ЦИФРОВАЯ КУЛЬТУРА
УНИВЕРСИТЕТ ИТМО

Объекты базы данных

Высшая Школа Цифровой Культуры
Университет ИТМО
dc@itmo.ru

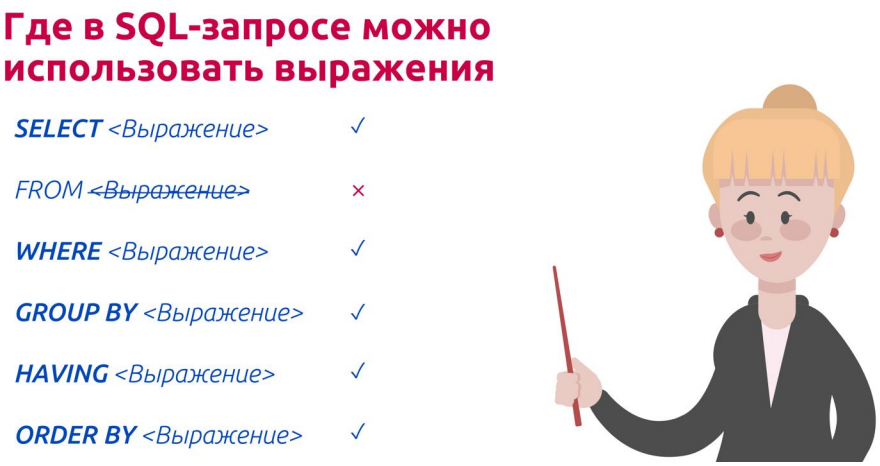
Содержание

1 Выражения на SQL

В прошлой лекции мы рассмотрели, как составляются запросы к данным. Мы уже упоминали о том, что из названий столбцов, констант, математических и логических операций можно составлять выражения. Давайте рассмотрим это подробнее:

- Константами являются значения любого типа данных, используемые в СУБД. Строковые константы и даты принято указывать в кавычках, обычно одинарных.
- Вместо названия столбцов для каждой строки в выражение подставляются соответствующие значения.
- Круглые скобки помогают изменить приоритет операций.
- Операции сравнения – это знаки равно, не равно, меньше, меньше равно и т.д.
- Логические операции – это операции AND (И), OR (ИЛИ), NOT (Отрицание).
- SQL поддерживает стандартные арифметические операции (сложение, вычитание, умножений, деление и остаток от деления).
- Для конкатенации строк используется либо знак ||, либо +.
- Кроме того, в выражениях могут участвовать функции. В любой СУБД есть большое количество встроенных функций, позволяющих обрабатывать данные.

Давайте определимся, где в запросе могут быть использованы выражения.



Выражения могут быть использованы для указания выводимых значений после слова `SELECT`. Простейшими выражениями являются имена столбцов и константы. Например, запрос, который выводит номер зачетки и имя студента из таблицы `Students`. В этом запросе два простых выражения – имена столбцов:

```
SELECT StudentId, StudentName FROM Students
```

Добавим еще одно выражение – третий столбец. Выведем для всех строк одно единственное значение `1 year student`:

```
SELECT StudentId, StudentName,
'1 year student' AS Status FROM Students
```

Следующий запрос составлен при помощи операции конкатенации строк – мы добавляем к имени студента строковую константу:

```
SELECT StudentId,
StudentName || '1 year student' AS Status FROM Students
```

Теперь переведем оценку из пятибалльной шкалы в стобальную. Выведем данные о результатах каждого студента по каждому экзамену, умножив оценку на 100 и поделив на 5:

```
SELECT StudentId, ExamSheetId, Grade*100/5 AS SCORE
FROM Exam_Result
```

Все рассмотренные примеры не использовали агрегирования, и результатом выборки является множество строк. В таком случае выражение вычисляется для каждой строки.

Выражения могут использоваться как параметр функции агрегирования. Если функция агрегирования вычисляется от заданного выражения, то сначала будет найдено значение выражения для каждой строки, а затем полученные результаты проагрегированы.

```
SELECT StudentId, SUM(Grade*100/5) AS Total
FROM Exam_Result GROUP BY StudentId
```

Функции агрегирования также могут входить в составленное выражение. Например, можно изменить предыдущий запрос, если сначала вычислить общий балл для каждого студента, а затем результат перевести в 100-балльную шкалу.

```
SELECT StudentId, SUM(Grade)*100/5 AS Total
FROM Exam_Result GROUP BY StudentId
```

После слова `WHERE` в запросе указывают критерий выбора строк. Простейший критерий выборки – это логическое выражение. Логическое выражение выдает результат Истина или Ложь. Простые логические выражения в условии составляются из имен столбцов, констант и операторов сравнения, например:

```
SELECT * FROM Students WHERE StudentId < 345569

SELECT * FROM Students WHERE GroupCode <> 'M2020_1'
```

Со значением столбца в выражении можно производить арифметические действия. Например, перепишем простой запрос – результат будет ровно такой же. Конечно, выражения могут быть и более сложными.

```
SELECT * FROM Students WHERE StudentId+1 < 345570
```

Впрочем, хороший совет – чтобы методы оптимизации, которые есть в каждой СУБД, могли работать эффективно, следует использовать после `WHERE` значения столбцов без всяких преобразований, а вот сравнивать их можно с результатами других выражений.

Более сложные выражения могут содержать в себе логические операции, связывающие несколько условий. Условия связываются операторами `AND`, `OR`, `NOT`, например:

```
SELECT * FROM Students
WHERE Address LIKE '% Station Road%' AND Birthdate >= '01/01/1999'

SELECT * FROM Students
WHERE StudentId < 345574 OR StudentId > 345586
```

Мы привыкли оперировать бинарной логикой, когда каждое логическое выражение имеет результат Истина или Ложь. Часто их обозначают 1 – истина, 0 – ложь. С отрицанием все просто, эта операция имеет всего один операнд. Отрицание истины дает ложь, и наоборот, отрицание от лжи равно истине.

Логическое И и логическое ИЛИ действуют от двух операндов. Логическое И, или логическое умножение, дает 1 только в том случае, если оба операнда равны 1. А в операции логическое ИЛИ, или логическое сложение, 1 получается, если хоть один из операндов равен 1.

В базах данных к значениям Истина и Ложь добавляется еще одно – NULL, или неопределенное значение. Что произойдет теперь с результатами логических операций? Операция отрицания от неопределенное значения даст

Двузначная логика

1 - истина, 0 - ложь

- ***NOT*** — отрицание

X	***NOT*** *X*
1	0
0	1

- ***AND*** — логическая операция «И», логическое умножение
- ***OR*** — логическая операция «ИЛИ», или логическое сложение

X	*Y*	*X* ***AND*** *Y*	*X* ***OR*** *Y*
1	1	1	1
1	0	0	1
0	1	0	1
0	0	0	0

в результате тоже неопределенное. Ведь мы не знали, какое было значение до операции отрицания, не знаем его и после операции.

Логическое умножение: если умножить неопределенное значение на Истину, то результат становится неопределенным, а если умножить на неопределенное значение Ложь, то результат будет Ложь. Логическое сложение: если хоть один из операндов равен Истине, то и результат будет Истина. А вот если один из операндов Ложь, а другой – неопределенное значение, то в результате получим неопределенное значение.

Трехзначная логика

1 - истина, 0 - ложь, null - неопределенное значение

- ***NOT*** — реверсирование результата логического выражения (условия)

X	***NOT*** *X*
1	0
0	1
null	null

- ***AND*** — логическая операция «И», логическое умножение
- ***OR*** — логическая операция «ИЛИ», или логическое сложение

X	*Y*	*X* ***AND*** *Y*	*X* ***OR*** *Y*
1	1	1	1
1	0	0	1
1	null	null	1
0	1	0	1
0	0	0	0
0	null	0	null
null	1	null	1
null	0	0	null
null	null	null	null

Давайте посмотрим расписание экзаменов – это записи таблицы `ExamSheet`. Обратите внимание, что для 10 экзамена аудитория еще не определена – прочерком отображаются неопределенные значения.

EXAMSHEETID	GROUPCODE	COURSEID	TEACHERID	CLASSROOM	EXAMDATE
1	M2020_1	1	1	2408	25.01.2020
2	M2020_1	2	2	2408	27.01.2020
3	M2020_1	3	3	2410	28.01.2020
4	B2020_1	1	4	2411	26.01.2020
5	B2020_1	2	5	2412	24.01.2020
6	B2020_1	3	1	2410	25.01.2020
7	B2020_1	4	2	2410	27.01.2020
8	B2020_2	1	3	2410	27.01.2020
9	B2020_2	2	3	2411	28.01.2020
10	B2020_2	3	4	-	28.01.2020

10 rows returned in 0.01 seconds Download

Найдем экзамены, которые проводятся в аудитории 2411. В результате видим всего две строки.

```
SELECT * FROM ExamSheet WHERE ClassRoom = 2411
```

EXAMSHEETID	GROUPCODE	COURSEID	TEACHERID	CLASSROOM	EXAMDATE
4	B2020_1	1	4	2411	26.01.2020
9	B2020_2	2	3	2411	28.01.2020

2 rows returned in 0.01 seconds Download

Теперь найдем экзамены, которые проводятся не в аудитории 2411. В результате видим еще 7 строк.

```
SELECT * FROM ExamSheet NOT ClassRoom = 2411
```

EXAMSHEETID	GROUPCODE	COURSEID	TEACHERID	CLASSROOM	EXAMDATE
1	M2020_1	1	1	2408	25.01.2020
2	M2020_1	2	2	2408	27.01.2020
3	M2020_1	3	3	2410	28.01.2020
5	B2020_1	2	5	2412	24.01.2020
6	B2020_1	3	1	2410	25.01.2020
7	B2020_1	4	2	2410	27.01.2020
8	B2020_2	1	3	2410	27.01.2020

7 rows returned in 0.01 seconds Download

Как же нам увидеть 10-ю? Сформулируем вопрос чуть иначе: как найти экзамены, которые МОГУТ проводиться в аудитории 2411. Вот теперь мы видим три строки: записи, где аудитория равна 2411, или аудитория не определена.

```
SELECT * FROM ExamSheet
MINUS
SELECT * FROM ExamSheet NOT ClassRoom = 2411
```

EXAMSHEETID	GROUPCODE	COURSEID	TEACHERID	CLASSROOM	EXAMDATE
4	B2020_1	1	4	2411	26.01.2020
9	B2020_2	2	3	2411	28.01.2020
10	B2020_2	3	4	-	28.01.2020

3 rows returned in 0.01 seconds Download

Выражение может быть критерием группировки. Обычно, если нужно группировать по нескольким столбцам, их имена указывают через запятую, но можно составить из них и одно выражение. Ниже простой пример группировки с выражением от одного столбца `GRADE`.

```
SELECT COUNT(*) AS Amount, Grade*100/5 AS SCORE
FROM Exam_Result GROUP BY Grade*100/5
```

При группировке строк можно использовать условия на выборку групп. Например, сгруппируем результаты экзаменов для каждого студента и найдем хорошистов – это те, у кого нет оценок меньше 4.

```
SELECT StudentId FROM EXAM_RESULT GROUP BY StudentId
HAVING MIN(Grade) >= 4
```

Давайте считать, что если человек сдал только один экзамен, то его еще нельзя называть хорошистом, его оценка могла быть случайностью.

```
SELECT StudentId FROM EXAM_RESULT GROUP BY StudentId
HAVING MIN(Grade) >= 4 AND COUNT(*) > 1
```

Еще один запрос демонстрирует список «стабильных» студентов – тех, кто сдал все экзамены на одну оценку:

```
SELECT StudentId FROM EXAM_RESULT GROUP BY StudentId
HAVING MAX(Grade) - MIN(Grade) = 0
```

Предложение `ORDER BY` используется для сортировки результатов выборки. Мы уже использовали сортировку по возрастанию, по убыванию, по одному полю и по нескольким полям. Сортировку можно производить и по выражениям, в которых используются поля выборки. Например, произведя конкатенацию двух полей:

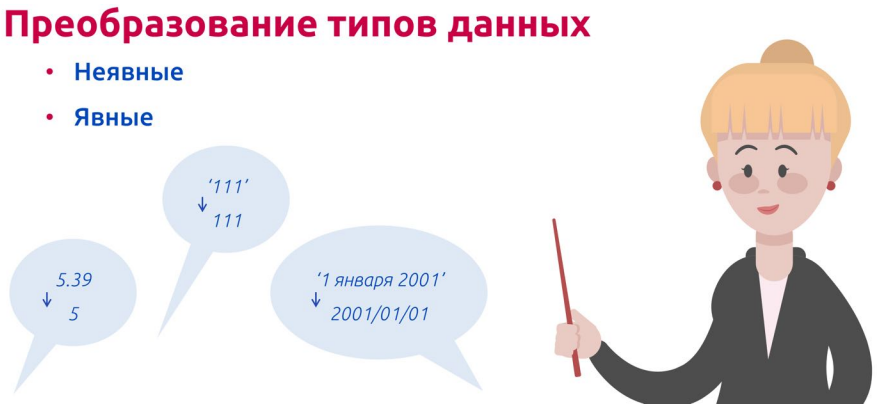
```
SELECT StudentName, GroupCode FROM Students
ORDER BY StudentName || GroupCode
```

Впрочем, сортировка с использованием выражений является достаточно экзотическим видом сортировки, и навряд ли такой запрос будет исполнен быстро на большом объеме данных.

2 Встроенные функции на SQL

Мы обсудили разные компоненты для составления выражений, теперь поговорим про функции. В любой СУБД есть большое количество встроенных функций, позволяющих обрабатывать данные.

Начнем с функций преобразования типов данных. В ходе выполнения запроса иногда требуется преобразование данных из одного типа в другой. Преобразование может выполняться двумя способами: явно и неявно. Явное преобразование выполняется, когда вызывается соответствующая функция.



Как правило, выражение не может содержать значения данных разных типов, а если все-таки они там оказались, то исполнитель запросов попробует самостоятельно выполнить преобразование данных. Такое преобразование называется неявным. Например, при делении целого числа на вещественное в результатом будет вещественное число. Тут никаких разночтений.

```
SELECT 7/2.0 FROM DUAL
```

СУБД	Результат
Oracle	3.5
PostgreSQL	3.5
MS SQL Server	3.5

А если делить целое на целое, такой запрос уже отрабатывает по разному. Oracle выдаст результат как дробное число, а PG и MS SQL Server оставят целое значение.

```
SELECT 7/2 FROM DUAL
```

СУБД	Результат
Oracle	3.5
PostgreSQL	3
MS SQL Server	3

Некоторые СУБД при арифметических действиях со строкой, состоящей из цифр, преобразуют ее к числу,

```
SELECT '1' + 3 FROM DUAL
```

СУБД	Результат
Oracle	4
PostgreSQL	4
MS SQL Server	4

Некоторые не исполняют такой запрос, выдавая сообщение об ошибке. Так что иногда результат неявного преобразования может оказаться неожиданным. Лучше по возможности использовать явное преобразование данных, чтобы результат выражения был гарантирован.

```
SELECT '1' + '3' FROM DUAL
```

СУБД	Результат
Oracle	4
PostgreSQL	Error
MS SQL Server	'13'

Явное преобразование выполняется, когда вызывается соответствующая функция. Например, нужно преобразовать число в строку, или строку в дату. Самая популярная и универсальная функция преобразования данных называется `CAST`. У функции `CAST` два параметра – значение, которое надо преобразовать, и тип данных, к которому надо преобразовать. Например, преобразуем строку с символом `5` в целое число.

```
CAST ('5' AS INTEGER);
```

Или из целого в дробное:

```
CAST (5 AS decimal(3,2));
```

Или, наоборот, из числа сделаем строку:

```
CAST (5 AS CHAR(7));
```

Но не всегда можно произвести преобразование данных. Например, строку `a5` нельзя преобразовать в число, такое преобразование не может быть выполнено.

```
CAST ('a5' AS INTEGER) -- НЕ может быть выполнено!
```

Применить функцию в запросе очень просто – напишем запрос, который возвращает значения поля `StudentId` из таблицы `Students`, преобразованные к строковому типу:

```
SELECT 'Номер зачетки - ' || CAST (StudentId AS CHAR(7))
FROM Students
```

Можно вызывать функцию и от константы. В СУБД Oracle после этого нужно еще указать имя служебной таблицы DUAL.

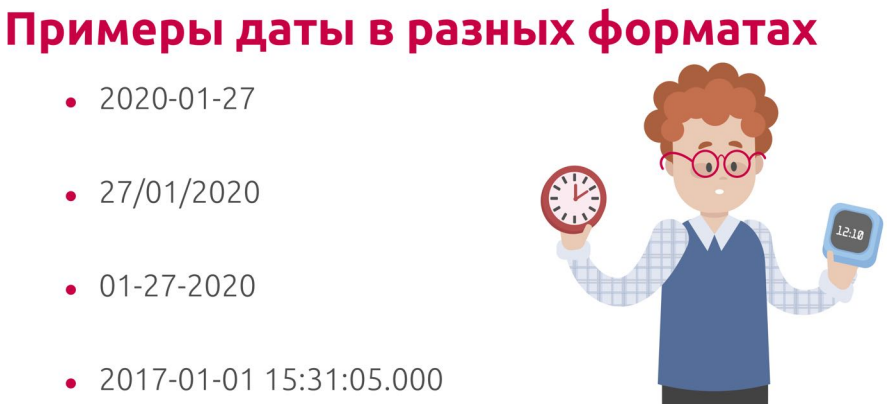
```
SELECT CAST (StudentId AS CHAR(7)) FROM Students

SELECT CAST (5 AS CHAR(7));

SELECT CAST (5 AS CHAR(7)) FROM DUAL;
```

Больше всего трудностей возникает при обработке данных в формате дата. В разных странах принято представлять дату в различных форматах. И в СУБД внешнее представление даты сильно зависит от системных настроек, и, более того, может быть настроено для разных пользователей базы по разному. Дата может представляться в американском формате (месяц, день, год) и европейском (день, месяц, год).

Форматы даты

Страна/язык	Формат даты
Россия	DD.MM.YYYY
США	MM-DD-YYYY
Международный английский	DD-MM-YYYY
Великобритания	DD/MM/YYYY
Германия	DD.MM.YYYY
Бельгия	DD/MM/YYYY
Дания	DD-MM-YYYY
Италия	DD/MM/YYYY
Испания	DD/MM/YYYY
Финляндия	YYYY-MM-DD

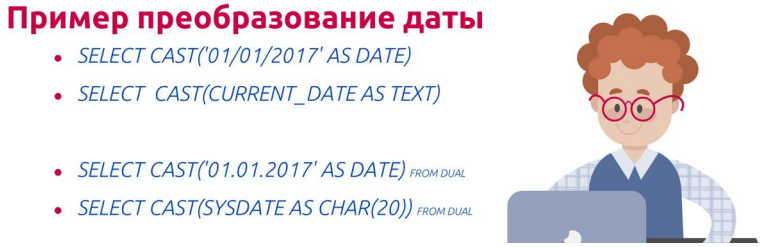
Могут различаться разделители между днем, месяцем и годом (слеши, точки, дефисы и т.п.). Год может быть записан полностью или в виде двух последних цифр. Месяц может быть записан цифрами или словами, а время может быть записано 12-часовом или 24-часовом формате. В некоторых СУБД в формат даты входит еще и время.



Чтобы увидеть формат даты, используемый в текущих настройках СУБД, проще всего вызвать функцию, показывающую текущую дату.

Oracle	*SELECT SYSDATE FROM DUAL*
PostgreSQL	*SELECT CURRENT_DATE*
MS SQL Server	*SELECT GETDATE()*

Часто используются операции преобразования даты в строку и наоборот. Такое преобразование можно выполнить с помощью функции `CAST`.



В СУБД Oracle и PostgreSQL традиционно используют функции преобразования данных `TO_date` и `TO_char`. Для того, чтобы четко понимать, как следует интерпретировать строку, представляющую дату, используют функцию `TO_date`. Это функция от двух параметров, первый параметр – строка, представляющая дату, а вторая – формат даты.

```
SELECT TO_date('28/02/2020 18:24', 'DD/MM/YYYY HH24:MI') FROM DUAL

SELECT TO_char(date_field, 'YYYY-MM-DD') FROM DUAL
```

Кроме того, есть и обратная операция, которая позволяет явно задать форму представления даты в виде строки и не рассчитывать на те параметры, которые по умолчанию заданы для сессии пользователя.

С датами и временем можно работать в любой СУБД. Однако операции и функции, которые для этого предназначены, могут сильно отличаться в разных СУБД. Приведем примеры нескольких операций и функций, которые применимы, по крайней мере в PostgreSQL и ORACLE к значениям типа `DATE`. Фактически для хранения даты используется число, которое соответствует количеству дней, прошедших от некоторой даты, принятой за начало отсчета.

К дате можно прибавить целое или вещественное число и оно будет интерпретироваться как добавление количества дней. Например, Если прибавить 1 день к 28 февраля в невисокосный год, то результатом будет первое марта. Из даты можно вычесть целое число и эта операция будет интерпретироваться как уменьшение даты на определенное количество дней.

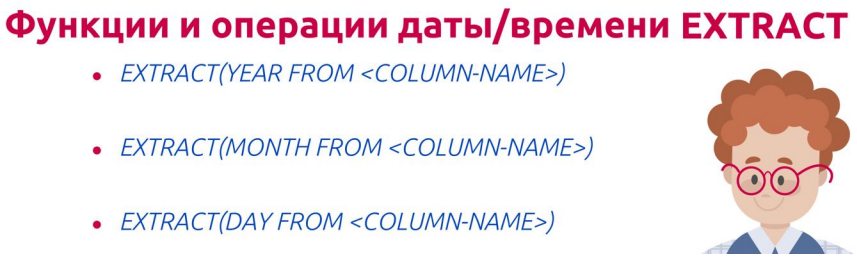
```
SELECT TO_DATE('28/02/2019', 'DD/MM/YYYY') + 1 FROM DUAL
```

Из одной даты можно вычесть другую и получить количество дней между этими датами (возможно вещественное). Например, можно вычислить максимальную разницу в днях между датами рождения студентов.

```
SELECT MAX(BirthDate) - MIN(BirthDate) FROM Students
```

Причем при этих операциях будут учитываться особенности, связанные с переходом на григорианский календарь. Напомним, что григорианский календарь был введён папой римским Григорием XIII в католических странах 4 октября 1582 года взамен прежнего юлианского: Следующим днём после четверга 4 октября стала пятница 15 октября. Эта особенность учитывается в операциях плюс и минус при работе с датами. Пример на экране демонстрирует эту коллизию, когда к 4 октября 1582 года добавляется один день, а в результате получается 15 октября 1582 года.

И, наконец, весьма полезной функцией при работе с датами является функция `EXTRACT`, которая позволяет из даты извлечь год, месяц или день. Например, можно узнать, какой сейчас месяц.




```
SELECT EXTRACT(MONTH FROM SYSDATE) FROM DUAL
```

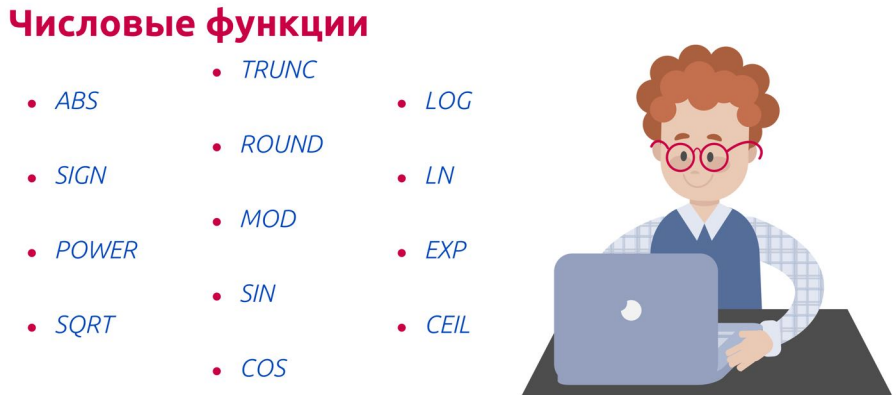
Или найти всех студентов, родившихся в мае.

```
SELECT * FROM STUDENTS WHERE EXTRACT(MONTH FROM BirthDate) = 5
```

Результат выборки можно отсортировать по числам месяца.

```
SELECT * FROM STUDENTS WHERE EXTRACT(MONTH FROM BirthDate) = 5
ORDER BY EXTRACT(DAY FROM BirthDate)
```

В любой СУБД есть математические функции, которые позволяют оперировать числовыми данными. Можно возводить в степень и извлекать квадратный корень, округлять и вычислять абсолютное значение, определять знак и находить значения тригонометрических функций, и еще много разных полезных вычислений.



Например, вычислим средний балл каждого студента и округлим по математическим правилам при помощи функции `ROUND`:

```
SELECT ROUND(AVG(Grade)), StudentId
FROM EXAM_RESULT GROUP BY StudentId
```

Затем возьмем от каждого результата только целую часть – функция `TRUNC` выполняет округление вниз.

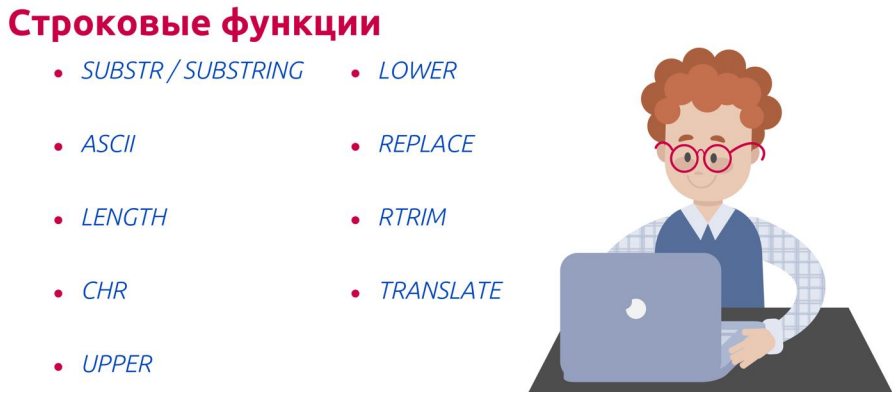
```
SELECT TRUNC(AVG(Grade)), StudentId
FROM EXAM_RESULT GROUP BY StudentId
```

А функция CEIL производит округление вверх – до ближайшего целого числа.

```
SELECT CEIL(AVG(Grade)), StudentId
FROM EXAM_RESULT GROUP BY StudentId
```

Округление можно производить не только до целого числа, а явно указывать требуемую точность вторым параметром функции.

Для работы со строками в СУБД есть строковые функции. Эти функции позволяют извлекать подстроку, менять регистр, убирать пробелы и многое другое. Узнать код символа можно при помощи функции ASCII:




```
SELECT ASCII('A') FROM DUAL
```

Символ по его коду вернет функция CHR:

```
SELECT CHR(66) FROM DUAL
```

Чтобы извлечь подстроку из заданной строки, нужно использовать функцию SUBSTR, указав строку, первый символ подстроки и ее длину.

```
SELECT SUBSTR(StudentName, 6, 5) FROM Students
WHERE StudentId = 345569
```

Функция INSTR находит вхождение в строку заданной подстроки. Например, найдем пробел, разделяющий имя и фамилию в имени студента.

```
SELECT StudentName, INSTR(StudentName, ' ', 1)
FROM Students
```

Запрос, приведенный ниже, позволяет поменять местами имя и фамилию в поле StudentName.

```
SELECT StudentName,
SUBSTR(StudentName,INSTR(StudentName, ' ')) || ' ' ||
SUBSTR(StudentName, 1, INSTR(StudentName, ' ')) FROM Students
```

3 Объекты базы данных

Все данные, которые может видеть пользователь базы данных, хранятся в таблицах. Мы научились создавать таблицы, заполнять их значениями и находить нужную информацию, задавая критерии поиска. Но кроме таблиц в базе данных хранится множество полезных объектов. Эти объекты не хранят непосредственно данные, но отражают бизнес-логику информационной системы. Для того, чтобы создавать такие объекты, надо обладать соответствующими правами доступа к базе. Для получения целостного представления о механизмах СУБД давайте немного о них поговорим. Для поиска инфор-

мации запросы могут быть очень большими и сложными, иметь несколько уровней вложенности и проверять множество условий. Чтобы не писать такие запросы повторно, существуют представления.

Представление – это поименованный запрос, сохраненный в базе данных.

Давайте представим, нам нужно найти информацию об экзаменах на ближайшую неделю. Для этого можно написать запрос, который собирает данные из трех таблиц: `Exam_SHEET`, в которой хранится информация об экзамене, COURSE с данными об изучаемых предметах, и `TEACHER`, где хранится информация об учителях. Например, такой:

```
SELECT ExamSheetId AS ExamID, CourseTitle, TeacherName,
ClassRoom, ExamDate FROM EXAM_SHEET JOIN COURSE
ON EXAM_SHEET.CourseId = COURSE.CourseId
JOIN TEACHER ON EXAM_SHEET.TeacherId =TEACHER.TeacherId
WHERE ExamDate > SYSDATE AND ExamDate < SYSDATE + 7
```

Выполнив запрос, получим нужную информацию. А что если через какое-то время возникла необходимость снова исполнить такой запрос. Писать его заново? Чтобы этого избежать, часто выполняемому запросу можно дать название и сохранить в базе данных, создав таким образом представление.

Чтобы создать представление, нужно перед запросом написать слова `CREATE VIEW`, указать название, `AS`, далее написать запрос к данным. После

EXAMID	COURSETITLE	TEACHERNAME	CLASSROOM	EXAMDATE
8	Data storage	David Greenwood	2410	27.01.2020
4	Data storage	Paul Hill	2411	26.01.2020
1	Data storage	Christopher Mills	2408	25.01.2020
9	Machine Learning	David Greenwood	2411	28.01.2020
5	Machine Learning	Charles Spencer	2412	24.01.2020
2	Machine Learning	Mary James	2408	27.01.2020
10	Data processing and analysis	Paul Hill	-	28.01.2020
6	Data processing and analysis	Christopher Mills	2410	25.01.2020
3	Data processing and analysis	David Greenwood	2410	28.01.2020
7	Artificial intelligence	Mary James	2410	27.01.2020

выполнения этой команды мы не получим вывод данных на экран, а лишь уведомление, что команда выполнена. Теперь в базе появился новые объект - представление `Exam_next_week`. Представление сохраняет лишь запрос в виде текста, а не копирует данные из таблиц. Представление еще называют виртуальной таблицей. Все его столбцы должны быть явно поименованы и иметь различные названия.

```
CREATE VIEW Exam_next_week AS
SELECT ExamSheetId AS ExamID, CourseTitle, TeacherName,
ClassRoom, ExamDate FROM EXAM_SHEET JOIN COURSE
ON EXAM_SHEET.CourseId = COURSE.CourseId
JOIN TEACHER ON EXAM_SHEET.TeacherId =TEACHER.TeacherId
WHERE ExamDate > SYSDATE AND ExamDate < SYSDATE + 7
```

Теперь можно его использовать, как обычную таблицу, например:

```
SELECT * FROM Exam_next_week
```

Наверное, вы догадались, что имя представления не может совпасть с именем таблицы в базе данных. Когда в запросе возникает имя представления, на это место подставляется сохраненный под этим именем запрос.

Кроме удобства в повторении сложного запроса, представления могут добавить еще один уровень защиты данных (например, можно создать представление для таблицы, где пользователю, выполняющему запросы, видна только определенная часть данных). Кроме того представления могут скрывать сложность данных, комбинируя нужную информацию из нескольких таблиц. Представления могут скрывать настоящие имена столбцов, порой трудные для понимания, и показывать более простые имена, которые и могут быть использованы приложением, работающим с базой данных. Кроме того, если меняется структура таблицы или названия ее полей, то за счет использования представлений можно оставить функции бизнес-логики приложения без изменения.

Возможности представлений

```
CREATE VIEW Exam_next_week AS
SELECT ExamSheetId AS ExamID, CourseTitle, TeacherName,
ClassRoom, ExamDate
FROM EXAM_SHEET JOIN COURSE
ON EXAM_SHEET.CourseId = COURSE.CourseId
JOIN TEACHER ON  EXAM_SHEET.TeacherId =TEACHER.TeacherId
WHERE ExamDate>SYSDATE AND ExamDate<SYSDATE+7
```

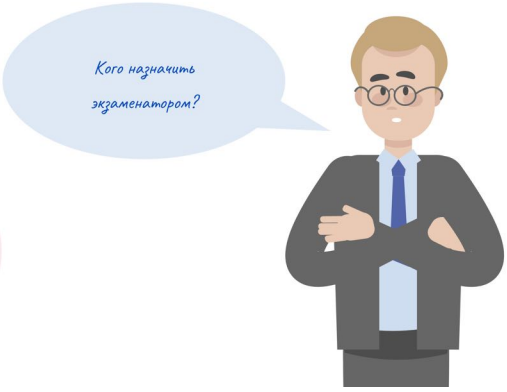
```
CREATE VIEW Exam_next_week AS
SELECT ExamSheetId AS ExamID, CourseTitle, TeacherName,
ClassRoom, ExamDate
FROM EXAM_SHEET JOIN COURSE
ON EXAM_SHEET.CourseId = COURSE.CourseId
JOIN TEACHER ON  EXAM_SHEET.TeacherId =TEACHER.TeacherId
WHERE ExamDate>SYSDATE AND ExamDate<SYSDATE+7
```

Но функции бизнес-логики трудно реализовать одним запросом. Часто нужно найти данные, проверить какие-то условия, изменить найденные данные или создать новые, внести изменения в базу данных. Например, назначить экзамен по некоторому предмету на определенную дату. Нужно найти

Процедуры и функции

TEACHER

TeacherId	TeacherName
1	Christopher Mills
2	Mary James
3	David Greenwood
4	Paul Hill
5	Charles Spencer



свободную аудиторию, проверить, что в этот день у группы нет другого экзамена, назначить преподавателя, и только тогда занести информацию в базу данных. Тут одной командой никак не обойтись.

В период экзаменационной сессии такую последовательность действий надо выполнять много раз. Для того, чтобы объединить несколько команд, решающих какую-то функцию бизнес-логики, и сохранить ее в базе данных, там же, где хранятся данные, можно создавать хранимые процедуры и определяемые пользователем функции.

Для того, чтобы создать такие процедуры и функции, уже недостаточно команд, которые входят в стандарт языка SQL. Могут понадобиться переменные, операторы цикла, условные операторы и т.д. Для этого в каждой СУБД существует набор команд, который называют процедурным расши-

рением, дополняющих возможности стандартных операторов языка SQL. В

разных СУБД механизм процедур-функций представлен по-разному, но как правило, процедуры могут изменять данные в таблицах, а функции возвращают результат в виде значения или таблицы.

Давайте рассмотрим простую функцию, которая по номеру группы выдает в качестве результата список студентов этой группы, разделенных запятыми. Любой объект создается командой `CREATE`, далее указываем тип объекта – функция, затем имя нового объекта. В круглых скобках укажем имя и тип параметра, номер группы имеет строковый тип. После этого опишем возвращаемый результат: строка переменной длины до 1000 символов, заведем для этого переменную `varTemp`. Теперь опишем само тело функции: инициализируем значение переменной, записав туда название группы и двоеточие, далее в цикле формируем значение переменной `varTemp`.

```
create function GetStudentsList(P_GroupCode in varchar) return varchar
is varTemp varchar(1000);
begin
varTemp:=P_GroupCode || ': ' ;
```

```
for rec in (select StudentName FROM Students
where GroupCode = P_GroupCode ) loop
      varTemp := varTemp  || rec.StudentName ||  ' , ';
    end loop;
return varTemp;
end;
```

Чтобы вызвать эту функцию, составим запрос, указав в качестве искомого выражения имя функции и ее параметр. Можно вызвать ее с параметром в виде константы, а можно исполнить функцию сразу для всех названий групп из таблицы `ST_GROUP`:

```
select GetStudentsList (GROUPCODE) from ST_GROUP
```

Созданные процедуры или функции хранятся в базе данных и могут вызваны из любой прикладной программы. Функции могут участвовать в выражениях, точно так же, как и стандартные, встроенные функции. Процедуры могут быть вызваны либо с помощью команды `EXECUTE` (MS SQL Server, Oracle), либо командой `CALL` (My SQL), либо после слова `SELECT` в PostgreSQL.

```
CREATE PROCEDURE ADD_EXAM (p_GroupCode VARCHAR,
p_CourseName VARCHAR, p_ExamDate DATE)
...

EXECUTE ADD_EXAM('B2020_1', 'Text processing',
To_Date('2020-01-19', 'YYYY-MM-DD'))
```

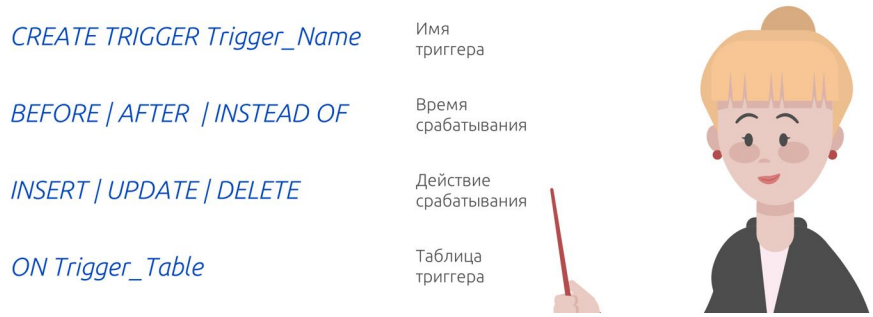
Есть особые процедуры, которые не вызываются в явном виде, а срабатывают при наступлении определенных событий в базе данных. Эти процедуры называются триггерами. Такие процедуры нужны либо для того, чтобы предотвратить наступление некоторых событий в базе данных, либо отреагировать на их наступление и инициировать другие события.

На какие события могут реагировать триггеры? На изменение данных в таблице – добавление новых строк, модификацию или удаление существующих, на создание новых таблиц в базе данных, на доступ какого-то пользователя к базе данных. При этом триггеры могут срабатывать перед наступлением заданного события или после него.

Еще триггеры могут применяться для: обеспечения сложного протоколирования; обеспечения ссылочной целостности (если этого нельзя сделать средствами правил целостности); задания сложных правил целостности; обеспечения контроля над некоторыми событиями и многих других задач.

Давайте рассмотрим пример. Информация о назначенных экзаменах хранится в таблице `EXAM_SHEET`. Чтобы назначить дату экзамена для группы по определенному предмету, нужно добавить строку в эту таблицу.

Ограничениями целостности проверяется, что у экзаменационной ведомости должен быть уникальный номер, ссылки на предмет и преподавателя берутся из соответствующих таблиц. Но кроме этого можно проверить много других условий, которые нельзя указать в правилах целостности. Например, что у группы не назначен другой экзамен в этот день, что аудитория свободна, день экзамена не является выходным и пр. И если эти условия нарушены, то назначать экзамен с такими параметрами нельзя. Проверить выполнение



этих условий поможет триггер. Давайте рассмотрим совсем простой пример - напишем триггер, который будет проверять, что дата экзамена не является воскресеньем. Триггер будет срабатывать на добавление строк в таблице `EXAM_SHEET` и срабатывать до наступления событий. В теле триггера из даты экзамена будем извлекать день недели и проверять, и если он оказался воскресным, то триггер вызовет прерывание и запись не будет добавлена в базу данных.

```
create trigger EXAM_SHEET_T
```

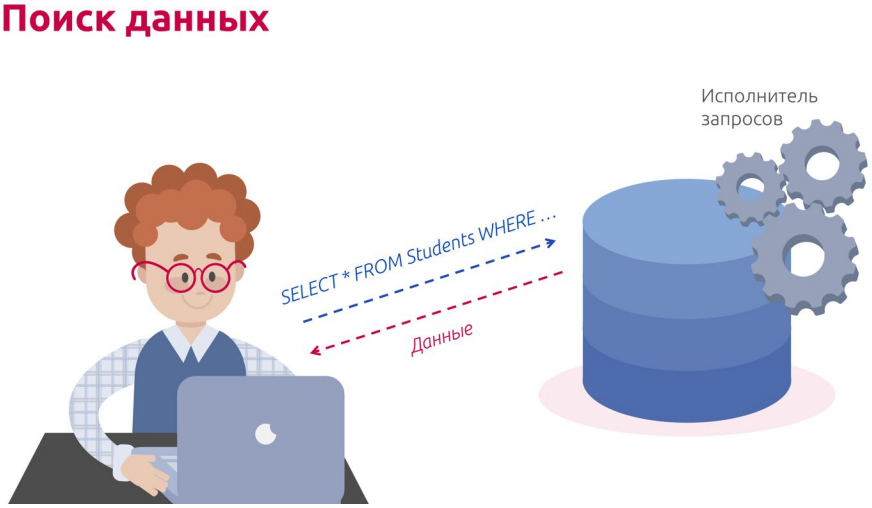
```
before insert on EXAM_SHEET
for each row
declare
not_on_weekends exception;
begin
   if to_char(:new.ExamDate, 'DY') = 'SUN') then
    RAISE not_on_weekends; end if;
   exception
   when not_on_weekends then
     raise_application_error(-20324,
     'You can't schedule an exam for Sunday!');
end;
```

Таким образом мы рассмотрели объекты, которые вместе с таблицами хранятся в базе данных и обеспечивают обработку данных. В следующем фрагменте мы поговорим о еще одном виде объектов – индексах.

4 Индексы

Запросы к данным пишутся на языке SQL, который является языком высокого уровня. Мы пишем команду, указываем, из каких таблиц надо извлечь данные, задаем нужные параметры, и СУБД извлекает нам требуемую информацию.

В любой СУБД есть внутренний механизм, который отвечает за исполнение запроса. Большинство запросов можно выполнить разными способами, т.к. запрос обычно состоит из нескольких отдельных операций, и можно меняя порядок и способ выполнения отдельных операций получить очень разную производительность. Пока мы пишем запросы к маленьким таблицам, в



которых десятки или сотни строк, запрос выполняется за доли секунды. Но давайте представим, что нам нужно найти информацию в таблицах, в которой сотни тысяч, или даже миллионы строк. Конечно, выполнять запрос в любом случае будет СУБД, но мы можем ей помочь, используя структуры, которые позволяют выполнить запросы быстрее.

Сначала давайте определимся, в чем измеряется скорость поиска. Данные хранятся на каком-то устройстве хранения, обычно в устройстве хранения выделяются блоки данных, которые считываются целиком. Чтобы оценить эффективность выполнения запроса, подсчитывают количество блоков, которые нужно прочитать в оперативную память, или, наоборот, записать на диск. Дело в том, что обработка данных в оперативной памяти выполняется очень быстро по сравнению с чтением/записью данных с устройств хранения.

StudentId	StudentName	GroupCode	BirthDate	Address
345568	William Johnson	M2020_1	01/20/1999	711 Station Road, London N63 5SF
345569	Lily Brown	M2020_1	05/17/1999	93 Manchester Road, London SE02 6VS
345570	Dan Black	M2020_1	08/11/1998	701 Station Road, London N63 5SF
345571	Matthew Wilson	M2020_1	07/12/1999	55 Springfield Road, London SE51 3ZU
345572	Jack Lewis	M2020_1	11/24/1999	7 Albert Road, London E20 5DW
345573	Cindy Clark	M2020_1	10/21/1999	63 Church Street, London W56 3JQ
345574	Helen Larsson	B2020_1	03/15/1999	512 Station Road, London N63 5SF
345575	Lucy Brooks	B2020_1	01/24/1999	55 Alexander Road, London EC78 5GI
345576	Josie Reed	B2020_1	05/22/1999	47 Grange Road, London W07 1XW

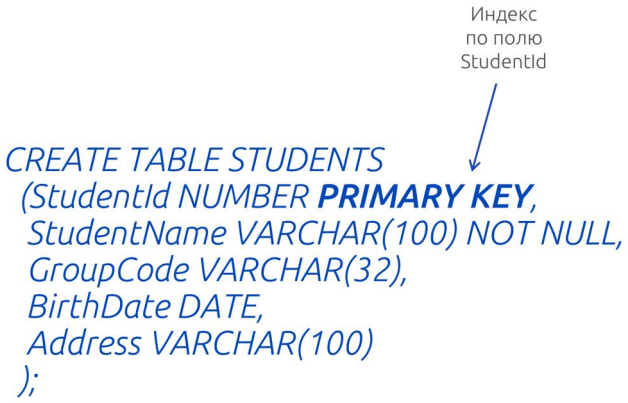
Давайте представим, что у нам нужно найти в большой таблице какие-то данные по определенным критериям. Если строки таблицы хранятся в произвольном порядке, то есть всего один способ – это последовательно считывать блоки данных и проверять все строки - соответствуют они критерию или нет. Такой метод называется сканированием таблицы. Скорость доступа в таком случае будет равна $N/2$, где N – это количество блоков в таблице, т.к. искомая строка с равной вероятностью может оказаться в любом блоке, в среднем нам придется просмотреть половину. Если блоков 1000, то количество чтений будет равно 500. Давайте представим, что мы взяли словарь английских слов и перепутали все слова местами. Легко ли будет найти нужное слово?

А теперь отсортируем все слова по алфавиту – стало искать намного легче. Чтобы найти нужное слово, можно воспользоваться методом половинного деления – открыли словарь на середине, нашли половинку, в которой находится искомое слово, и также ищем в ней. Продолжаем делить пополам, пока не найдем нужный элемент. Поиск еще не очень быстрый, но намного лучше, чем полное сканирование. Его оценка – $\log_2 N$. При том же количестве блоков 1000 теперь среднее количество прочитанных блоков стало равно 10.

Некоторые индексы строятся автоматически при создании таблицы. Они

всегда создаются для полей, которые объявлены первичным ключом или уникальными значениями.

Без индекса невозможно поддержать уникальность значений, т.к. при добавлении каждого нового значения нужно убедиться, что такого значения еще нет.



Но что делать, если нам нужен быстрый поиск, а значение поля не является уникальным? Неужели поиск по этим полям возможен только при полном сканировании таблицы?

Оказывается, все не так плохо. Мы можем построить индекс специальной командой, после создания таблицы. Параметры этой команды могут немного отличаться в разных СУБД, но в любом случае нужно указать таблицу и поля, по которым строится индекс, и дать новому объекту название.

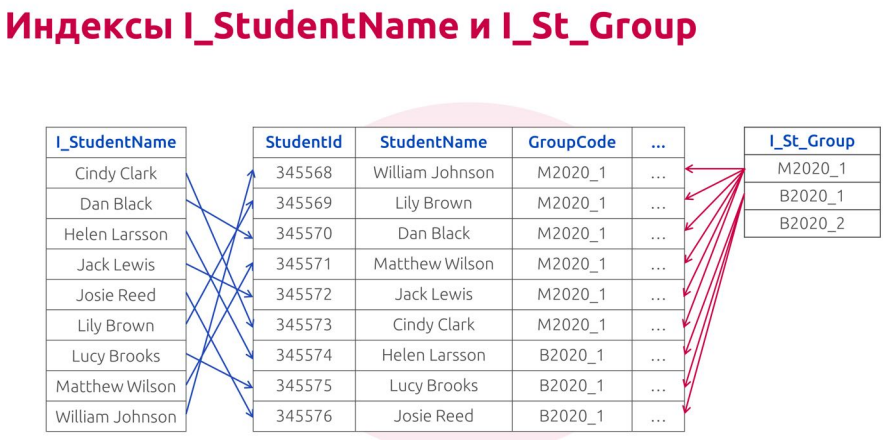
```
CREATE INDEX IndexName
ON TableName(TableField [,TableField2])
```

Индексы надо строить по тем полям, которые часто задаются в качестве критерия поиска данных. Например, студентов можно искать не только по номеру зачетки, но и по фамилии – значит, нужно построить индекс.

```
CREATE INDEX I_StudentName
ON Students(StudentName)
```

Построенный индекс создаст упорядоченный список значений выбранного поля и указатели на строки исходной таблицы, где хранится соответствующее значение Если поиск по какому-то критерию работает очень долго, то нужно проверить, построен ли индекс.

В то же время лишних индексов строить не нужно, так как любое обновление данных потребует обновления индексов, и если их много, то обновление будет занимать очень много времени. Например, никогда не нужно строить индексы к маленьким таблицам, т.к. прямой просмотр будет работать с маленькой таблицей быстрее. Кроме того, не все виды индексов подходят для полей с небольшим набором возможных значений: например, поле Пол, день недели, месяц.



На этом мы заканчиваем знакомство с реляционными СУБД, которые представляют собой удобные механизмы для работы со структурированными данными. Однако в настоящее время отмечается значительный рост объемов неструктурированных данных – потоков видео, фотографий, документов, текстов программ и прочего. Для работы с такими данными созданы другие хранилища – их часто называют NoSQL. И там используются совсем другие технологии.