Ограничения

Files & media

▼ Целостность данных.

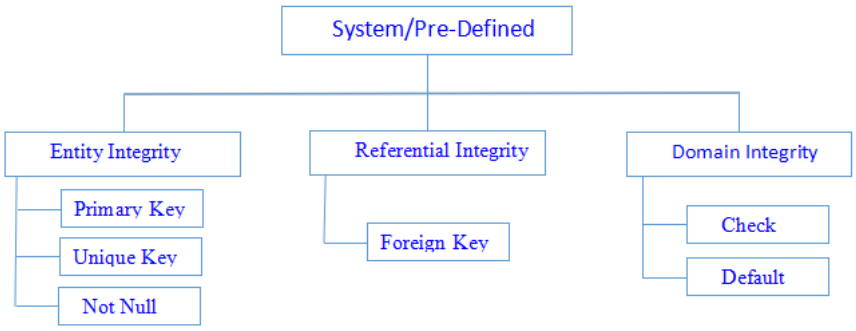
Обеспечение целостности данных гарантирует **качество данных** в базе данных, с точки зрения бизнеса это **очень** важно и гораздо важнее, чем производительность.

При изменении содержимого базы данных с помощью инструкций `INSERT`, `DELETE` или `UPDATE` может произойти нарушение целостности содержащихся в ней данных. Чтобы информация, хранящаяся в базе данных, была **однозначной и непротиворечивой,** в реляционной модели устанавливаются некоторые **ограничительные условия** – правила, определяющие возможные значения данных и обеспечивающие **логическую основу для поддержания корректных значений**.

Типы целостности данных

- **Декларативная:** ограничения целостности данных задаются **как часть схемы базы данных с помощью декларативных языковых конструкций**, таких как ограничения SQL (просто, автоматизировано). **Пример:** Primary Key, UNIQUE, Foreign Key, DEFAULT, CHECK.
- **Процедурная:** обеспечение соблюдения ограничений целостности данных с помощью **процедурной или программной логики (**написание пользовательского кода). Дает нам гибкость (пишем то, что нельзя сделать с помощью декларативных ограничений) и контроль (контролируем процесс сами). **Пример:** Скрипты на PL/SQL или T-SQL, Триггеры, Функции

Категории целостности данных



Ссылочная целостность (Referential Integrity)

Обеспечивает согласованность отношений **между таблицами** в базе данных. Гарантирует, что значения **внешних ключей** в одной таблице совпадают со значениями **первичных ключей в другой**, предотвращая появление "бесхозных" или "висячих" ссылок. Ссылочная целостность поддерживается с помощью `FOREIGN KEY` constraints.

Целостность Объекта (Entity Integrity)

Гарантирует, что каждая строка в таблице является **уникальным идентифицируемым объектом**. Обеспечивается `PRIMARY KEY`, `UNIQUE`, `NOT NULL`**.** Это предотвращает появление дубликатов или нулевых значений в столбце **первичного ключа, который обычно и позволяет нам уникально индефицировать объект**.

Целостность домена (Domain Integrity)

О**беспечивает правильные и допустимые значения, формат и диапазон для каждого столбца** или атрибута в таблице базы данных. Достигается за счет использования ограничений `CHECK` и `DEFAULT`.

Целостность, определяемая пользователем (User – Defined Integrity)

Специфические бизнес-правила и ограничения, которые не охватываются предыдущими типами целостности, но имеют решающее значение для обеспечения согласованности данных. Эти правила обычно определяются пользователями или разработчиками приложений с помощью **триггеров, хранимых процедур или пользовательского кода.**

Пример: Реализация правила, которое рассчитывает и применяет максимальное количество заказа на основе имеющегося уровня запасов.

▼ Ограничения

Ограничения SQL — это правила, применяемые к **столбцам** (одному, нескольким или всей таблице) данных таблицы для ограничения их типа. Это обеспечивает **целостность базы данных.**

Виды ограничений

▼ **NOT NULL Constraint** — столбец не может иметь значение `NULL`.

Если мы не уверены, что данные, которые мы вставляем, всегда будут без пропусков, то лучше в совокупности использовать это ограничение с `DEFAULT`

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255) NOT NULL,
    Age int
);

ALTER TABLE table_name
ALTER COLUMN column_name SET NOT NULL;
```

▼ **DEFAULT Constraint** — задает значение по умолчанию для столбца, если оно не указано (вместо NULL).

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    City varchar(255) DEFAULT 'Sandnes'
);

ALTER TABLE table_name
ALTER COLUMN column_name SET DEFAULT 1;
```

▼ **UNIQUE Constraint** — все значения в столбце должны быть уникальными.

Может содержать `NULL`, но только один раз. Можно применять к нескольким столбцам, тогда их совокупность должна быть уникальна (сочетание a1 + a2)

```
CREATE TABLE Persons (
    ID int NOT NULL UNIQUE,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int
);

ALTER TABLE table_name
ADD CONSTRAINT constraint_name UNIQUE (column1, column2, ... column_n);
```

▼ **PRIMARY KEY** — уникально идентифицирует каждую **строку** в таблице базы данных.

Должен содержать значения **UNIQUE** и имеет неявное (можно не писать) ограничение `NOT NULL`. **`NOT NULL` должно быть у каждого столбца (если первичный ключ составной).**

> **Сколько может быть PRIMARY KEY?**
> Таблица в SQL строго ограничена наличием **одного** и **только одного** первичного ключа, состоящего из одного или нескольких полей (столбцов). Добавление первичного ключа автоматически создаст уникальный индекс B-дерева на столбец или группу столбцов, перечисленных в первичном ключе, и заставит столбец (столбцы) быть помеченным `NOT NULL`.

Виды PRIMARY KEY:

- **Natural Primary Key.** Используется один или несколько столбцов из реальных данных таблицы,
  - **Плюсы:** имеют ценность для бизнес-пользователя, скорее всего будут более уникальные и при изменении целостность не нарушится (например, номер паспорта гражданина РФ).
  - **Минусы:** обычно длиннее и более сложные, чем суррогатные ключи (например, ФИО + дата рождения). Также такие ключи чаще содержат чувствительную информацию, которую необходимо защищать. Наконец, если такие ключи основаны на информации, которая может быть изменена (ФИО, например), то их будет необходимо обновлять.
  - **Когда используем:** в данных имеются **значимые, стабильные и простые идентификаторы**, а также **отсутствуют серьезные проблемы с конфиденциальностью и безопасностью**. Они могут быть хорошим выбором для небольших, четко определенных наборов данных.
- **Surrogate Primary Key.** Автогенерируем бессмысленные ключ, который однозначно будет определять строку, не содежит никакой полезной информации (например, UUID, который может сделать наша БД самостоятельно)
  - **Плюсы:** простые и короткие (часто один столбец), неизменяемые (и потому надежные), хороши для индексирования из-за того, что коротенькие.
  - **Минусы:** не несут никакой бизнес-информации и занимают доп. место. По сути, мы тратим память на хранение дополнительной колонки вместо того, чтобы выбрать уже существующую.
  - **Когда используем:** в больших базах данных, где важны производительность, стабильность и простота. Они также полезны при работе с конфиденциальными данными или в тех случаях, когда естественные ключи могут изменяться.

▼ **FOREIGN Key** — уникально идентифицирует **строку** в любой **другой** таблице базы данных.

Состоит из одного или нескольких столбцов, которые **ссылаются** на **PRIMARY KEY** в другой таблице. Ограничение внешнего ключа обеспечивает **ссылочную целостность** в отношениях между двумя таблицами.

Таблица может иметь **более одного** ограничения внешнего ключа. Это используется для реализации отношений "**many-to-many**" между таблицами

Таблица с **внешним ключом** называется **дочерней**, а с **первичным** — **родительской**. **При вставке строки в дочернюю таблицу проверяется наличие значения внешнего ключа в родительской таблице. Если его нет, то ключ не добавится.**

```
SQL Exception: Integrity constraint violation: FOREIGN KEY constraint "FK_Orders_Customers" violation. The "customer_id" value
```

> **Обрати внимание!**
>
> столбец `FOREIGN KEY` может содержать как и `NULL` значения (например, нам пока не известно, кто возьмет заказ), так и дубликаты (например, сотрудник может отвечать за сборку нескольких заказов).

Обновление и удаление из родительской таблицы

С помощью выражений `ON DELETE` и `ON UPDATE` можно установить действия, которые выполняться соответственно при удалении и изменении связанной строки из родительской таблицы. И для определения действия мы можем использовать следующие опции:

- `CASCADE`: автоматически удаляет или изменяет строки из дочерней таблицы при удалении или изменении связанных строк в родительской таблице.
- `NO ACTION` или `RESTRICT`: ничего не произойдет. Различия минорные, зависят от БД (+ доп. информация в *Отложенность проверки ограничений*, т.к. NO ACTION - отложенная проверка)
- `SET NULL`: при удалении связанной строки из родительской таблицы устанавливает для столбца внешнего ключа значение NULL.
- `SET DEFAULT`: при удалении связанной строки из родительской таблицы устанавливает для столбца внешнего ключа значение **по умолчанию**, которое задается с помощью ограничения `DEFAULT`. Если для столбца не задано значение по умолчанию, то в качестве него применяется значение `NULL`.

Если мы явно не указываем `ON DELETE` и `ON UPDATE`, то они работают как `RESTRICT`**: мы не можем удалить ничего из главной таблице или обновить, пока сначала не сделаем это в наших дочерних таблицах.**

```
CREATE TABLE Orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    -- Define the foreign key constraint with ON DELETE CASCADE
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id) ON DELETE CASCADE
);
```

▼ **CHECK Constraint** — ограничение CHECK обеспечивает, чтобы все значения в столбце удовлетворяли определенным условиям.

```
CREATE TABLE Persons (
    ID int NOT NULL,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int CHECK (Age>=18)
);
```

▼ **INDEX** — используется для быстрого поиска по таблице в Базе Данных.

Подробнее см Индексы.

▼ *Создание и удаление ограничений*

Можно сделать при создании таблицы или через `ALTER TABLE` (команды `DROP`, `ADD`)

```
--добавляем ограничение при создании таблицы
CREATE TABLE order_details
( order_detail_id integer CONSTRAINT order_details_pk PRIMARY KEY,
  order_id integer NOT NULL,
  order_date date,
  quantity integer,
  notes varchar(200),
  CONSTRAINT order_date_unique UNIQUE (order_id, order_date)
);

--создание ограничения через ALTER TABLE
ALTER TABLE table_name ADD CONSTRAINT constraint_name UNIQUE (column1, column2, ... column_n);

--удаление ограничения
ALTER TABLE table_name
DROP CONSTRAINT constraint_name;
```

▼ Отложенность проверки ограничений (Транзакции)

Ограничения на столбцы могут быть как отложенными (`DEFERRED`), так и немедленными (`IMMEDIATE`). Немедленные ограничения проверяются в **конце каждого оператора (statement)**, в то время как отложенные ограничения не проверяются до фиксации (**commit**) транзакции. Каждое ограничение имеет свой собственный режим IMMEDIATE или DEFERRED.

Есть несколько градаций, в ходе которых будет выполнена проверка ограничений. Для PostgreSQL это:

- **По строкам (By Row)**. В этом случае оператор, обновляющий несколько строк, будет немедленно, возможно, даже частично, завершен, **если одна из строк нарушит ограничение**.
- **По оператору (By Statement).** В этом случае оператору разрешается выполнить весь набор изменений, прежде чем база данных проверит его на наличие нарушений.
- **По транзакции (By Transaction).** Любой оператор внутри транзакции может нарушать ограничения. Однако во время **фиксации (операция commit)** ограничения будут проверены, и транзакция завершится неудачей, если какие-либо ограничения не будут выполнены.

Таблица и информация актуальная для PostgresSQL, которая может не соот. другим БД

<u>При создании ограничению присваивается одна из трех характеристик:</u>

- `NOT DEFERRABLE` (по умолчанию, эквивалентно immediate): ограничение проверяется сразу после каждого оператора (statement). Это поведение **НЕ может быть изменено** с помощью команды `SET CONSTRAINT`.

- `INITIALLY IMMEDIATE`: ограничение проверяется сразу после каждого утверждения, однако впоследствии это поведение может быть изменено с помощью команды `SET CONSTRAINT`.
- `INITIALLY DEFERRED`: ограничения не проверяются до фиксации транзакции. В дальнейшем это поведение может быть изменено с помощью команды set constraint.

По умолчанию, вид “Отложенности” для каждого из ограничений ниже является **NOT DEFERRABLE** и показывает, где (на каком этапе) будет произведена проверка:

	NOT DEFERRABLE	INITIALLY IMMEDIATE	INITIALLY DEFERRED
CHECK	row	row	row
NOT NULL	row	row	row
UNIQUE	row	statement	transaction
PRIMARY KEY	row	statement	transaction
FOREIGN KEY	statement	statement	transaction
EXCLUDE	statement	statement	transaction

```
ALTER TABLE foo
  ADD CONSTRAINT foo_bar_fk
  FOREIGN KEY (bar_id) REFERENCES bar (id)
  DEFERRABLE INITIALLY IMMEDIATE; -- the magic line

--А также мы можем работать с транзакциями и выполнение внутри
BEGIN;
-- defer the constraint
SET CONSTRAINTS foo_bar_fk DEFERRED;

-- ...
-- now we can do whatever we want to bar_id
-- ...

COMMIT; -- but it better be correct at this point
```

Без явной команды `BEGIN` каждый оператор выполняется в своей собственной однотранзакционной транзакции, где нет разницы между `INITIALLY IMMEDIATE` или `DEFERRED`. Попытка отложить (defer) ограничения вне транзакции ничего не даст и приведет к предупреждению

```
WARNING: 25P01: SET CONSTRAINTS can be used only in transaction blocks
```

Когда используем отложенное исполнение?

▼ **[Cyclic Foreign Keys]**

Классический пример — создание двух элементов, связанных циклическими проверками согласованности.

```
CREATE TABLE husbands (
  id int PRIMARY KEY,
  wife_id int NOT NULL
);

CREATE TABLE wives (
  id int PRIMARY KEY,
  husband_id int NOT NULL
);

ALTER TABLE husbands ADD CONSTRAINT h_w_fk
  FOREIGN KEY (wife_id) REFERENCES Wives;

ALTER TABLE wives ADD CONSTRAINT w_h_fk
  FOREIGN KEY (husband_id) REFERENCES husbands;
```

Нам невозможно вставить строку ни в одну из таблиц, так как как только мы захотим добавить командой жену/мужа, проверяя операторы по одному, а нам для успешной вставки нужно вставить сразу два стейтмента.

Решение: поместим две вставки внутри транзакции, отложив проверку ограничения до коммита

```
ALTER TABLE husbands ALTER CONSTRAINT h_w_fk
  DEFERRABLE INITIALLY DEFERRED;

ALTER TABLE wives ALTER CONSTRAINT w_h_fk
  DEFERRABLE INITIALLY DEFERRED;

--Начинаем транзакцию
BEGIN;
INSERT INTO husbands (id, wife_id) values (1, 1);
INSERT INTO wives (id, husband_id) values (1, 1);
COMMIT;
-- and they lived happily ever after
```

▼ **[Reallocating Items, One per Group]**

В этом примере у нас есть набор школьных классов, в каждом из которых назначен ровно один учитель. Предположим, мы хотели бы поменять учителей двух классов. Ограничения усложняют задачу.

```
CREATE TABLE classes (
  id int PRIMARY KEY,
  teacher_id int UNIQUE NOT NULL
);

INSERT INTO classes VALUES (1, 1), (2, 2);
```

Трюк с CTE не сработает для замены учителей, поскольку UNIQUE NOT DEFERRED проверяется по row, а не по statement.

```
-- fails unless deferrable since PostgreSQL
-- checks per row, not per statement
WITH swap AS (
    UPDATE classes
       SET teacher_id = 2
     WHERE id = 1
  )
UPDATE classes
   SET teacher_id = 1
 WHERE id = 2;

ERROR:  23505: duplicate key value violates unique constraint "classes_teacher_id_key"
DETAIL:  Key (teacher_id)=(1) already exists.
```

Чтобы не свапать учителей через третью переменную и не придумывать костыли, идеально будет просто задизайнить таблицу, подкорректировав ограничение:

```
CREATE TABLE classes (
  id int PRIMARY KEY,
  teacher_id int NOT NULL UNIQUE
    DEFERRABLE INITIALLY IMMEDIATE
);

BEGIN;
SET CONSTRAINTS classes_teacher_id_key DEFERRED;

UPDATE classes SET teacher_id = 1 WHERE id = 2;
UPDATE classes SET teacher_id = 2 WHERE id = 1;
COMMIT;
```

Кроме того, теперь будет работать подход CTE без явных транзакций, поскольку отложенное ограничение уникальности переключается на проверку каждого оператора.

▼ **[Data Ingestion]**

Отсрочка ограничений при загрузке данных может сделать процесс более удобным. Например, отложенные внешние ключи позволяют загружать таблицы в любом порядке: дочерние перед родительскими. Это позволяет обрабатывать данные, поступающие не по порядку, возможно, из сети.

Не ускоряет загрузку!
Во время фиксации одинаковое количество элементов должно быть проверено на согласованность, и время запроса выравнивается, поэтому то, что отложенное ограничение может хоть как-то ускорить загрузку данных - это миф.
Единственный способ ускорить загрузку — временно отключить *внешний* ключ (при условии, что мы заранее знаем, что загружаемые данные верны). В целом **это плохая идея**, поскольку в конечном итоге нам может потребоваться больше времени на отладку несогласованности в дальнейшем, чем на проверку согласованности во время приема.

Когда не использовать?

(Штраф за производительность планировщика запросов). Планировщик запросов будет использовать оптимизацию только тогда, когда ограничения базы данных **гарантируют правильность**. По определению, отложенные ограничения не относятся к этой категории, поэтому они мешают планировщику.
Пример: Планировщик может определить, что набор условий в таблице обеспечивает уникальность результата (ограничение UNIQUE). Если существует unique, non-deferrable index, то планировщик может исключить необходимый в противном случае шаг sort/unique или хеширования. Он не может сделать это для отложенных ограничений, потому что могут присутствовать фактические повторяющиеся значения.

```
CREATE TABLE foo (
  a integer UNIQUE,
  b integer UNIQUE DEFERRABLE
);

EXPLAIN
SELECT t1.*
FROM foo t1
LEFT JOIN foo t2
  ON (t1.a = t2.a);

QUERY PLAN
------------------------------------------------------------
 Seq Scan on foo t1  (cost=0.00..32.60 rows=2260 width=8)
```

```
---------------------------ОТЛОЖЕННОЕ ОГРАНИЧЕНИЕ--------------------
EXPLAIN
SELECT t1.*
FROM foo t1
LEFT JOIN foo t2
  ON (t1.b = t2.b);
QUERY PLAN
------------------------------------------------------------------------
 Hash Left Join  (cost=60.85..124.53 rows=2260 width=8)
   Hash Cond: (t1.b = t2.b)
   -> Seq Scan on foo t1  (cost=0.00..32.60 rows=2260 width=8)
   -> Hash  (cost=32.60..32.60 rows=2260 width=4)
       -> Seq Scan on foo t2  (cost=0.00..32.60 rows=2260 width=4)
```

Другой пример: в PostgreSQL есть оптимизация, которая превращает semi-join подзапроса IN в обычный JOIN, если столбец для подзапроса с ограничением UNIQUE (т.е. уникален). Отложенное ограничение блокирует эту оптимизацию

```
EXPLAIN
SELECT *
FROM foo
WHERE a IN (
  SELECT a FROM foo
);

QUERY PLAN
------------------------------------------------------------------------
 Hash Join  (cost=60.85..121.97 rows=2260 width=8)
   Hash Cond: (foo.a = foo_1.a)
   ->  Seq Scan on foo  (cost=0.00..32.60 rows=2260 width=8)
   ->  Hash  (cost=32.60..32.60 rows=2260 width=4)
         ->  Seq Scan on foo foo_1  (cost=0.00..32.60 rows=2260 width=4)
```

```
---------------------------ОТЛОЖЕННОЕ ОГРАНИЧЕНИЕ--------------------
EXPLAIN
```

```
SELECT *
FROM foo
WHERE b IN (
  SELECT b FROM foo
);
QUERY PLAN
------------------------------------------------------------------------
 Hash Semi Join  (cost=60.85..124.53 rows=2260 width=8)
   Hash Cond: (foo.b = foo_1.b)
   ->  Seq Scan on foo  (cost=0.00..32.60 rows=2260 width=8)
   ->  Hash  (cost=32.60..32.60 rows=2260 width=4)
         ->  Seq Scan on foo foo_1  (cost=0.00..32.60 rows=2260 width=4)
```

Также для того чтобы указать **внешний ключ** на столбец с ограничением уникальности или первичного ключа, это ограничение должно быть **non-deferrable**, иначе база данных выдаст ошибку

▼ Вопросы по теме “Ограничения”

1. Что такое целостность данных? Зачем она нужна?
2. Какие типы ограничений целостности данных есть? Приведи примеры.
3. Какие категории целостности данных есть? Какие способы в SQL под каждый вид есть?
4. Что такое ограничения в SQL?
5. Сколько основных ограничений существуют?
6. Рассказать про одно из ограничений (на выбор)
7. Сколько может быть PRIMARY KEY? Сколько может быть FOREIGN KEY
8. Что будет, если использовать ограничение UNIQUE на совокупность столбцов?
9. Какие правила накладываются на PRIMARY KEY?
10. Что такое Натуральные и Суррогатные первичные ключи?
11. В каких случаях мы используем суррогатный ключ, а в каких - натуральный.
12. В чем разница между PRIMARY KEY и ограничением UNIQUE? В каких случаях можно использовать одно из них вместо другого?
13. Что такое FOREIGN KEY? Какие у него ограничения?
14. Что будет, если мы попытается добавить внешний ключ, значения которого не существует в родительской таблице?
15. Какие настройки мы можем сделать, если хотим удалить или обновить строки из родительской таблицы? Какая настройка по умолчанию?
16. Что такое “отложить ограничения”? Какие градации есть для проверки ограничения?
17. Какие виды отложенных ограничений есть?
18. Полностью изобрази таблицу, а также какие ограничения можно отложить.
19. Перечисли случаи, когда мы используем отложенные ограничения.
20. Описать одну из ситуаций подробнее (Cyclic Foreign Keys, Reallocating Items, One per Group, Data Ingestion)
21. Какие минусы у отложенных ограничений и когда их лучше не использовать?
22. Что такое CHECK ограничение?