**История баз данных**

История активного развития баз данных начинается с одного из самых значительных и неоднозначных событий: полета на Луну. Тогда для участия в проекте Apollo правительством США была привлечена компания Rockwell. Для того, чтобы построить космический корабль, как многие, наверное, догадываются, нужно собрать несколько миллионов деталей. И в те далекие времена была создана система управления файлами, которая отслеживала информацию о каждой детали.  
  
Но когда решили проверить эту систему, то обнаружили, что данные в ней повторяются по нескольку раз. Налицо была огромная избыточность. К сотрудничеству была привлечена небезызвестная IBM, и в 1968 году была разработана автоматизированная система IMS. Кардинальным нововведением было разделение функций деловой логики и данных, т.е. программисты смогли работать с информацией на логическом уровне, при этом база данных выполняла функцию физического хранения. Это позволило существенно повысить производительность труда.

Данная система имела следующую модель хранения данных (на рисунке представлена абстракцией!):

**Эволюция.**Следующий большой шаг в истории развития баз данных сделал доктор Эдгар Кодд (Edgar Codd) - научный сотрудник все той же самой небезызвестной IBM. В 1970 году он опубликовал свою работу "Реляционная модель для больших банков совместно используемых данных", которая в корне изменила теорию баз данных. Он предложил избавиться от иерархий и преобразовать их в разделение на логические сущности. Для каждой такой сущности подразумевалось наличие отдельной таблицы и связей между ними.

(РИСУНОК 1)

Реляционная модель данных  
В реляционной модели, придуманной Коддом, данные можно было свободно описывать в их естественном виде, без каких-либо ограничений, которые накладываются средой физического хранения. Главная особенность такой модели – зависимость всех таблиц друг от друга.

(РИСУНОК 2)

**Основные понятия.**

База данных — набор логически связанных данных, предназначенный для удовлетворения информационных потребностей.

Реляционная база данных представляет собой множество взаимосвязанных таблиц, каждая из которых содержит информацию об объектах определенного вида. Каждая строка таблицы содержит данные об одном объекте, а столбцы таблицы содержат различные характеристики этих объектов - атрибуты.

СУБД – программный комплекс для создания и администрирования (управления) базами данных.

**Распространённые СУБД**

- Oracle (~60% рынка СУБД);

- MS SQL Server (~15%);

- IBM DB (~13%);

- MySQL, PostgreSQL, SQLite, Sybase, Access и другие (~12%);

**Трехзвенная схема работы приложений**  
(РИСУНОК 3)

**SQL**

Structured Query Language – структурированный язык запросов, предназначенный для работы с данными в рамках СУБД. Язык SQL состоит из нескольких основных подмножеств:

1. Data Definition Language (язык определения данных) :

a)CREATE (создать сущность);

b)ALTER (изменить сущность);

c)DROP (удалить сущность);

d)TRUNCATE (усечь сущность);

2. Data Manipulation Language (язык управления данными) :

a)SELECT (выборка данных);

b)UPDATE (обновление данных);

c)INSERT (вставка данных);

d)DELETE (удаление данных);

3. Data Control Language (язык контроля данных) :

a)GRANT (предоставить привилегии на какую-либо команду какому-либо пользователю);

b)REVOKE (изъять привилегии у пользователя).

4. Transaction Control Language (язык управления транзакциями) :

a)COMMIT применяет транзакцию;

b)ROLLBACK «откатывает» все изменения, сделанные в контексте текущей транзакции;

c)SAVEPOINT делит транзакцию на более мелкие участки.

**SQL. Типы данных**

В базах данных у каждого столбца, локальной переменной, выражения и параметра есть определенный тип данных. Тип данных — атрибут, определяющий, какого рода данные могут храниться в объекте: целые числа, символы, данные денежного типа, метки времени и даты, двоичные строки и так далее. Ниже приведены самые распространённые типы.

Символьные типы данных - содержат буквы, цифры и специальные символы.

CHAR(n) – символьные строки фиксированной длины. Длина строки определяется параметром n. Для хранения таких данных всегда отводится n байт вне зависимости от реальной длины строки.

VARCHAR(n) – символьная строка переменной длины. Для хранения данных этого типа отводится число байт, соответствующее реальной длине строки.

Целые типы данных - поддерживают только целые числа (дробные части и десятичные точки не допускаются). Над этими типами разрешается выполнять арифметические операции и применять к ним агрегирующие функции (о них будет рассказано позже).

INTEGER или INT - целое, для хранения которого отводится, как правило, 4 байта. Интервал значений от - 2147483647 до + 2147483648

SMALLINT - короткое целое (2 байта), интервал значений от - 32767 до +32768

Вещественные типы данных - описывают числа с дробной частью.

FLOAT и SMALLFLOAT - числа с плавающей точкой (для хранения отводится обычно 8 и 4 байта соответственно).

DECIMAL(p) - тип данных аналогичный FLOAT с числом значащих (указанных после запятой) цифр p.

DECIMAL(p,n) - аналогично предыдущему, p - общее количество десятичных цифр, n - количество цифр после десятичной запятой.

Дата и время - используются для хранения даты, времени и их комбинаций. Большинство СУБД умеет определять интервал между двумя датами, а также уменьшать или увеличивать дату на определенное количество времени. DATE - тип данных для хранения даты. TIME - тип данных для хранения времени. INTERVAL - тип данных для хранения временного интервала. DATETIME - тип данных для хранения моментов времени (год + месяц + день + часы + минуты + секунды + доли секунд).

Помимо приведённых в этом перечне типов, в тех или иных СУБД существуют свои, специфические типы данных. Применение их зависит от задачи.

Программный код на языке SQL пишется в виде запросов к данным. Сам по себе язык является регистронезависимым, но стандарт языка рекомендует писать ключевые слова SQL в запросе прописными буквами или, хотя бы, в одном стиле. Общий вид запроса на выборку данных:

(РИСУНОК 4)

Например, нам нужно выбрать все записи из таблицы table\_Section, которая имеет вид:

(РИСУНОК 5)

где id и id\_Parent – целочисленные поля типа int, а Name – строчное поле типа varchar(30).

Для выборки всех данных из таблицы, напишем запрос:

SELECT \* FROM table\_Section

‘SELECT’ – это команда выборки. Затем, вместо явного указания имён колонок через запятую, пишем ‘\*’ (говорится как «звёздочка») – это символ, указывающий на выборку всех полей таблицы, ‘FROM’ – команда, которая указывает из какой именно сущности будет производиться выборка, и ‘table\_Section’ – название сущности.

Т.е., говоря , мы сообщаем СУБД, что хотим видеть выборку всех данных из таблицы table\_Section

Попробуем немного усложнить задачу, добавив условие. Допустим, теперь нам нужно вычленить записи с id больше, либо равным двум. Запрос будет иметь вид:

SELECT \* FROM table\_Sections

WHERE id >= 2

Результат:   
(РИСУНОК 6)

Мы сказали этим запросом, что хотим вычленить все данные, у которых (‘WHERE’ – ключевое слово, начало блока условий) поле id принимает значения от двух и выше.

**Логика AND**

Продолжаем изучение языка SQL и его логических операторов. Допустим, у нас есть таблица с названием Employee вида:

( РИСУНОК)

Основным оператором фильтрации данных служит ключевое слово AND, позволяющее накладывать несколько условий на выборку. Таким образом, данные должны удовлетворять всем (!) условиям, указанным в WHERE. Например, нам нужно вычленить всех IT’шников с зарплатой, больше 16 тысяч. Для этого пишем запрос вида:

SELECT name

FROM Employee

WHERE org = ‘IT’

AND salary > 16000;

Запрос выдаст нам следующее: ( РИСУНОК)

Обратите внимание на этот запрос, который мы только что написали:

SELECT name

FROM Employee

WHERE org = ‘IT’

AND salary > 16000;

В нём присутствует несколько важных аспектов. Во-первых, условие всегда начинается с блока WHERE, но данный оператор пишется в одном запросе лишь единожды. После него всегда пишется имя поля, на которое необходимо добавить условие, затем символ логики (‘>’ – строго больше, ‘<’ – строго меньше, ‘>=’ – больше или равно, ‘<=’ – меньше или равно, ‘=’ – строго равно, ‘!=’ или ‘<>’ – не равно) и только потом значение.

Во-вторых, числовые данные пишутся без кавычек, а строчные всегда в одинарных кавычках. Это очень важный момент!

В третьих, для начала нового условия применяется вспомогательный оператор AND. Никаких WHERE повторно не пишется, только дополнительный предикат!

**Локика OR**

Едем далее. Таблица остаётся той же. Теперь нас интересуют все бухгалтеры или снабженцы с зарплатой менее 8000. В этом нам поможет оператор OR, благодаря которому можно указать

несколько условий и выборка должна удовлетворять хотя бы одному из них: ( РИСУНОК)

SELECT name

FROM Employee

WHERE org = ‘Бухгалтерия’

OR (org = ‘Снабжение’ AND salary < 8000);

Запрос выдаст нам следующее: (РИСУНОК)

**Логика Between\ Distinct**

Когда мы работаем с числовым или табельным диапазоном значений, часто приходится прибегать к написанию интервальных запросов. В этом нам поможет оператор BETWEEN.

Давайте вычленим из таблицы все организации, которые платят от 10 до 30 тысяч:

(РИСУНОК)

SELECT DISTINCT org ---- DISTINCT исключит повторы!

FROM Employee

WHERE salary BETWEEN 10000 AND 30000;

Запрос выдаст нам следующее:

Важно: границы диапазона в BETWEEN будут включены в выборку!

Обратите внимание на вспомогательный оператор DISTINCT. Для того, чтобы исключить дубликаты (например, чтобы запись IT не повторялась), мы написали его перед именем поля и после оператора SELECT (строго так!). В итоге, получили уникальную выборку.

**SQL LIKE**

Очень часто возникает ситуация, когда мы хотим сделать выборку, но точных

условий задать не можем. Например, когда хотим выбрать данные по сотруднику,

но помним лишь первые несколько букв его фамилии. Для решения таких задач нам поможет оператор LIKE. Допустим, мы помним, что фамилия нужного нам человека начиналась с «Ва»:

SELECT \*

FROM Employee

WHERE name LIKE ‘Ва%’;

Символ ‘%’ означает, что после букв ‘Ва’ могут быть какие угодно символы. Их количество от 0 до N. Важно понимать, что этими символами могут быть цифры, буквы и т.д. Знак % можно использовать несколько раз. Например, так выглядит запрос, который выводит данные о сотрудниках, у которых в зарплате содержится цифра 500:

SELECT name

FROM Employee

WHERE salary LIKE ‘%500%’;

( 3 РИСУНКА)

**ЛОГИКА IN \NOT IN**

Когда условий становится много и они накладываются на одно поле, можно совместить их множеством, а выборку с оператора OR изменить на конструкцию IN или NOT IN.

К примеру, нас интересуют все сотрудники из отделов IT и Бухгалтерии:

SELECT name

FROM Employee

WHERE org IN (‘Бухгалтерия’, ‘IT’);

Запрос выдаст нам следующее:

Как не сложно догадаться, оператор IN работает похожим на OR способом, т.е., в блоке WHERE можно было бы написать “org = ‘Бухгалтерия’ OR org = ‘IT’”, но если условий много, например, 10 или 20? В таких случаях и помогает IN, который проверяет поле на каждое из значений. Совпадение найдено? Значит результат попадёт в выборку.

В предыдущем запросе заменим IN на NOT IN. Получим:

Запрос вернул нам все значения, которые НЕ(!) попали во множество (‘Бухгалтерия’, ‘IT’)

(2 РИСУНКА)

**ЛОГИКА ПОДЗАПРОСОВ**

Очень часто мы сталкиваемся с ситуацией,

когда в явном виде не знаем условий, но

знаем как эти самые условия получить

запросом. Такие ситуации порождают

использование подзапросов.

Например, нужно извлечь сотрудников,

которые работают в IT, получают зарплату

меньше 20 тысяч и идентификатор у них больше идентификатора Гришина. Вот это самое важное. Как такое написать? Пробуем:

SELECT name

FROM Employee

WHERE org = ‘IT’

AND salary < 20000

AND id > (SELECT id FROM Employee WHERE name = ‘Гришин’);

Обратите внимание: подзапрос содержится в круглых скобках. С его помощью мы находим идентификатор работника Гришина (подзапрос найдёт единицу) и это значение будет подставлено в основной запрос в виде «id > 1». Таким образом, не зная явного значения id\_Гришина, мы написали дополнительный запрос на извлечение ячейки как условия

**NULL в СУБД**

NULL в СУБД — специальное значение

(псевдозначение), которое может быть

записано в поле таблицы базы данных

(БД). NULL соответствует понятию

«пустое поле», то есть «поле, не содержащее

никакого значения» / пустое множество.

Введено для того, чтобы различать в полях

БД пустые (визуально не отображаемые) значения (например, строку нулевой длины) и отсутствующие значения (когда в поле не записано вообще никакого значения, даже пустого).

NULL означает отсутствие, неизвестность информации. Значение NULL не является значением в полном смысле слова: по определению оно означает отсутствие значения и не принадлежит ни одному типу данных. Поэтому NULL не равно ни логическому значению FALSE, ни пустой строке, ни нулю. При сравнении NULL с любым значением будет получен результат NULL, а не FALSE и не 0. Более того, NULL не равно NULL!

Представим себе ситуацию, когда Петрову решили изменить заработную плату, но пока не определились с конечной цифрой. В итоге, у него может быть в поле Salary всё что угодно – хоть 0, хоть миллион баксов, но пока неизвестно. А теперь попытаемся найти всех работников, у которых показатель заработной платы был бы пуст / неизвестен:

SELECT name

FROM Employees

WHERE salary IS NULL;

Обратите внимание: в проверке никаких знаков равенства! Только IS NULL

**Связи**

В этой презентации мы попробуем понять, как связывать таблицы друг с другом, но изначально необходимо определиться с нюансами и тонкостями этих связок. Связь позволяет моделировать отношения между объектами предметной области. Существует 3 типа связей:

1.«Один-к-одному» (1:1) - любому экземпляру сущности А соответствует только один экземпляр сущности В, и наоборот. Например (обыкновенно), у одного мужа может быть лишь одна жена и у одной жены – только один муж.

(РИСУНОК)

2. «Один-ко-многим» (1:М) - любому экземпляру сущности А соответствует 0, 1 или несколько экземпляров сущности В, но любому экземпляру сущности В соответствует только один экземпляр сущности А. Например, у поэта может быть множество стихов, но автор у них один.

(РИСУНОК)

3. «Многие-ко-многим» - любому экземпляру сущности А соответствует 0, 1 или несколько экземпляров сущности В, и любому экземпляру сущности В соответствует 0, 1 или несколько экземпляров сущности А. Например, у каждого преподавателя в университете может быть множество студентов. В свою очередь, каждого студента обучает некоторое множество преподавателей

(РИСУНОК)

**Обеспечение целостности**

В теории баз данных целостность данных означает корректность данных и их непротиворечивость. Обычно она также включает целостность связей, которая исключает ошибки связей между первичным и вторичным ключом. К примеру, когда существуют дочерние записи-сироты, которые не имеют связи с родительскими записями. В данном случае, таблица с секциями ссылается на таблицу со ступенями (M:1), но у секции B отсутствует ссылка на родительскую запись, а секция С вообще ссылается на несуществующую ступень.

(РИСУНОК)

Чтобы обеспечить целостность, работа с данными должна производиться с учетом нижеперечисленных правил:

1. Невозможно ввести в связанное поле подчиненной таблицы значение, отсутствующее в связанном поле главной таблицы. Однако можно ввести пустое значение, показывающее, что для данной записи связь отсутствует.

2. Не допускается удаление записи из главной таблицы, если существуют связанные с ней записи в подчиненной таблице.

3. Невозможно изменить значение ключевого поля в главной таблице, если существуют записи, связанные с данной таблицей

(РИСУНОК)

**Нормализация И NF**

Нормальная форма (НФ или NF) — требование, предъявляемое к структуре таблиц в теории реляционных баз данных для устранения из базы избыточных функциональных зависимостей между атрибутами (полями таблиц).

 Процесс преобразования отношений базы данных к виду, отвечающему нормальным формам, называется нормализацией. Нормализация предназначена для приведения структуры БД к виду, обеспечивающему минимальную логическую избыточность, и не имеет целью уменьшение или увеличение производительности работы или же уменьшение или увеличение физического объёма базы данных. Конечной целью нормализации является уменьшение потенциальной противоречивости хранимой в базе данных информации

1NF

Переменная отношения находится в первой нормальной форме (1NF) тогда и только тогда, когда каждое из значений его полей атомарно.

Исходная ненормализованная (тоесть не являющаяся правильным представлением некоторого отношения) таблица: Таблица, приведённая к 1NF (являющаяся правильным представлением некоторого отношения): (РИСУНОК)

2NF   
Для второй нормальной формы (2NF) таблица должна находиться в первой нормальной форме. Любое её поле, не входящее в состав первичного ключа, функционально полно должно зависеть от первичного ключа.

Например, пусть в следующем отношении первичный ключ образует пара атрибутов {Сотрудник, Должность}: (рисунок)  
  
Зарплату сотруднику каждый начальник устанавливает сам (хотя её границы зависят от должности). Наличие же компьютера у сотрудника зависит только от должности, то есть зависимость от первичного ключа неполная. В результате приведения к 2NF исходное отношение следует декомпозировать (логически разбить) на два отношения: ( РИСУНОК)

**3NF**

Для третьей нормальной формы (3NF) таблица должна находиться во второй нормальной форме. Любой её не ключевой атрибут функционально зависит только от первичного ключа.

Рассмотрим в качестве примера отношение, которое находится во 2NF, но не соответствует 3NF.

**(РИСУНОК)**

В отношении атрибут «Сотрудник» является первичным ключом. Личных телефонов у сотрудников нет, и телефон сотрудника зависит исключительно от отдела.

Таким образом, в отношении существуют следующие функциональные зависимости: Сотрудник → Отдел, Отдел → Телефон, Сотрудник → Телефон. Зависимость Сотрудник → Телефон является побочной, следовательно, отношение не находится в 3NF. В результате разделения отношения получаются два отношения, находящиеся в 3NF:

(РИСУНОК)

**SQL JOIN**

Переходим непосредственно к коду. Наша задача – научиться связывать несколько таблиц в рамках одного запроса. В этом нам поможет оператор JOIN.

Ключевое слово JOIN в SQL используется при построении select-выражений. Инструкция JOIN позволяет объединить колонки из нескольких таблиц в одну. Объединение происходит временное и целостность таблиц не нарушается. Существует три типа join-выражений:

- inner join;

- outer join;

- cross join;

В свою очередь, outer join может быть left, right и full (слова inner и outer обычно опускаются). В качестве примера возьмём две простые таблицы (таблица А с работниками ссылается на таблицу B с должностями) и сконструируем для них SQL-выражения с использованием join.

( рисунок)

**INNER JOIN**

**(рисунок)**

**SELECT \* FROM a**

**JOIN b**

**ON a.id\_B = b.id ( рисунок)**

Обратите внимание на диаграмму множеств. Внутреннее соединение INNER JOIN производит выборку только тех записей, которые соответствуют пересечению таблиц А и В. Работник «Иванов» из таблицы А не нашёл себе соответствий в таблице В, также, как и вид должности «Охрана» из таблицы В не привязан ни к одному работнику, поэтому данные значения не попали в конечный результат выборки.

**LEFT OUTER JOIN**

**(рисунок)**

**SELECT \* FROM a**

**LEFT JOIN b**

**ON a.id\_B = b.id (рисунок)**

**RIGHT OUTER JOIN**

**(рисунок)**

**SELECT \* FROM a**

**RIGHT JOIN b**

**ON a.id\_B = b.id ( рисунок)**

Правостороннее внешнее соединение действует по аналогии с LEFT JOIN’ом. RIGHT OUTER JOIN производит полный выбор записей из таблицы, которая указана после слов RIGHT JOIN (в данном случае из В) с соответствующими записями (если таковые имеются) таблицы А. Если совпадения нет, то левая часть будет содержать NULL.

**FULL OUTER JOIN**

(рисунок)

SELECT \* FROM a

FULL JOIN b

ON a.id\_B = b.id ( рисунок)

Полное внешнее соединение FULL OUTER JOIN производит выборку множества всех записей из таблицы А и B с соответствующими записями с обеих сторон при их наличии. Если совпадения нет, отсутствующая сторона будет содержать NULL. Обратите внимание на две последних строки в итоговой выборке.

**CROSS JOIN**

**(рисунок)**

Также существует выборка перекрестного соединения (называемое ещё декартовым произведением), - CROSS JOIN, с перебором все вариантов, которое сложно объяснить диаграммами:

SELECT \* FROM a

CROSS JOIN b;

---(ещё можно написать запрос как

---SELECT \* FROM a, b)

Данное перекрестное соединение выбирает буквально "всё ко всему", в результате мы

получим 4 x 4 = 16 записей, т.е. намного больше, чем в оригинале мы имеем в таблицах