**Реляционные СУБД**

Сегодня поговорим о реляционных СУБД. Реляционные базы данных представляют данные как набор связанных **таблиц**, то есть одна база данных может содержать в себе множество таблиц. В каждой таблице может быть несколько полей (столбцов) и множество строк. В базе данных строки также называют **записями.**

Аналогией из реального мира может быть шкаф (база данных), внутри которого лежит множество коробок (таблиц), и уже внутри каждой коробки хранятся некие карточки с данными (записи).

**MySQL** — пример популярной реляционной системы управления базами данных, РСБД. Таблица баз данных хранит схожие данные одного вида с одинаковым набором характеристик. Например, в интернет-магазине игрушек в одной таблице могут храниться покупатели, у которых есть имена, фамилии, телефон и адреса, а в другой — товары, у которых есть артикул, название и цена. Другой пример: когда мы сохраняем данные из веб-формы на сайте, то добавляем новую запись в определенную таблицу базы данных. В этой записи будет несколько значений из формы: имя клиента, его телефон, его почта, а кроме них будут еще служебные значения: дата и время добавления записи, IP-адрес пользователя, заполнившего форму.

В одной записи присутствуют значения разных типов. Например, номер телефона — это целое число, начинающееся на 7 или 8, а имя клиента — текст ограниченной длины. В каждом поле (столбце) таблицы должны храниться данные заранее определенного типа. Аналогичная строгая типизация бывает и в языках программирования. Но не во всех: например, в JavaScript ее как раз нет.

Некоторые значения СУБД может подставить сама (например, подставлять текущую дату и время). Другие значения наше приложение должно явно передать СУБД для создания записи. Поскольку все значения одной записи (строки) СУБД разложит по полям таблицы (по столбцам), то при создании каждого поля в таблице базы данных необходимо заранее подумать, какой тип данных будет там храниться (дата или текст), нужно ли этому полю значение по умолчанию (например, текущая дата), может ли это поле быть пустым, какая предварительная настройка необходима для того, чтобы в будущем в нее были записаны неподходящие значения. Кроме того, если СУБД заранее знает тип поля, то она может хранить данные оптимальным образом, а затем быстрее по этим данным делать выборки.

В уже заполненные таблицы мы можем добавлять столбцы и даже удалять существующие. Но это становится рискованной процедурой, когда у данных, удаляемых в столбце, имеются связи с другими таблицами. Например, если в таблице с товарами мы имеем столбец с артикулами, то его уже нельзя будет удалить без последствий, если на эти артикулы указывают заказы из другой таблицы.

Проектирование реляционной базы данных — ответственный процесс. Зато во время работы такую строго устроенную базу проще понимать и использовать.

Типы данных и структура таблиц БД

При создании таблицы в базе данных:

* Задают имя таблицы
* Определяют её поля (и для каждого поля указывают тип данных, которые в нём могут храниться)
* Задают иные настройки полей: определяют значение по умолчанию, необходимость уникального значения, допустимость пустого поля, задают ключи и индексы — всё это мы обсудим далее
* Настраивают связи между таблицами

Типы данных

Рассмотрим различные типы данных на примере СУБД MySQL. В других СУБД набор типов может незначительно отличаться, но категории типов будут схожи. Мы будем рассматривать не все доступные типы, а только наиболее популярные. Полное руководство по типам вы при необходимости найдёте в [официальной документации](https://dev.mysql.com/doc/refman/8.0/en/data-types.html).

Числовые типы

В СУБД существуют отдельные типы для хранения целых и натуральных (беззнаковых) чисел разных диапазонов, вещественных чисел и даже денежных сумм.

Для примера рассмотрим три типа для хранения **целых чисел** с их беззнаковыми (UNSIGNED) вариациями:

|  |  |  |
| --- | --- | --- |
| **Тип** | **Объём памяти для хранения** | **Диапазон значений** |
| TINYINT | 1 байт | –128…127 |
| TINYINT UNSIGNED | 1 байт | 0…255 |
| INT | 4 байта | -2147483648…2147483647 |
| INT UNSIGNED | 4 байта | 0…4294967295 |
| BIGINT | 8 байт | -263...263-1 |
| BIGINT UNSIGNED | 8 байт | 0…264-1 |

Как видите, определение подходящего типа позволяет нам сэкономить память. Кроме того, именно по целочисленным значениям СУБД умеют максимально быстро искать.

Целые типы применяются для нумерации строк и во всех прочих случаях, когда значение будет целым числом.

Например, если некоторый магазин торгует только штучным товаром, то количество единиц в заказе разумно хранить как целое число. Притом выбрать беззнаковый вариант. И ограничиться типом INT (вряд ли в одном заказе будет больше, чем 4,2 млн. штук одного товара).

Но если тот же магазин торгует ещё и развесным товаром (например, продуктами питания, взвешивая их в килограммах), то целым количеством уже не обойтись.

Потребуются **вещественные** числа. В MySQL для этого существуют типы FLOAT (4 байта) и DOUBLE (8 байтов). В новых версиях MySQL эти типы уже не имеют беззнаковых вариантов.

Записать в такой тип количество, выраженное как «2,5 килограмма» можно без проблем. Однако могут возникать незначительные погрешности при операции над вещественными числами (их ещё называют «числами с плавающей точкой» или «числами с плавающей запятой», имея в виду разделитель целой и вещественной частей числа).

Погрешности допустимы, если мы считаем вес крупы в мешке, но когда мы работаем с финансовой информацией — это уже серьезная проблема. Для хранения денежных сумм в MySQL существует специальный тип DECIMAL или NUMERIC (в современной MySQL это можно рассматривать как синонимы). Операции над значениями таких типов происходят с учётом возможной детализации (например, до двух копеек после знака), эти типы называют **числами с фиксированной точкой**.

Дата и время

В зависимости от детализации в MySQL можно хранить только год, дату (год, число, месяц), дату и время, только время. Но наиболее часто используются на практике такие типы:

|  |  |  |
| --- | --- | --- |
| **Тип** | **Назначение** | **Пример значения** |
| DATE | Хранение дат, без времени | 2022-10-01 |
| DATETIME | Хранение даты и времени. Используется 8 байтов памяти | 2022-10-01 12:15 |
| TIMESTAMP | Хранение даты и времени в секундах, прошедших с 1 января 1970 года по Гринвичу. Используется 4 байта памяти | 1664615700 |

Типы даты и времени примечательны тем, что их можно хранить с учётом часовых поясов, а также легко выводить в разных международных и национальных форматах, например: 01.10.2022, 2022-10-01, 10/1/2022.

Строки и тексты

Для хранения текстовой информации, в зависимости от её объёма, в MySQL существуют разные типы:

|  |  |
| --- | --- |
| **Тип** | **Назначение** |
| CHAR | Строки фиксированной длины |
| VARCHAR | Строки переменной длины |
| TEXT | Текст неопределенной длины |

При создании полей типов CHAR и VARCHAR указывается максимальная длина хранимой строки. Например, поле с типом CHAR(10) сможет хранить 10 символов, а при попытке записи более длинного значения, все символы начиная с одиннадцатого будут отброшены. При этом CHAR всегда будет хранить именно 10 символов (а если строка короче, то оставшиеся места, грубо говоря, будут заполнены пробелами). VARCHAR же для каждого значения сохраняет его фактическую длину (не более заданной для всего поля), но при этом вместе с каждым значением приходится хранить ещё и количество значимых символов.

Если в поле вы храните тексты примерно одинаковой длины (например, серию и номер документа или артикул товара), то уместно использовать CHAR. Для строк переменной длины (например, для хранения адресов) подойдёт VARCHAR с разумным ограничением, например: VARCHAR(1000).

Тип TEXT хранит до 65КБ текстовой информации, но по умолчанию сортировка и поиск возможны только по небольшому фрагменту в начале текста.

Другие параметры полей

Для создаваемых в MySQL полей можно указывать дополнительные параметры кроме их типа. Рассмотрим некоторые из них:

* NOT NULL или NULL — указывает на то, может ли поля быть незаполненным (пустым)
* AUTO\_INCREMENT — для полей с целым типом автоматически заполняет очередную строку следующим значением (например, если в предыдущей строке в поле было 15, то в следующей в том же поле будет 16)
* UNIQUE — требует, чтобы значение поля было уникальным (т.е. в этой таблицы внутри этого столбца значения повторяться не могут)
* DEFAULT — значение по умолчанию, которым будет заполняться поле, если при создании очередной строки в этот столбец не передано конкретное значение
* PRIMARY KEY — указывает, что поле будет первичным ключом таблицы (это уникальное индексируемое значение, по которому можно однозначно различать строки таблицы, обычно — это номер строки)

Ключи

Вы уже знаете про первичный ключ — обычно это уникальное непустое поле, которое однозначно определяет строку с таблицей (например, номер строки). Важно учесть два момента:

* Когда мы говорим номер, то имеем в виду очередной номер присвоенный при добавлении строки в таблицу. Если при выводе таблицы мы её как-то иначе отсортируем, то этот номер не изменится (хотя в отсортированной таблице строка может оказаться на совсем другой позиции)
* В одной таблице может быть только один первичный ключ
* Определить уникальность строки можно с помощью комбинации полей. Например, предположим, что мы храним в базе данных паспорта и завели два отдельных поля: для серии и для номера. Тогда уникальность строки (паспорта) определяется комбинацией серии и номера. При этом может существовать несколько паспортов с одинаковыми номерами, но серии у них будут разные. Такие составные ключи мы с вами далее рассматривать не будем, это возможный, но достаточно нетривиальный случай

Кроме этого существуют внешние ключи (FOREIGN KEY), которые используются для связи таблиц. Внешних ключей в таблице может быть несколько. Внешний ключ в записи текущей таблицы обычно хранит значение первичного ключа в какой-то записи другой связанной таблицы БД.

СУБД автоматически контролирует, чтобы при создании новой записи текущей таблицы мы ссылались на существующую запись связанной таблицы.

Можно настроить таблицы так, чтобы при удалении записи также удалялись и записи, связанные с ней через внешний ключ.

Таким образом именно ключи синхронизируют данные между таблицами.

Связи

Связи упоминаются даже в определении реляционных баз данных и являются их неотъемлемым свойством.

Ещё до того, как мы заполним базу данными и начнём делать запросы — нужно продумать, как записи в таблицах должны быть связаны между собой.

Далее рассмотрим возможные типы связей.

**Один ко многим**

Одной записи из первой таблицы могут соответствовать сразу несколько записей из второй.

Например, у нас есть гражданин и его номера телефонов. У некоторых людей по несколько номеров. Но у каждого номера – только один владелец.

Для реализации такой связи во второй таблице добавляют внешний ключ (поле для хранения ключа из первой таблицы). Например, для каждого номера телефона храним идентификатор гражданина.

Обратите внимание, что при этом у нас могут быть граждане, у которых нет номера телефона вовсе. Тогда во второй таблице с номерами не будет ни одной записи, где бы упоминался идентификатор такого старомодного гражданина.

**Один к одному**

Из названия понятно, что при такой связи двух таблиц одной строке из первой таблицы должна соответствовать только одна строка из другой таблицы. Например, у гражданина России может быть только один действующий российский паспорт (и, наоборот, у паспорта может быть только один хозяин).

Возникает вопрос: а почему бы в таком случае не сложить данные гражданина (имя, фамилию, дату рождения) и паспорта (серию и номер, дату выдачи) в одну таблицу? Основная причина в том, что нам могут для других нужд потребоваться эти данные по отдельности. Например, мы можем коллегам гражданина предоставить доступ к его имени, фамилии и дате рождения, но при этом не должны показывать серию и номер паспорта.

Такую связь реализуют обычно также, как и один ко многим (во второй таблице заводят внешний ключ со значениями из первой), но при этом накладывают на поле требование по уникальности. Например, с учётом требования по уникальности в таблице паспортов у нас не сможет существовать двух строк с одинаковым идентификатором гражданина.

Связь один к одному можно реализовать и в обратном направлении: то есть в таблице с гражданами сделать внешний ключ, указывающий на связанный паспорт.

**Многие ко многим**

Одной записи из первой таблицы могут соответствовать несколько строк из второй и наоборот.

Пример: граждане и объекты недвижимости. У одного гражданина может быть несколько квартир. Но и одной квартирой могут владеть сразу несколько граждан (через долевую собственность).

Для реализации такой связи приходится использовать третью таблицу, каждая запись которой хранит сразу два ключа: из первой таблицы (например, идентификатор гражданина) и из второй таблицы (например, кадастровый номер объекта недвижимости).

Именования таблиц и полей

Есть разные подходы к именованию в базах данных, рассмотрим наиболее простой, на наш взгляд, подход.

Предположим мы проектируем базу данных магазина. В ней будут такие сущности: товары, покупатели, заказы.

В этом случае для каждой сущности создают отдельную таблицу и называют её существительными во множественном числе: products, customers, orders.

Поля внутри таблиц именуют также существительными, но в единственном числе. Например, если в таблице товаров мы должны хранить артикулы, названия и цену товаров, то поля будут: sku (англ. *артикул*), title, price. Но, скорее всего, к этим полям ещё добавится поле первичного ключа id – уникальный числовой идентификатор товара (номер строки при вставке).

При этом в той же базе будет вторая таблица с покупателями, где для каждого из них мы храним имя, фамилию, почту и телефон. Названия полей будут: name, surname, email, phone. И к ним тоже добавится первичный ключ id.

Нет никакой проблемы в том, что одноименное поле id присутствует сразу в двух или более таблицах. Главное, чтобы в одной таблице все поля имели разные имена.

Если требуется создать служебную таблицу для реализации связи многие ко многим, то имя этой таблицы составляют из названий сущностей в единственном числе, располагая их по алфавиту и разделяя символом подчеркивания. Например, если у нас есть таблица товаров (products) и заказов (orders), то таблица для связи между ними должна называться order\_product.

Язык SQL. Основные операторы

**SQL** — это структурированный язык запросов (от англ. *Structured Query Language*), созданный для того, чтобы получать, добавлять или изменять информацию в базе данных.

С помощью SQL пользователь формирует запрос (англ. *query*) и направляет его в СУБД. Та в свою очередь обрабатывает эту информацию, «понимает», что именно нужно пользователю, и отправляет ответ. Пользователем может быть необязательно человек (он в таком случае называется оператором БД), но и другая программа. Ответом может быть таблица (или её часть, или наоборот новая таблица составленная из фрагментов исходных), отдельная строка или даже одно значение.

SQL — это не язык программирования, поэтому написать целый сайт с его помощью не выйдет. Но при этом приложения на других языках (например, на PHP) почти всегда включают в себя SQL-запросы для общения с базой.

Поиск в Яндексе чем-то похож на работу с базой данный: вы отправляете запросы и получаете результаты с ответами. Но с Яндексом вы общаетесь на естественном языке, а с СУБД придётся говорить на SQL.

SQL используют разные виды специалистов:

* Аналитики и продуктовые маркетологи. Знание SQL даёт им возможность не зависеть от программистов и самостоятельно получать и обрабатывать данные
* Разработчики. С помощью SQL они сохраняют данные из своих приложений и получают нужные выборки
* Менеджеры тоже должны понимать, как устроены и работают СУБД, чтобы оставаться в контексте ИТ-команды

SQL состоит из операторов (или, как их ещё называют, команд), которые используются для построения запросов и выполнения операции — например, для выбора из множества по конкретному параметру.

Операторы в SQL делятся на несколько групп в соответствии с задачами, которые они решают.

Существуют операторы, которые работают с целыми таблицами:

* CREATE — создание таблицы в базе данных
* ALTER — изменение таблицы
* DROP — удаление таблицы

Существуют также операторы манипуляции данными. Эти операторы работают с содержимым таблиц — строками, полями и значениями:

* SELECT — выборка данных
* INSERT — добавление строк
* UPDATE — изменение данных
* DELETE — удаление данных

Рассмотрим пример запроса на создание таблицы товаров:

1CREATE TABLE products (

2 id INT UNSIGNED AUTO\_INCREMENT PRIMARY KEY,

3 sku CHAR(15) NULL UNIQUE,

4 title VARCHAR(100) NOT NULL,

5 price DECIMAL(10,2) NOT NULL

6);

Итак, мы создаем таблицу с названием products, в которой будет 4 поля.

Первое из них id будет первичным ключом (натуральное число, значение которого будет автоматически прирастать на единицу при добавлении очередной строки).

Поле sku будет хранить 15 символов, но при этом может быть незаполненным (например, пока для товара уточняется артикул). Однако, в случае заполнения, артикул должен быть уникальным (не повторяться для разных товаров).

Поле title не должно быть пустым и может иметь переменную длину до 100 символов.

Наконец, в поле price мы сможем записывать суммы (до 10 знаков перед запятой и 2 знака после). Цена также не должна быть пустой.

После создания таблицы вставить в неё данные можно таким запросом (добавим пару строк):

1INSERT INTO products (sku, title, price) values ('PEN756RED', 'Красный карандаш', 22.49);

2INSERT INTO products (sku, title, price) values ('PEN756BLUE', 'Синий карандаш', 20.49);

Обратите внимание, что мы явно не указываем id, как раз для того, чтобы он заполнился автоматически. Зато указали артикул, название и цену товара.

И поскольку теперь в нашей таблице есть данные, то мы можем их все отобразить с помощью запроса:

1SELECT \* FROM products;

Получим такой вывод:

|  |  |  |  |
| --- | --- | --- | --- |
| id | sku | title | price |
| 1 | PEN756RED | Красный карандаш | 22.49 |
| 2 | PEN756BLUE | Синий карандаш | 20.49 |

Звёздочка \* означает, что мы хотим вывести все столбцы из таблицы products. Но можно указать конкретные и даже в желаемом порядке:

1SELECT title, sku, price FROM products;

Получим такой вывод:

|  |  |  |
| --- | --- | --- |
| title | sku | price |
| Красный карандаш | PEN756RED | 22.49 |
| Синий карандаш | PEN756BLUE | 20.49 |

Теперь добавим ещё пару товаров:

1INSERT INTO products (sku, title, price) values ('PEN756GRAPHIT', 'Простой карандаш', 12.49);

2INSERT INTO products (sku, title, price) values ('PEN756GREEN', 'Зеленый карандаш', 20.00);

И выведем их отсортировав по цене c помощью оператора ORDER BY:

1SELECT \* FROM products ORDER BY price ASC;

Оператору мы сообщаем по какому столбцу надо сортировать и в каком порядке (ASC — по возрастанию, DESC — по убыванию, а если не указать порядок сортировки, то будет по умолчанию сортироваться по возрастанию).

В результате получим:

|  |  |  |  |
| --- | --- | --- | --- |
| id | sku | title | price |
| 3 | PEN756GRAPHIT | Простой карандаш | 12.49 |
| 2 | PEN756BLUE | Синий карандаш | 20.49 |
| 1 | PEN756RED | Красный карандаш | 22.49 |
| 4 | PEN756GREEN | Зеленый карандаш | 20.00 |

Что если мы хотим получить не все результаты, а отфильтровать их по какому-то условию? Для этого существует оператор WHERE, задающий условия выборки. Выберем товары, у которых цена от 10 до 20 рублей включительно:

1SELECT \* FROM products WHERE price >= 10 AND price <= 20;

Обратите внимание, что после WHERE указали сразу два условия: «больше или равно 10» и «меньше и равно 20». Отдельные условия чаще всего связывают операторами AND (логическое И) либо OR (логическое ИЛИ), также условия можно группировать с помощью круглых скобок и инвертировать (с ложного на истинное и наоборот) с помощью оператора NOT.

В результате получим две записи из таблицы:

|  |  |  |  |
| --- | --- | --- | --- |
| id | sku | title | price |
| 3 | PEN756GRAPHIT | Простой карандаш | 12.49 |
| 4 | PEN756GREEN | Зеленый карандаш | 20.00 |

К строкам тоже можно применять арифметическое сравнение, тогда они будут сравниваться посимвольно. Но обычно при работе со строками нас интересует совпадение с какой-то подстрокой (поиск).

Для поиска подстрок используется оператор LIKE, после него аргумент заключается в кавычки, как и все строковые значения в SQL. В аргументе обычно используют знак процента %, который означает «любое количество каких-то символов», или знак подчеркивания, который означает «один какой-то символ».

Вот таким запросам мы выберем все товары, в названии которых есть подстрока «ый»:

1SELECT \* FROM products WHERE title LIKE '%ый%';

В результате получим две подходящих записи:

|  |  |  |  |
| --- | --- | --- | --- |
| id | sku | title | price |
| 1 | PEN756RED | Красн**ый** карандаш | 22.49 |
| 4 | PEN756GREEN | Зелен**ый** карандаш | 20.00 |

В SQL существуют агрегатные функции, которые производят вычисления над строками. Наиболее популярные из них:

* AVG— вычисляет среднее значение в указанном столбце
* SUM — вычисляет сумму по указанному столбцу
* MIN — находит наименьшее значение в указанном столбце
* MAX— находит наибольшее значение в указанном столбце
* COUNT— подсчитывает количество строк в результате

Например, вот таким запросом мы можем посчитать суммарную стоимость всех добавленных товаров:

1SELECT SUM(price) FROM products;

В качестве ответа мы получим одно значение — 75.47.

Далее мы на практике познакомимся с SQL.

# Клиент-серверные и встроенные СУБД

СУБД — программа для управления базой данных, она может быть запущена на отдельном компьютере (и тогда это будет выделенный сервер баз данных). Доступ к серверу предоставляется через сеть.

Одним мощным сервером СУБД могут пользоваться тысячи клиентов — людей и программ, которые обращаются к СУБД на сервере. Примером **клиент-серверной** СУБД является, в том числе MySQL.

Но, если база и все её клиенты находятся на одном компьютере, то СУБД называется **локальной**. В этом случае для СУБД не предоставляется доступ в интернет для большей безопасности.

Однако, на одном компьютере может быть запущено несколько программ, все они могут по-прежнему иметь доступ в базу, как и несколько локальных пользователей. То есть локальная СУБД при этом остаётся многопользовательской. По-прежнему несколько клиентов одновременно могут читать и записывать данные.

Существуют базы данных, предназначенные для нужд конкретного приложения. Такая СУБД, как правило, не запущена на компьютере постоянно, а запускается либо для обработки конкретного запроса, либо только на время работы родительского приложения.

В таких СУБД не предусмотрена параллельная запись, поскольку одно приложение может выстроить все запросы последовательно (а никакое другое приложение к СУБД подключаться в это время и не будет).

Подобные СУБД называются **встроенными**. Пример — SQLite. Она существует для всех платформ, но встраивается в приложения как дополнительный готовый модуль, и все данные хранит в виде одного бинарного файла.

Построение концептуальной модели в нотации Чена

Модели, которые мы с вами изучаем в этой теме, также называются **ER-моделями** (от англ. Entity-Relationship, т.е. «сущность-связь»). **Сущность** – это объект из предметной области (из бизнеса, из реального мира), который будет в последующем храниться в базе данных.

Приведем примеры сущностей интернет-магазина:

* Товар;
* Заказ;
* Клиент;
* Менеджер;
* Курьер.

**Связь** — это отношение между сущностями.

Обычно связь можно описать глаголом или используя глагольную форму. Приведем примеры связей сущностей интернет-магазина:

* Заказ содержит товары;
* Клиенты создают заказы;
* Заказы обрабатывают менеджеры;
* Заказы доставляют курьеры.

Ещё в 70-х годах Питер Чен предложил **нотацию** (набор обозначений символов), с помощью которых можно строить наглядные концептуальные ER-модели с сущностями и связями.

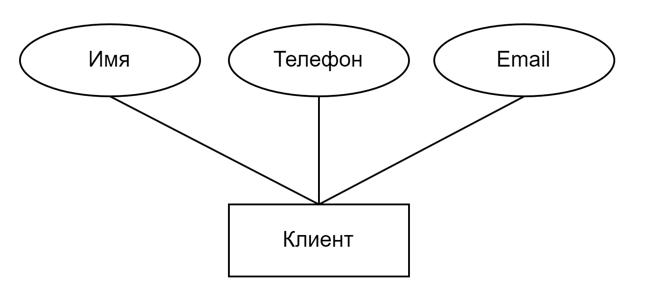
**Основные элементы нотации Чена**

Сущность — изображается в виде прямоугольника с названием сущности:

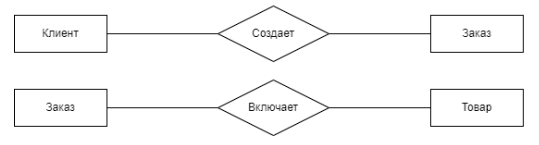


У сущности могут быть свойства (например, свойства товара — производитель, название, цена), которые в ER-моделях обычно называются **атрибутами**. У разных сущностей есть разные атрибуты, хотя они могут и повторяться. Например, и у курьера, и у менеджера, и у клиента есть такой атрибут, как номер телефона. Если телефон важен для нас в контексте создаваемой системы, то он будет существовать у каждой из трех сущностей.

Изображаются сущности в виде овалов, связанных линиями с определяемой сущностью. Пример изображения атрибутов клиента:



Связь (отношение) графически изображается ромбом с указанием названия отношения (обычно в форме глагола):



**Типы связей**

Тип связи (или степень родства) между сущностями  показывает, как сущности взаимодействуют между собой, сколько сущностей участвуют в связи. Рассмотрим эти типы:

1. **Связь один к одному (1:1)**

Показывает, что один объект из сущности Х будет связан с одним объектом из сущности Y, и наоборот. Например, сотрудник интернет-магазина может управлять только одним отделом, и наоборот — каждый отдел управляется только одним сотрудником. Связь 1-к-1 графически изображается следующим образом:



1. **Связь один ко многим (1:N)**

Показывает, что один объект из сущности X может быть связан с множеством объектов из сущности Y, но при этом каждый объект из сущности Y может быть связан только с одним объектом из сущности X. Например, клиент может создать множество заказов, но при этом каждый такой заказ может быть создан только одним клиентом (не может быть в интернет-магазине у заказа сразу много заказчиков). Связь графически изображается следующим образом:



Иногда отдельно упоминают связь многие к одному (N:1), но по сути это просто обратный вариант связи один ко многим.

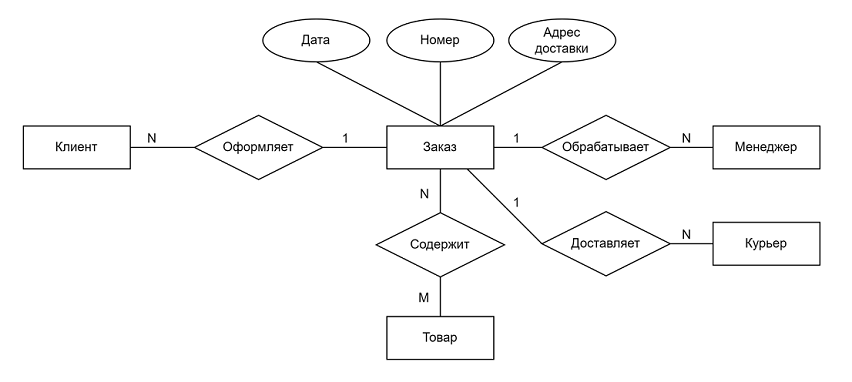
1. **Связь многие ко многим (M:N)**

Показывает, что один объект из сущности X может быть связан с несколькими объектами из сущности Y и наоборот (один объект Y может быть в то же самое время связан с несколькими объектами X). Например, в один заказ может быть включено несколько товаров, и при этом один товар может входить в состав сразу нескольких разных заказов. Связь графически изображается следующим образом:



В нотации Чена есть и другие элементы, но мы ограничимся рассмотренными. С их помощью можно строить наглядную визуализацию концептуальной модели.

Теперь рассмотрим пример концептуальной ER-модели интернет-магазина, спроектированной по правилам нотации Чена:



В данном примере показаны атрибуты, сущности, связи между ними и типы этих связей, а также указаны ограничения по каждому типу связи.

Атрибуты представлены только для сущности Заказ. Концептуальная модель допускает отсутствие атрибутов у некоторых сущностей, но мы рекомендуем вам по возможности делать два варианта диаграмм: более компактную без атрибутов и более детальную с атрибутами.

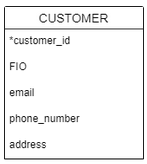
В качестве упражнения придумайте атрибуты для остальных сущностей интернет-магазина, кроме Заказа.

Для построения логической модели считается наиболее подходящей ER-модель в нотации Джеймса Мартина (и группы других исследователей), эта нотация широко известна как **Crow's Foot** («воронья лапка», скоро вы узнаете откуда взялось такое забавное название).

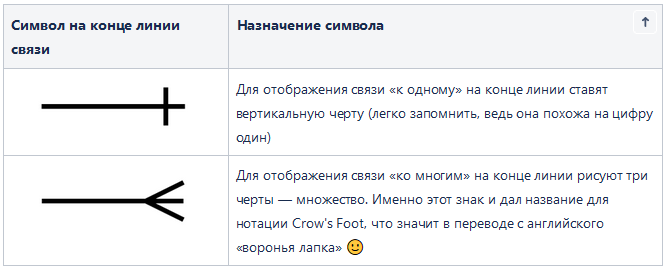
Рассмотрим как в этой нотации отображаются основные элементы:

**Сущность** изображается в виде **таблицы**, заголовок которой отделяется чертой и содержит название сущности, а последующие строки — содержат **атрибуты** этой сущности. Притом среди атрибутов можно с помощью звёздочки **\*** можно выделять **ключевые** (по которым можно будет однозначно идентифицировать экземпляр сущности).

Приведем пример одной из сущностей интернет-магазина (клиент):



**Связь** (отношение) изображается с помощью **линии** между сущности. Но концы этой линии содержат специальные обозначения, которые, в частности, определяют тип связи (один к одному, один ко многим, многие ко многим). В подписи к линии указывают название связи (обычно тоже в виде глагола).



Если требуется отобразить связь «один ко многим», то, соответственно, на одном конце линии связи будет черта, а на другом конце линии лапка.

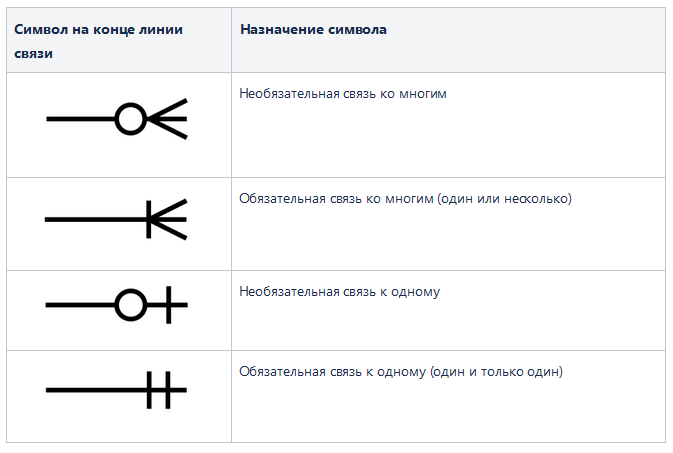
Но в этой нотации обычно линии отражают не только тип связи, но и её **обязательность** или, как ещё говорят, ограничение связи.

### Ограничение связи

Ограничение показывает, может ли сущность не участвовать в связи. Например, в интернет-магазине могут быть товары, которые пока ни разу не заказаны (не связаны ни с одним заказом).

* Полное участие (**обязательная связь**) — показывает, что каждый объект сущности вовлечен в отношение. Например, у любого заказа есть связанный покупатель (клиент), который этот заказ оформил. И не может быть заказа без покупателя.
* Частичное участие (**необязательная связь**) – показывает, что не все объекты в сущности вовлечены в отношение. Например, в интернет-магазине могут быть клиенты, которые уже зарегистрировались на сайте, но ещё не оформили ни одного заказа.

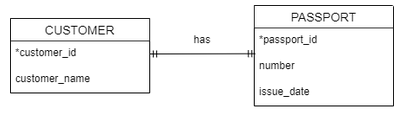
Ограничения связи изображаются следующим образом — на конец линии добавляют второй символ: черту для обозначения обязательной связи и пустой кружок для обозначения необязательной связи.



## Примеры

Далее рассмотрим конкретные примеры (фрагменты логической диаграммы в нотации Crow's Foot).

Пример обязательной связи «один к одному» — гражданин и его паспорт:



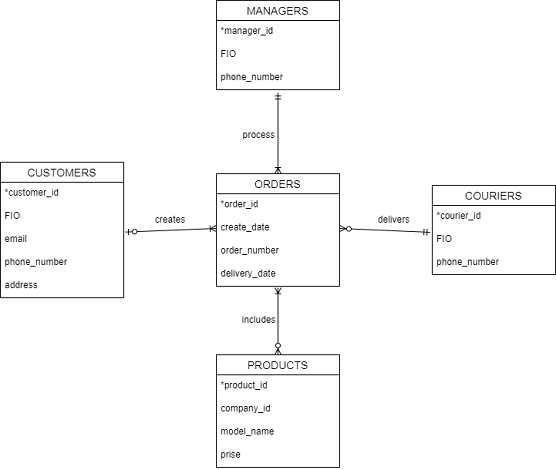
Подразумеваем, что в рамках некой информационной системы у гражданина обязательно должен быть паспорт (связь обязательная), и при этом паспорт тоже обязательно должен иметь одного хозяина (не может быть в системе бесхозных паспортов).

Пример связи многие ко многим, обязательной только с одной из сторон:



Итак, в примере мы встретились с сущностями Заказ и Товар. В заказе может быть много товаров, поэтому на левом конце линии использована лапка. Но при этом заказ не может быть пустым, один или несколько товаров в нём всегда будут — поэтому связь обязательная (добавили вертикальную черту). В то же время товар тоже может попасть в несколько разных заказов (поэтому справа тоже лапка). Но при этом в интернет-магазине могут быть товары, которые пока ни разу не заказывали — связь необязательная (справа к лапке добавили пустой кружок, похожий на ноль, что только подчеркивает, что некоторые товары могут быть не участвовать в связи).

Теперь посмотрим, как могла знакомая концептуальная модель интернет-магазина видоизмениться и расшириться при переходе к логической модели:



Кроме рассмотренных нами нотаций Чена и «вороньих лапок» существуют и другие нотации для визуализации моделей данных. Например, стрелочная нотация. У нотаций разные символы, но схожая логика, поэтому с другими нотациями вы сможете разобраться даже самостоятельно, если встретите их в рабочих проектах.

**Нормализация** — это  процесс сокращения избыточности данных (чтобы одни и те же данные не дублировались в разных таблицах) и устранения аномалий, которые могут возникать при добавлении, изменении, удалении данных.

Для выполнения нормализации таблицы БД приводятся в соответствие определенным требованиям. Своды этих требований называются **нормальными формами**.

Существует ненормализованная  форма данных (UNF), пять основных нормальных форм (1NF-5NF), а также нормальная форма Бойса-Кодда (BCNF), доменно-ключевая нормальная форма (DKNF) и шестая нормальная форма (6NF).

А теперь расслабьтесь, потому что вам не требуется всё это запоминать.

База данных считается нормализованной, если она находится в третьей нормальной форме (3NF). Так что первыми тремя нормальными формами (1NF-3NF) мы и ограничимся в текущей теме.

**1NF**

Чтобы таблица соответствовала **первой нормальной форме (1NF)** необходимо, чтобы:

* В таблице не должно быть дублирующихся полностью строк
* В каждом поле таблицы хранится атомарное значение (одно, неделимое, несоставное)

Предположим, у нас имеется таблица R1 с заказами интернет-магазина игрушек в таком первоначальном виде:

|  |  |
| --- | --- |
| **Покупатель** | **Товары** |
| Иванова Мария, 7916001223 | Плюшевый мишка - 1 шт., Синее ведёрко - 1 шт. |
| Фролов Пётр, 7919002334 | Жёлтый совочек - 1 шт., Синее ведёрко - 1 шт. |
| Карпова Зоя, 7911003445 | Красный мячик - 2 шт., Плюшевый мишка - 1 шт. |
| Иванова Мария, 7916001223 | Синее ведёрко - 1 шт. |
| Зорина Вера, 7913004556 | Железная дорога - 1 шт. |

Таблица не нормализована по 1NF, потому что в обеих колонках данные не атомарны (хранятся составные значения, например, список товаров).

Преобразуем таблицу, обеспечив атомарность:

|  |  |  |  |
| --- | --- | --- | --- |
| **Телефон**  **покупателя** | **Покупатель** | **Товары** | **Кол-во** |
| 7916001223 | Иванова Мария | Плюшевый мишка | 1 |
| 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 7919002334 | Фролов Пётр | Жёлтый совочек | 1 |
| 7919002334 | Фролов Пётр | Синее ведёрко | 1 |
| 7911003445 | Карпова Зоя | Красный мячик | 2 |
| 7911003445 | Карпова Зоя | Плюшевый мишка | 1 |
| 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 7913004556 | Зорина Вера | Железная дорога | 1 |

Но появилась другая проблема: Иванова Мария дважды в разных заказах приобретала товар «Синее ведёрко», теперь это приводит к полному дублированию двух строк (второй и предпоследней).

Чтобы различать заказы — начнём хранить их дату:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Дата заказа** | **Телефон**  **покупателя** | **Покупатель** | **Товары** | **Кол-во** |
| 2022-10-22 11:21 | 7916001223 | Иванова Мария | Плюшевый мишка | 1 |
| 2022-10-22 11:21 | 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 2022-10-22 17:22 | 7919002334 | Фролов Пётр | Жёлтый совочек | 1 |
| 2022-10-22 10:55 | 7919002334 | Фролов Пётр | Синее ведёрко | 1 |
| 2022-09-21 17:33 | 7911003445 | Карпова Зоя | Красный мячик | 2 |
| 2022-09-21 17:33 | 7911003445 | Карпова Зоя | Плюшевый мишка | 1 |
| 2022-09-21 17:33 | 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 2022-09-21 14:18 | 7913004556 | Зорина Вера | Железная дорога | 1 |

Теперь все строки уникальны (ведь два заказа Ивановой Марии отличаются по дате), теперь таблица R1 в первой нормальной форме.

**Ключ**

Про ключевые поля мы уже говорили, но давайте уточним это понятие.

**Ключ** – это столбец или набор столбцов, по которым гарантировано можно отличить строки друг от друга, т.е. ключ идентифицирует каждую строку таблицы. По ключу мы можем обратиться к конкретной строке данных в таблице.

Ключи может быть **простыми** (состоит из одного поля) или **составными** (состоит сразу из нескольких полей, которые вместе определяют уникальность строки). Даже если ключ составной, то по возможности надо включать в него как можно меньше полей.

Ключи бывают **естественными** (следуют из реальных данных, например, как номер телефона) и **суррогатными** (генерируются СУБД, например ,в виде последовательных номеров из множества натуральных чисел).

Наконец, ключ может быть **основным** (англ. *primary key*) или, как ещё говорят, **первичным** — по этому ключу другие таблицы связывают с текущей (делают ссылки из других таблиц на текущую).

И, наоборот, если сущности текущей таблицы мы собираемся связывать с сущностями какой-то другой таблицы, то в текущей создаём **внешний ключ** (англ. *foreign key*). Внешних ключей может быть несколько.

**2NF**

Чтобы таблица соответствовала **второй нормальной форме (2NF)** необходимо, чтобы:

* Таблица должна находиться в первой нормальной форме
* Таблица должна иметь ключ
* Все неключевые столбцы таблицы должны зависеть от полного ключа (в случае если он составной)

В таблице R1 ключом может быть пара полей — Дата заказа и Телефон покупателя. Взять в роли ключа только дату — мы не можем, потому что у нас есть заказы от разных покупателей, совершенные одновременно. Взять в роли ключа отдельно только телефон покупателя — тоже не можем, потому что нам уже известен пример Марии, которая совершила пару заказов.

Заголовок ключевого поля (или нескольких полей в составном ключе) в таблице обозначается подчеркиванием. Поэтому с составным ключом из Даты заказа и Телефона наша таблица будет выглядеть так:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Дата заказа** | **Телефон**  **покупателя** | **Покупатель** | **Товары** | **Кол-во** |
| 2022-10-22 11:21 | 7916001223 | Иванова Мария | Плюшевый мишка | 1 |
| 2022-10-22 11:21 | 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 2022-10-22 17:22 | 7919002334 | Фролов Пётр | Жёлтый совочек | 1 |
| 2022-10-22 10:55 | 7919002334 | Фролов Пётр | Синее ведёрко | 1 |
| 2022-09-21 17:33 | 7911003445 | Карпова Зоя | Красный мячик | 2 |
| 2022-09-21 17:33 | 7911003445 | Карпова Зоя | Плюшевый мишка | 1 |
| 2022-09-21 17:33 | 7916001223 | Иванова Мария | Синее ведёрко | 1 |
| 2022-09-21 14:18 | 7913004556 | Зорина Вера | Железная дорога | 1 |

Товар и его количество в заказе зависят от заказа в целом. Заказ в целом определяется ключом из пары значений (дата и телефон).

Но вот покупатель (фамилия и имя) никак не связаны с датой заказа. Могут происходить заказы в разные даты от одного покупателя, но в составном ключе нас будет интересовать только телефон (именно телефон однозначно связан с покупателем). То есть поле Покупатель зависит не от целого ключа (не от комбинации двух полей), а только от одного (от поля Телефон).

Чтобы устранить проблему, вынесем покупателей в отдельную таблицу:

Таблица R1:

|  |  |  |  |
| --- | --- | --- | --- |
| **Дата заказа** | **Телефон**  **покупателя** | **Товары** | **Кол-во** |
| 2022-10-22 11:21 | 7916001223 | Плюшевый мишка | 1 |
| 2022-10-22 11:21 | 7916001223 | Синее ведёрко | 1 |
| 2022-10-22 17:22 | 7919002334 | Жёлтый совочек | 1 |
| 2022-10-22 10:55 | 7919002334 | Синее ведёрко | 1 |
| 2022-09-21 17:33 | 7911003445 | Красный мячик | 2 |
| 2022-09-21 17:33 | 7911003445 | Плюшевый мишка | 1 |
| 2022-09-21 17:33 | 7916001223 | Синее ведёрко | 1 |
| 2022-09-21 14:18 | 7913004556 | Железная дорога | 1 |

Таблица R2:

|  |  |
| --- | --- |
| **Телефон**  **покупателя** | **Покупатель** |
| 7916001223 | Иванова Мария |
| 7919002334 | Фролов Пётр |
| 7911003445 | Карпова Зоя |
| 7913004556 | Зорина Вера |

Теперь таблица R1 соответствует второй нормальной форме.

Впрочем, и таблица R2 — тоже, ведь во второй таблице ключом мы сделали поле Телефон покупателя, а это у нас простой (не составной) ключ.

**Если в таблице, которая уже нормализована к 1NF, ключ является простым (состоит из одного поля), а не составным, то таблица автоматически переходит во вторую нормальную форму.**

Так что вводить во всех таблицах отдельным полем суррогатные ключи (уникальные числовые ключи из последовательности натуральных чисел) — это общепринятая и полезная практика.

**3NF**

Чтобы таблица соответствовала **третьей нормальной форме (3NF)** необходимо, чтобы:

* Таблица должна находиться во второй нормальной форме
* Все неключевые столбцы в таблице должны зависеть только от первичного ключа (но не от значений других неключевых столбцов).

Предположим, что теперь наш магазин игрушек продаёт товары с доставкой. Поэтому к каждому заказу добавляется информация о курьере, который доставляет заказ.

Таблица R1 теперь такая:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Дата заказа** | **Телефон**  **покупателя** | **Товары** | **Кол-во** | **Курьер** | **Телефон**  **курьера** |
| 2022-10-22 11:21 | 7916001223 | Плюшевый мишка | 1 | Игорь | 79230098877 |
| 2022-10-22 11:21 | 7916001223 | Синее ведёрко | 1 | Игорь | 79230098877 |
| 2022-10-22 17:22 | 7919002334 | Жёлтый совочек | 1 | Павел | 79260087766 |
| 2022-10-22 10:55 | 7919002334 | Синее ведёрко | 1 | Игорь | 79230098877 |
| 2022-09-21 17:33 | 7911003445 | Красный мячик | 2 | Павел | 79260087766 |
| 2022-09-21 17:33 | 7911003445 | Плюшевый мишка | 1 | Павел | 79260087766 |
| 2022-09-21 17:33 | 7916001223 | Синее ведёрко | 1 | Игорь | 79230098877 |
| 2022-09-21 14:18 | 7913004556 | Железная дорога | 1 | Павел | 79260087766 |

Таблица по-прежнему соответствует 2NF.

Но ситуация с курьерами похожа на ту, что была с покупателями: имя курьера зависит не от заказа (не от первичного ключа), а по сути от номера телефона курьера.

При этом номер телефона часть первичного ключа не является. Поэтому третьей нормальной форме таблица пока не соответствует.

Решим это созданием третьей таблицы, куда и вынесем имена курьеров.

Создадим новую таблицу R3:

|  |  |
| --- | --- |
| **Телефон**  **курьера** | **Курьер** |
| 79230098877 | Игорь |
| 79260087766 | Павел |

А таблица R1 теперь примет такой вид:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Дата заказа** | **Телефон**  **покупателя** | **Товары** | **Кол-во** | **Телефон**  **курьера** |
| 2022-10-22 11:21 | 7916001223 | Плюшевый мишка | 1 | 79230098877 |
| 2022-10-22 11:21 | 7916001223 | Синее ведёрко | 1 | 79230098877 |
| 2022-10-22 17:22 | 7919002334 | Жёлтый совочек | 1 | 79260087766 |
| 2022-10-22 10:55 | 7919002334 | Синее ведёрко | 1 | 79230098877 |
| 2022-09-21 17:33 | 7911003445 | Красный мячик | 2 | 79260087766 |
| 2022-09-21 17:33 | 7911003445 | Плюшевый мишка | 1 | 79260087766 |
| 2022-09-21 17:33 | 7916001223 | Синее ведёрко | 1 | 79230098877 |
| 2022-09-21 14:18 | 7913004556 | Железная дорога | 1 | 79260087766 |

И теперь таблица R1 соответствует третьей нормальной форме.

С формальной точки зрения наша таблица R1 теперь нормализована.

Однако сокращать избыточность данных можно и дальше. Например, чтобы повторно в каждой записи о заказе не сохранять название товара, можно создать четвертую таблицу с товарами, туда записать их названия и для каждого товара придумать первичный ключ (например, уникальный артикул). Тогда в таблице R1 сможем хранить только артикулы (которые никогда не меняются, в отличии от названий).

Также, чтобы избавиться от потенциальных проблем с составными ключами — часто вводят отдельное поле id, которое выступает как суррогатный ключ. И это даёт свои преимущества в смысле борьбы с избыточностью: если у клиента изменится номер телефона, то нам не придётся менять его в нескольких таблицах или строках, вынесем телефон отдельным полем в таблице клиентов, добавим туда ключ id, во всех прочих таблицах для связи будем использовать только этот добавленный суррогатный ключ. Тогда при редактировании телефона, его придётся изменить только в одной строке одной таблицы.

На практике вы увидите, как описанные подходы применяются в СУБД.

**Физическая модель** базы данных показывает по какой схеме должна быть устроена база в СУБД с учётом таблиц, полей, их типов, ссылок между таблицами.

Схема реляционной БД состоит из основных элементов — таблиц и ссылок, кроме этого существуют дополнительные: триггеры, хранимые процедуры, представления. Далее в рамках текущей темы мы с вами обсудим только основные элементы схемы БД.

Для начала обсудим как на уровне данных в таблицах СУБД **реализуются связи**.

Предположим, у нас есть таблица клиентов (customers) и таблица заказов (orders). В каждой из таблиц добавлен суррогатный ключ (поле id), поэтому мы можем быть уверены, что каждый добавленный в базу клиент и заказ — имеют уникальные id.

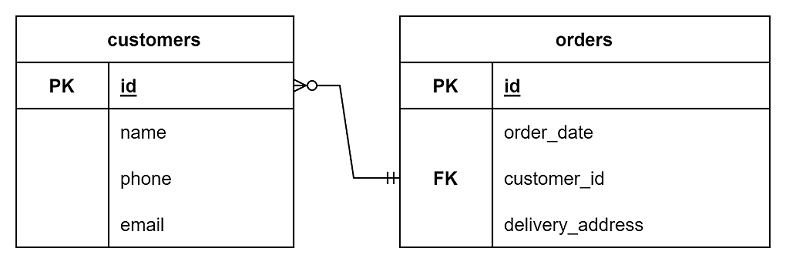
При этом, чтобы из каждой строки в таблице заказов (из каждого заказа) установить связь с неким клиентом — в таблицу заказов добавляют специальное поле (его логично назвать customer\_id).

И тогда в каждый заказ (а именно в поле customer\_id) мы будем записывать идентификатор клиента (id из таблицы customers), который этот заказ оформил.

В общем, получается, что если мы хотим заказы привязать к клиентам, то нам надо для этого добавить в заказы специальное связующее поле.

**Внешний ключ** (или ссылка, от англ. foreign key) – это связующий атрибут соответствующий, как правило, первичному ключу другой или той же самой таблицы. Правильно говорить, что внешний ключ одной таблицы является ссылкой на первичный ключ другой таблицы.

Рассмотрим пример:



В примере видим, что внешний ключ таблицы заказов (orders) ссылается на первичный ключ таблицы клиентов (customers). То есть в каждом заказе (как и обсуждали в примере) мы теперь храним связь с клиентом, указав его уникальный идентификатор. И при таком подходе нам не важно, как изменятся остальные поля клиента. Идентификатор останется прежним, а значит связь не пострадает.

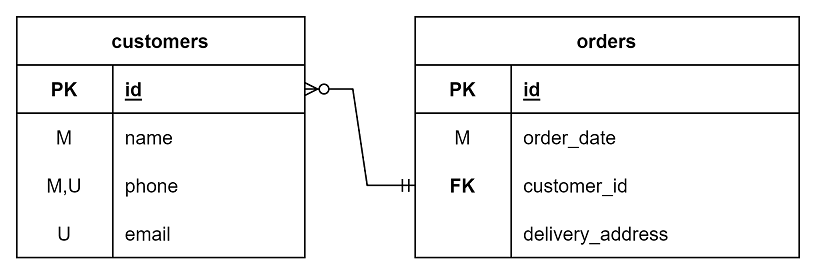
Нотация пример немного отличается от той, которую вы видели в диаграмме логической модели:

* Первичный ключ подчёркнут и отделён чертой от остальных атрибутов
* В таблицу сущности добавился столбец с метками (PK, FK и др.)

Разберемся, какие метки для атрибутов могут встречаться:

|  |  |
| --- | --- |
| **Метка** | **Свойство** |
| PK | Основной ключ (от англ. *primary key*) |
| FK | Внешний ключ (от англ. *foreign key*) |
| М | Обязательное поле (от англ. *mandatory*), каждая запись таблицы в этом поле должна содержать значение |
| О | Необязательное поле (от англ. *optional*), могут быть записи в таблице, у которых это поле не заполнено (считается, что в таком случае в поле лежит специальное значение *NULL*) |
| U | Поле с уникальными значениями (от англ. *unique*). Все непустые значения в этом поле должны быть уникальными (не повторяться в таблице дважды и более раз) |

Добавим метки атрибутам:



Получается, что у клиента обязательно указывать имя, но при этом имена могут быть неуникальными (среди клиентов, действительно, могут встречаться тёзки).

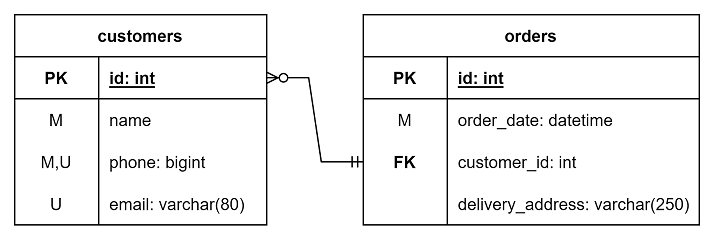
Телефон должен быть уникальным и при этом обязателен к заполнению. А вот без электронной почты мы клиентам разрешим оформлять заказы, но если клиент решил всё-таки указать почту, то она тоже должна быть уникальной.

В заказах даты хоть и обязательны, но могут повторяться (например, пара разных клиентов могут оформить заказ в один момент, это допустимо). А вот адрес доставки необязателен (могут быть заказы с самовывозом, там адрес не нужен) и может повторяться (один и тот же или даже разные клиенты — например, супруги — могут совершать разные заказы с доставкой на один и тот же адрес).

У ключевых полей дополнительные метки можно не писать: уникальность там следует из того, что поле является ключом, а обязательность следует из особенностей связи.

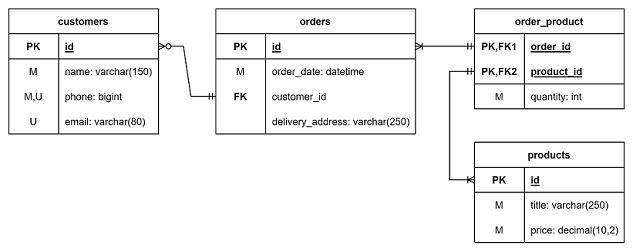
Следующий шаг в детализации схемы — указание типов данных.

Рассмотрим знакомый вам пример с небольшим дополнением



Как видите, после каждого атрибута теперь прописан тип данных, который будет использоваться в СУБД. Конкретно в примере задействованы типы MySQL.  
  
Перечень типов данных СУБД MySQL 8 представлен, например, тут <https://www.w3schools.com/mysql/mysql_datatypes.asp>

Почему типы может определять аналитик, а не программист? Например, потому, что для определения типов иногда требуется уточнять требования. Например, мы собираемся хранить имя клиента в базе. А будет ли это только имя или ФИО? А какой максимальной длины могут быть эти имена? Если с такими ограничениями аналитик определится сам, то не придётся затем возвращать задачу на уточнение от программиста.  
  
Теперь на примере интернет-магазина посмотрим, как детализировалась вся логическая модель, став физической:



Обратите внимание, что в физической модели для связи заказов и товаров пришлось ввести новую промежуточную таблицу order\_products. Без неё не удалось бы удобно реализовать связь многие ко многим. Обратите внимание на пару ключей order\_id и product\_id, каждый из которых является внешним, а вместе — они являются первичным в таблице order\_product. Кроме этих ключей в таблице ещё есть поле quantity (количество), с помощью которого мы храним информацию о том, в каком количестве товар добавлен в заказ.