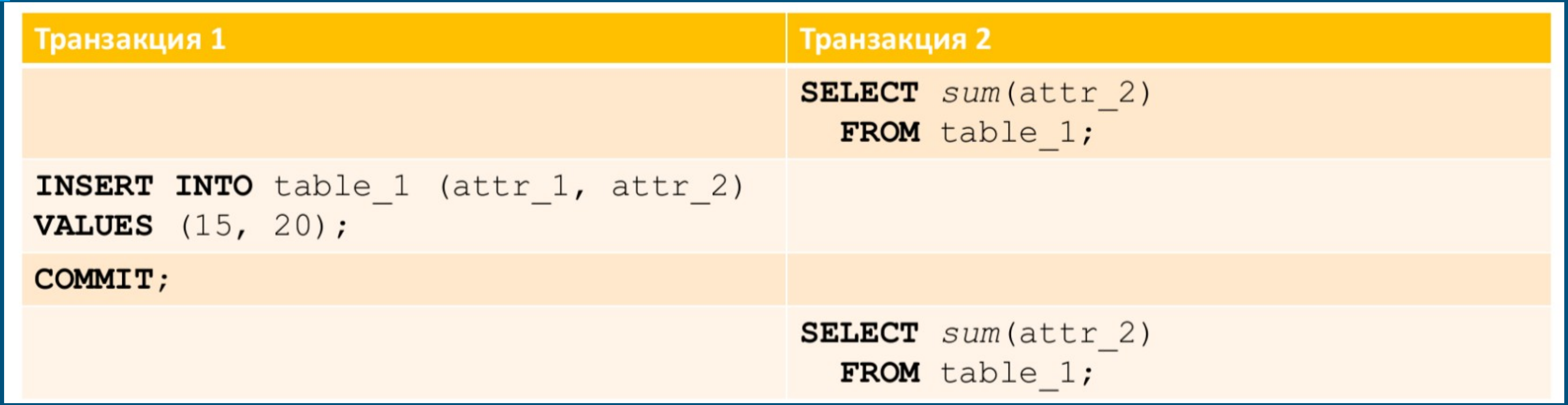
NON-REPEATING READING



Транзакция 1	Транзакция 2
	`SELECT attr_2 FROM table_1 WHERE attr_1 = 1;`
`UPDATE table_1 SET attr_2 = attr_2 + 20 WHERE attr_1 = 1;`	
`COMMIT;`	
	`SELECT attr_2 FROM table_1 WHERE attr_1 = 1;`

You have created a shopping cart on the website and click on the "Pay" button
The cost of goods on the site is increasing
A larger amount is debited from the card than you expected

READING PHANTOMS



Транзакция 1	Транзакция 2

	`SELECT sum(attr_2)`
	`  FROM table_1;`

`INSERT INTO table_1 (attr_1, attr_2)`
`VALUES (15, 20);`	

`COMMIT;`	

	`SELECT sum(attr_2)`
	`  FROM table_1;`

The boss asked you to make a general spending report based on the results of the current month and in the context of the weeks of the same month
You calculate the total amount of transactions per month
A new write-off is taking place
You calculate the amount of transactions in the context of weeks → take into account a new transaction –> the amounts do not beat

TRANSACTION ISOLATION LEVELS

- Read uncommitted (read uncommitted data)
- Read committed (reading fixed data)
- Repeatable read (read repeatability)
- Serializable (orderability)

READ UNCOMMITTED

- First level of isolation
- Ensures no lost updates
- The final value is the result of each transaction
- It is possible to read unfixed changes
- Data is blocked for the time of making changes
- There is no lock during data reading

Read uncommitted

- In PostgreSQL, you can query for any of the four transaction isolation levels, but only three different levels are implemented internally, meaning the Read Uncommitted mode in PostgreSQL acts like Read Committed. The reason for this is that it is the only way to map standard isolation levels to PostgreSQL's Multi-Version Concurrency Control (MVCC) architecture.

READ COMMITTED

- Second level of isolation
- Used in most DBMS
- Protection against dirty reading
- In the process of execution, one of the transactions

completes successfully, then the rest work with the changed data

- Implementation of IRL at the discretion of DBMS developers:
    - Blocking of readable and modifiable data
    - Saving multiple versions of lines being modified in parallel

Repeatable read

- The reading transaction ignores changes to data that were previously read to it
- No transaction can modify data read by the current transaction until the read is complete
- Saves from the effect of non-repeating reading
- This level differs from Read Committed in that a request in a transaction at this level sees a snapshot of the data at the start of the first statement in the transaction (not counting transaction control commands), and not the start of the current statement. Thus, for example, successive SELECT commands in the same transaction see the same data; they do not see changes made and committed by other transactions after their current transaction began.

SERIALIZABLE

- The fourth (highest) level of isolation
- Transactions are completely isolated from each other
- Parallel transactions do not seem to exist at all
- Transactions are not subject to the effect of "phantom reading"

Serializable

- This level models the sequential execution of all committed transactions, as if the transactions were executed one after the other, sequentially rather than in parallel. In fact, this isolation mode works in the same way as Repeatable Read, only it additionally monitors for conditions under which the result of serializable transactions executed in parallel may not be consistent with the result of the same transactions executed in turn.

TRANSACTION ISOLATION LEVELS

Isolation level	Lost update	“Dirty” reading	Non-repeating reading	Reading phantoms
Read uncommited	+	-	-	-
Read committed	+	+	-	-
Repeatable read	+	+	+	-
Serializable	+	+	+	+

“+” – prevented
“-” – not prevented

TRANSACTION ISOLATION LEVELS Postgres

Isolation level	“Dirty” reading	Non-repeating reading	Reading phantoms	Serialization anomalies
Read uncommited	Possible, but not in Postgres	Possible	Possible	Possible
Read committed	Impossible	Possible	Possible	Possible
Repeatable read	Impossible	Impossible	Possible, but not in Postgres	Possible
Serializable	Impossible	Impossible	Impossible	Impossible

- “Serialization anomaly” - the result of a successful commit of a group of transactions turns out to be inconsistent under all possible options for executing these transactions in turn (that is, unlike a lost update, there will be no result at all)