МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

Проектирование баз данных

Методические указания к выполнению лабораторных работ (draft)

Санкт-Петербург

2021

Составитель: Н.В. Путилова

Рецензент:

Методические указания предназначены для студентов специальностей 02.03.03 и 09.03.04, изучающих дисциплину «Проектирование баз данных». В методические указания включены краткие теоретические сведения, необходимые для выполнения лабораторных работ, требования к содержанию отчетов и порядку выполнения работ, а также варианты индивидуальных заданий.

Методические указания подготовлены кафедрой компьютерных технологий и программной инженерии.

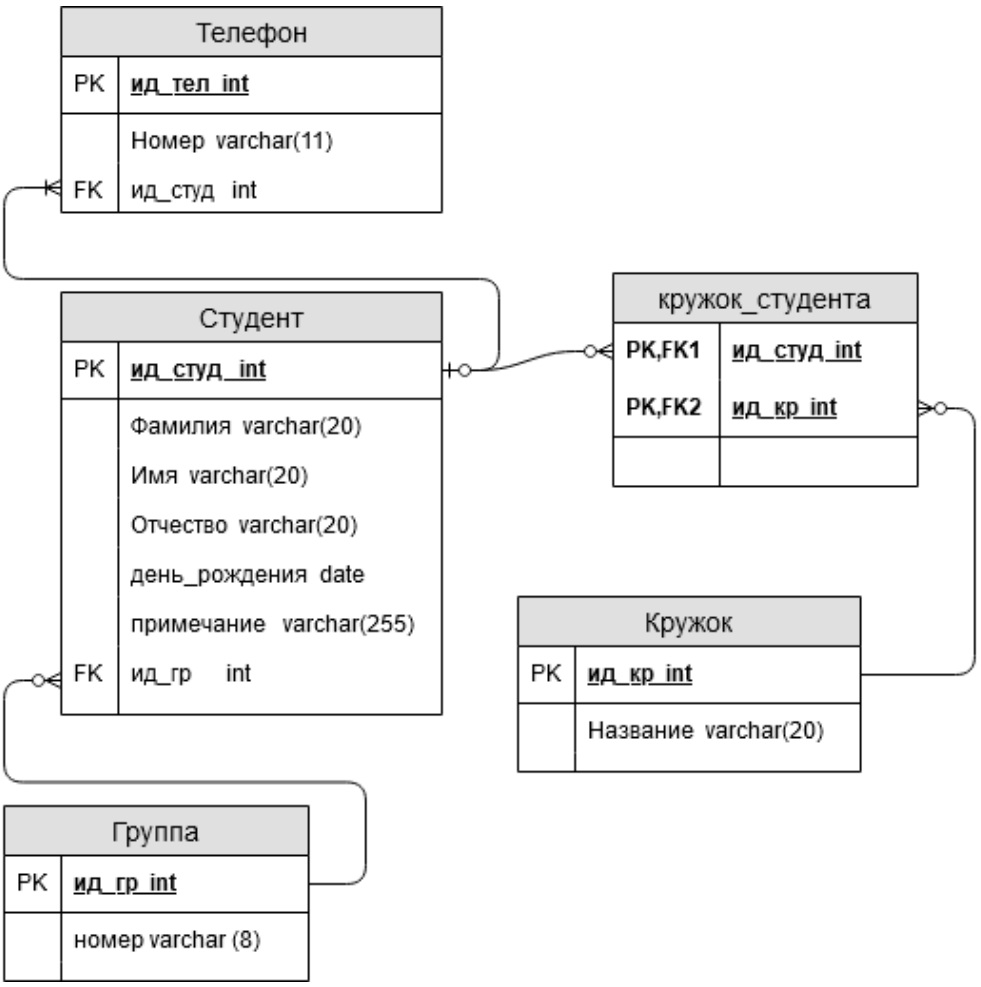
Оглавление

Лабораторная работа №1 Разработка физической модели базы данных с учетом декларативной ссылочной целостности

1. Теоретическая часть

Физические модели данных (PDM) обычно генерируются из логических моделей, но могут быть и воссозданы из действующей базы данных. Средства физического моделирования поддерживают множество стандартных нотаций, а также документирование, генерацию кода и воссоздание структур данных для более чем 70 РСУБД (в том числе новейших версий СУБД Oracle, IBM, Microsoft, Sybase, Teradata, MySQL, HP NeoView и многих других). Поддерживаются все объекты СУБД, необходимые для связывания таблиц и достижения требуемой производительности, такие как индексы, констрейнты, партиции и кластеры, а также новые технологии, такие как Java, XML и веб-сервисы в БД; проектируются средства обеспечения безопасности, применяются передовые методы создания представлений, обеспечивается ряд других возможностей. Физическую модель данных, как правило, можно использовать для оценки объема будущей БД; она может содержать специфическую информацию о выделении дисковой памяти для заданной СУБД.

1. Можно выделить следующие этапы методологии физического проектирования баз данных.
2. 1. Перенос глобальной логической модели данных в среду целевой СУБД.
3. 2. Проектирование основных отношений.
4. 3. Разработка способов получения производных данных.
5. 4. Реализация ограничений предметной области.
6. 5. Проектирование физического представления базы данных.
7. 6. Анализ транзакций.
8. 7. Выбор файловой структуры.
9. 8. Определение индексов.
10. 9. Определение требований к дисковой памяти.
11. 10. Проектирование пользовательских представлений.
12. 11. Разработка механизмов защиты.
13. 12. Обоснование необходимости введения контролируемой избыточности.
14. 13. Текущий контроль и настройка операционной систем
15. Логическая модель расширяет концептуальную путем определения для сущностей их атрибутов, описаний и ограничений, уточняет состав сущностей и взаимосвязи между ними.
16. Физическая модель данных описывает реализацию объектов логической модели на уровне объектов конкретной базы данных.



Пример Физической модели БД

Логическое представление реляционных баз данных упрощается созданием связей между данными на основе (логической) конструкции, называемой таблицей. Под таблицей понимается двумерная структура, состоящая из строк и столбцов. Пользователь должен понимать, что таблица содержит группу связанных сущностей, т. е. набор сущностей; по этой причине термины набор сущностей и таблица чаще всего означают одно и то же. Таблица также называется отношением (relation), поскольку создатель реляционной модели Э. Ф. Кодд (E. F. Codd) использовал термин "отношение" как синоним слова "таблица".

Правила целостности

ЦЕЛОСТНОСТЬ НА УРОВНЕ СУЩНОСТИ

Требование Все элементы первичного ключа уникальны и никакая часть первичного ключа не может быть пустой (null)

Назначение Гарантирует, что каждая сущность (логический объект) будет иметь уникальную идентификацию, а значения внешнего ключа могут должным образом ссылаться на значения первичного ключа

Пример Счет не может иметь несколько дублирующихся значений и не может иметь пустое значение (null). Короче говоря, все счета уникально идентифицируются своим номером

ЦЕЛОСТНОСТЬ НА УРОВНЕ ССЫЛКИ

Требования Внешний ключ может иметь или пустое значение (если только он не является частью первичного ключа данной таблицы), или значение, совпадающее со значением первичного ключа в

связанной таблице. (Каждое непустое значение внешнего ключа должно ссылаться на существующее значение первичного ключа.)
Назначение Допускается, что атрибут не имеет соответствующего значения, но атрибут не может принимать недопустимые значения. Выполнение правила целостности на уровне ссылки делает невозможным удаление строки в одной таблице, где первичный ключ имеет обязательное соответствие со значением внешнего ключа в другой таблице
Пример Клиенту может быть не назначен (еще) торговый агент, но невозможно назначить клиенту несуществующего агента

Теперь о *внешних ключах*:

- Если сущность C связывает сущности A и B, то она должна включать внешние ключи, соответствующие первичным ключам сущностей A и B.
- Если сущность B обозначает сущность A, то она должна включать внешний ключ, соответствующий первичному ключу сущности A.
- Здесь для обозначения любой из ассоциируемых сущностей (стержней, характеристик, обозначений или даже ассоциаций) используется новый обобщающий термин "Цель" или "Целевая сущность".
- Таким образом, при рассмотрении проблемы выбора способа представления ассоциаций и обозначений в базе данных основной вопрос, на который следует получить ответ: "Каковы внешние ключи?". И далее, для каждого внешнего ключа необходимо решить три вопроса:
- 1. Может ли данный внешний ключ принимать неопределенные значения (NULL-значения)? Иначе говоря, может ли существовать некоторый экземпляр сущности данного типа, для которого неизвестна целевая сущность, указываемая внешним ключом? В случае поставок это, вероятно, невозможно – поставка, осуществляемая неизвестным поставщиком, или поставка неизвестного продукта не имеют смысла. Но в случае с сотрудниками такая ситуация однако могла бы иметь смысл – вполне возможно, что какой-либо сотрудник в данный момент не зачислен вообще ни в какой отдел. Заметим, что ответ на данный вопрос не зависит от прихоти проектировщика базы данных, а определяется фактическим образом действий, принятым в той части реального мира, которая должна быть представлена в рассматриваемой базе данных. Подобные замечания имеют отношение и к вопросам, обсуждаемым ниже.
- 2. Что должно случиться при попытке УДАЛЕНИЯ целевой сущности, на которую ссылается внешний ключ? Например, при удалении поставщика, который осуществил по крайней мере одну поставку. Существует три возможности:

КАСКАДИРУЕТСЯ	Операция удаления "каскадируется" с тем, чтобы удалить также поставки этого поставщика.
ОГРАНИЧИВАЕТСЯ	Удаляются лишь те поставщики, которые еще не осуществляли поставок. Иначе операция удаления отвергается.
УСТАНАВЛИВАЕТСЯ	Для всех поставок удаляемого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а затем этот поставщик удаляется. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

- 3. Что должно происходить при попытке ОБНОВЛЕНИЯ первичного ключа целевой сущности, на которую ссылается некоторый внешний ключ? Например, может быть предпринята попытка обновить номер такого поставщика, для которого имеется по крайней

мере одна соответствующая поставка. Этот случай для определенности снова рассмотрим подробнее. Имеются те же три возможности, как и при удалении:

КАСКАДИРУЕТСЯ Операция обновления "каскадируется" с тем, чтобы обновить также и внешний ключ впоставках этого поставщика.

ОГРАНИЧИВАЕТСЯ Обновляются первичные ключи лишь тех поставщиков, которые еще не осуществляли поставок. Иначе операция обновления отвергается.

УСТАНАВЛИВАЕТСЯ Для всех поставок такого поставщика NULL-значение внешний ключ устанавливается в неопределенное значение, а затем обновляется первичный ключ поставщика. Такая возможность, конечно, неприменима, если данный внешний ключ не должен содержать NULL-значений.

- Таким образом, для каждого внешнего ключа в проекте проектировщик базы данных должен специфицировать не только поле или комбинацию полей, составляющих этот внешний ключ, и целевую таблицу, которая идентифицируется этим ключом, но также и ответы на указанные выше вопроса (три ограничения, которые относятся к этому внешнему ключу).

Ссылочную целостность при выполнении лабораторной необходимо описать в таблице вида

Дочерняя таблица (с внешним ключом)	Внешний ключ	Родительская таблица	Как поддерживается ссылочная целостность при удалении	Описание ссылочной целостности при удалении	Как поддерживается ссылочная целостность при обновлении	Описание ссылочной целостности при обновлении	Обоснование
Table1	Id_t2	Table2	Каскадируется	При удалении данных из Table2, удалятся все связанные данные из Table1	Каскадируется	При обновлении первичного ключа Table2, обновится внешний ключ из Table1	Логическое обоснование почему так
Table1	Id_t2	Table2	ограничивается	При удалении	ограничивается	При обновлении	Логическое обоснование

				данных из Table2, если есть связанные данные из Table1, удаление будет отменено/ запрещено		первичного ключа Table2, если есть связанные данные из Table1, обновление будет отменено/ запрещено	почему так
Table1	Id_t2	Table2	устанавливается	При удалении данных из Table2, внешний ключ Table1, ссылающийся на них получит значение null (или значение по умолчанию)	ограничивается	При обновлении первичного ключа Table2, внешний ключ Table1, ссылающийся на них получит значение null (или значение по умолчанию)	Логическое обоснование почему так
student	Id_gr	st_group	каскадируется	При удалении данных из таблицы «st_group», удалятся ссылающи	ограничивается	При обновлении первичного ключа «st_group», если есть связанные	Может быть необходимым удалять группу со всеми студентами, но

				еся на них данные в таблице «student»		данные из «student», обновление будет отменено/ запрещено	необходимость менять суррогатный внешний ключ маловероятна

Описание может быть не в таблице, но должно содержать следующие данные:

- Дочерняя таблица;
- Столбцы, составляющие внешний ключ;
- Родительская таблица;
- Наименование ссылочной целостности при удалении;
- Описание действий по поддержанию ссылочной целостности при удалении;
- Наименование ссылочной целостности при обновлении;
- Описание действий по поддержанию ссылочной целостности при обновлении;
- Обоснование выбора типа поддержки ссылочной целостности.

2. Выполнение лабораторной работы

Создать физическую модель базы данных, находящуюся в третьей нормальной форме в соответствии с заданным вариантом. Расписать ссылочную целостность БД в таблице

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— таблица с описанием ссылочной целостности.

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №2 Создание и модификация базы данных и таблиц базы данных

1. Теоретическая часть

Таблица . Основные типы данных языка SQL, определенные в стандарте ISO(SQL -92)

Тип данных	Объявления	Реализация Microsoft SQL Server	Реализация MySQL
boolean (Логический)	BOOLEAN		BOOL, BOOLEAN, TINYINT(1).
character (Символьный)	CHAR	CHAR[(M)], NCHAR[(M)](unicode)	CHAR [(M)]
	VARCHAR	VARCHAR[(M)], NVARCHAR[(M)](unicode)	VARCHAR[(M)]
bit (Битовый)	BIT, BIT VARYING	BIT	BIT BIT[(M)]
exact numeric (Точные числа)	NUMERIC	NUMERIC[(M[,D])]	DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
	DECIMAL	DECIMAL[(M[,D])]	DECIMAL[(M[,D])] [UNSIGNED] [ZEROFILL]
	SMALLINT	SMALLINT	SMALLINT[(M)] [UNSIGNED] [ZEROFILL]
	INTEGER	INT	INT[(M)] [UNSIGNED] [ZEROFILL] INTEGER[(M)] [UNSIGNED] [ZEROFILL]
	TINYINT	TINYINT	TINYINT[(M)] [UNSIGNED] [ZEROFILL] -128 .. 127 или 0..255
	BIGINT	BIGINT	BIGINT (8 байт)
approximate numeric (Округленные числа)	FLOAT	FLOAT[(N)]	FLOAT[(M,D)] [UNSIGNED] [ZEROFILL] REAL[(M,D)] [UNSIGNED] [ZEROFILL](если задан режим REAL_AS_FLOAT)
	REAL	REAL (эквивалентно Float(24))	
	DOUBLE PRECISION	DOUBLE PRECISION	DOUBLE[(M,D)] [UNSIGNED] [ZEROFILL], REAL[(M,D)] [UNSIGNED] [ZEROFILL](если не задан режим REAL_AS_FLOAT)
datetime (Дата/время)	DATE	DATE	DATE
	TIME	TIME	TIME
	TIMESTAMP	DATETIME TIMESTAMP	DATETIME TIMESTAMP
interval	INTERV		С другим

(Интервал)	AL		синтаксисом , но используются TIME и TIMESTAMP
LOB (Большой объект)	CHARACTER LARGE OBJECT,	TEXT, VARCHAR(MAX)	TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT
	BINARY LARGE OBJECT	VARBINARY(MAX) VARBINARY(n)	TINYBLOB, BLOB, MEDIUMBLOB, LONGBLOB
XML (Правильный XML документ)		XML	XML

Синтаксис оператора `CREATE TABLE`

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] tbl_name [(create_definition,...)]
[table_options] [select_statement]

create_definition:
  col_name type [NOT NULL | NULL] [DEFAULT default_value] [AUTO_INCREMENT]
              [PRIMARY KEY] [reference_definition]
  или   PRIMARY KEY (index_col_name,...)
  или   KEY [index_name] (index_col_name,...)
  или   INDEX [index_name] (index_col_name,...)
  или   UNIQUE [INDEX] [index_name] (index_col_name,...)
  или   FULLTEXT [INDEX] [index_name] (index_col_name,...)
  или   [CONSTRAINT symbol] FOREIGN KEY [index_name] (index_col_name,...)
              [reference_definition]
  или   CHECK (expr)

type:
        TINYINT[(length)] [UNSIGNED] [ZEROFILL]
  или   SMALLINT[(length)] [UNSIGNED] [ZEROFILL]
  или   MEDIUMINT[(length)] [UNSIGNED] [ZEROFILL]
  или   INT[(length)] [UNSIGNED] [ZEROFILL]
  или   INTEGER[(length)] [UNSIGNED] [ZEROFILL]
  или   BIGINT[(length)] [UNSIGNED] [ZEROFILL]
  или   REAL[(length,decimals)] [UNSIGNED] [ZEROFILL]
  или   DOUBLE[(length,decimals)] [UNSIGNED] [ZEROFILL]
  или   FLOAT[(length,decimals)] [UNSIGNED] [ZEROFILL]
  или   DECIMAL(length,decimals) [UNSIGNED] [ZEROFILL]
  или   NUMERIC(length,decimals) [UNSIGNED] [ZEROFILL]
  или   CHAR(length) [BINARY]
  или   VARCHAR(length) [BINARY]
  или   DATE
  или   TIME
  или   TIMESTAMP
  или   DATETIME
  или   TINYBLOB
  или   BLOB
  или   MEDIUMBLOB
  или   LONGBLOB
  или   TINYTEXT
  или   TEXT
  или   MEDIUMTEXT
```

```
  или   LONGTEXT
  или   ENUM(value1,value2,value3,...)
  или   SET(value1,value2,value3,...)

index_col_name:
        col_name [(length)]

reference_definition:
        REFERENCES tbl_name [(index_col_name,...)]
                   [MATCH FULL | MATCH PARTIAL]
                   [ON DELETE reference_option]
                   [ON UPDATE reference_option]

reference_option:
        RESTRICT | CASCADE | SET NULL | NO ACTION | SET DEFAULT

table_options:
        TYPE = {BDB | HEAP | ISAM | InnoDB | MERGE | MRG_MYISAM | MYISAM }
или     AUTO_INCREMENT = #
или     AVG_ROW_LENGTH = #
или     CHECKSUM = {0 | 1}
или     COMMENT = "string"
или     MAX_ROWS = #
или     MIN_ROWS = #
или     PACK_KEYS = {0 | 1 | DEFAULT}
или     PASSWORD = "string"
или     DELAY_KEY_WRITE = {0 | 1}
или     ROW_FORMAT= { default | dynamic | fixed | compressed }
или     RAID_TYPE= {1 | STRIPED | RAID0 } RAID_CHUNKS=#  RAID_CHUNKSIZE=#
или     UNION = (table_name,[table_name...])
или     INSERT_METHOD= {NO | FIRST | LAST }
или     DATA DIRECTORY="абсолютный путь к каталогу"
или     INDEX DIRECTORY="абсолютный путь к каталогу"

select_statement:
        [IGNORE | REPLACE] SELECT ...  (любое корректное выражение SELECT)
```

Оператор `CREATE TABLE` создает таблицу с заданным именем в текущей базе данных.

. Если нет активной текущей базы данных или указанная таблица уже существует, то возникает ошибка выполнения команды.

В версии MySQL 3.22 и более поздних имя таблицы может быть указано как `db_name.tbl_name`. Эта форма записи работает независимо от того, является ли указанная база данных текущей.

В версии MySQL 3.23 при создании таблицы можно использовать ключевое слово `TEMPORARY`. Временная таблица автоматически удаляется по завершении соединения, а ее имя действительно только в течение данного соединения. Это означает, что в двух разных соединениях могут использоваться временные таблицы с одинаковыми именами без конфликта друг с другом или с существующей таблицей с тем же именем (существующая таблица скрыта, пока не удалена временная таблица). В версии MySQL 4.0.2 для создания временных таблиц необходимо иметь привилегии `CREATE TEMPORARY TABLES`.

В версии MySQL 3.23 и более поздних можно использовать ключевые слова `IF NOT EXISTS` для того, чтобы не возникала ошибка, если указанная таблица уже существует. Следует учитывать, что при этом не проверяется идентичность структур этих таблиц.

Каждая таблица tbl_name представлена определенными файлами в директории базы данных. В случае таблиц типа MyISAM это следующие файлы:

Файл	Назначение
tbl_name.frm	Файл определения таблицы
tbl_name.MYD	Файл данных
tbl_name.MYI	Файл индексов

Чтобы получить более полную информацию о свойствах различных типов столбцов,

Если не указывается ни NULL, ни NOT NULL, то столбец интерпретируется так, как будто указано NULL.

- Целочисленный столбец может иметь дополнительный атрибут AUTO_INCREMENT. При записи величины NULL (рекомендуется) или 0 в столбец AUTO_INCREMENT данный столбец устанавливается в значение value+1, где value представляет собой наибольшее для этого столбца значение в таблице на момент записи. Последовательность AUTO_INCREMENT начинается с 1.. Если удалить строку, содержащую максимальную величину для столбца AUTO_INCREMENT, то в таблицах типа ISAM или BDB эта величина будет восстановлена, а в таблицах типа MyISAM или InnoDB - нет. Если удалить все строки в таблице командой DELETE FROM table_name (без выражения WHERE) в режиме AUTOCOMMIT, то для таблиц всех типов последовательность начнется заново. Примечание: в таблице может быть только один столбец AUTO_INCREMENT, и он должен быть индексирован. Кроме того, версия MySQL 3.23 будет правильно работать только с положительными величинами столбца AUTO_INCREMENT. В случае внесения отрицательного числа оно интерпретируется как очень большое положительное число. Это делается, чтобы избежать проблем с точностью, когда числа ``заворачиваются" от положительного к отрицательному и, кроме того, для гарантии, что по ошибке не будет получен столбец AUTO_INCREMENT со значением 0. В таблицах MyISAM и BDB можно указать вторичный столбец AUTO_INCREMENT с многостолбцовым ключом.. Последнюю внесенную строку можно найти с помощью следующего запроса (чтобы сделать MySQL совместимым с некоторыми ODBC-приложениями):
- SELECT * FROM tbl_name WHERE auto_col IS NULL
- Величины NULL для столбца типа TIMESTAMP обрабатываются иначе, чем для столбцов других типов. В столбце TIMESTAMP нельзя хранить литерал NULL; при установке данного столбца в NULL он будет установлен в текущее значение даты и времени. Поскольку столбцы TIMESTAMP ведут себя подобным образом, то атрибуты NULL и NOT NULL неприменимы в обычном режиме и игнорируются при их задании. С другой стороны, чтобы облегчить клиентам MySQL использование столбцов TIMESTAMP, сервер сообщает, что таким столбцам могут быть назначены величины NULL (что соответствует действительности), хотя реально TIMESTAMP никогда не будет содержать величины NULL. Это можно увидеть, применив DESCRIBE tbl_name для получения описания данной таблицы. Следует учитывать, что установка столбца TIMESTAMP в 0 не равнозначна установке его в NULL, поскольку 0 для TIMESTAMP является допустимой величиной.
- Величина DEFAULT должна быть константой, она не может быть функцией или выражением. Если для данного столбца не задается никакой величины DEFAULT, то MySQL автоматически назначает ее. Если столбец может принимать NULL как допустимую величину, то по умолчанию присваивается значение NULL. Если столбец объявлен как NOT NULL, то значение по умолчанию зависит от типа столбца:

- Для числовых типов, за исключением объявленных с атрибутом AUTO_INCREMENT, значение по умолчанию равно 0. Для столбца AUTO_INCREMENT значением по умолчанию является следующее значение в последовательности для этого столбца.
- Для типов даты и времени, отличных от TIMESTAMP, значение по умолчанию равно соответствующей нулевой величине для данного типа. Для первого столбца TIMESTAMP в таблице значение по умолчанию представляет собой текущее значение даты и времени.
- Для строковых типов, кроме ENUM, значением по умолчанию является пустая строка. Для ENUM значение по умолчанию равно первой перечисляемой величине (если явно не задано другое значение по умолчанию с помощью директивы DEFAULT).

Значения по умолчанию должны быть константами. Это означает, например, что нельзя установить для столбца даты в качестве значения по умолчанию величину функции, такой как NOW() или CURRENT_DATE.

- KEY является синонимом для INDEX.
- В MySQL ключ UNIQUE может иметь только различающиеся значения. При попытке добавить новую строку с ключом, совпадающим с существующей строкой, возникает ошибка выполнения команды.
- PRIMARY KEY представляет собой уникальный ключ KEY с дополнительным ограничением, что все столбцы с данным ключом должны быть определены как NOT NULL. В MySQL этот ключ называется PRIMARY (первичный). Таблица может иметь только один первичный ключ PRIMARY KEY. Если PRIMARY KEY отсутствует в таблицах, а некоторое приложение запрашивает его, то MySQL может превратить в PRIMARY KEY первый ключ UNIQUE, не имеющий ни одного столбца NULL.
- PRIMARY KEY может быть многостолбцовым индексом. Однако нельзя создать многостолбцовый индекс, используя в определении столбца атрибут ключа PRIMARY KEY. Именно таким образом только один столбец будет отмечен как первичный. Необходимо использовать синтаксис PRIMARY KEY(index_col_name, ...).
- Если ключ PRIMARY или UNIQUE состоит только из одного столбца и он принадлежит к числовому типу, то на него можно сослаться также как на _rowid (новшество версии 3.23.11).
- Если индексу не назначено имя, то ему будет присвоено первое имя в index_col_name, возможно, с суффиксами (_2, _3, . . .), делающими это имя уникальным. Имена индексов для таблицы можно увидеть, используя SHOW INDEX FROM tbl_name. SHOW Syntax.
- Только таблицы типов MyISAM, InnoDB и BDB поддерживают индексы столбцов, которые могут иметь величины NULL. В других случаях, во избежание ошибки, необходимо объявлять такие столбцы как NOT NULL.
- С помощью выражения col_name(length) можно указать индекс, для которого используется только часть столбца CHAR или VARCHAR. Это поможет сделать файл индексов намного меньше.
- Индексацию столбцов BLOB и TEXT поддерживают только таблицы с типом MyISAM. Назначая индекс столбцу с типом BLOB или TEXT, всегда НЕОБХОДИМО указывать длину этого индекса:
- CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
- При использовании выражений ORDER BY или GROUP BY со столбцом типа TEXT или BLOB используются только первые max_sort_length байтов.
- В версии MySQL 3.23.23 и более поздних можно создавать также специальные индексы FULLTEXT. Они применяются для полнотекстового поиска. Эти индексы поддерживаются только таблицами типа MyISAM и они могут быть созданы только из столбцов VARCHAR и TEXT.

Индексирование всегда выполняется для всего столбца целиком, частичная индексация не поддерживается. Более подробно эта операция описана в разделе

- Выражения `FOREIGN KEY`, `CHECK` и `REFERENCES` фактически ничего не делают. Они введены только из соображений совместимости, чтобы облегчить перенос кода с других SQL-серверов и запускать приложения, создающие таблицы со ссылками
- Для каждого столбца `NULL` требуется один дополнительный бит, при этом величина столбца округляется в большую сторону до ближайшего байта.
- Максимальную длину записи в байтах можно вычислить следующим образом:
- `длина записи = 1`
- `+ (сумма длин столбцов)`
- `+ (количество столбцов с допустимым NULL + 7)/8`
- `+ (количество столбцов с динамической длинной)`
- Опции `table_options` и `SELECT` реализованы только в версиях MySQL 3.23 и выше. Ниже представлены различные типы таблиц:

Тип таблицы	Описание
BDB или Berkeley_db	Таблицы с поддержкой транзакций и блокировкой страниц. See section 7.6 Таблицы BDB или Berkeley_DB.
HEAP	Данные для этой таблицы хранятся только в памяти. See section 7.4 Таблицы HEAP.
ISAM	Оригинальный обработчик таблиц. See section 7.3 Таблицы ISAM.
InnoDB	Таблицы с поддержкой транзакций и блокировкой строк. See section 7.5 Таблицы InnoDB.
MERGE	Набор таблиц MyISAM, используемый как одна таблица. See section 7.2 Таблицы MERGE.
MRG_MyISAM	Псевдоним для таблиц MERGE
MyISAM	Новый обработчик, обеспечивающий переносимость таблиц в бинарном виде, который заменяет ISAM. See section 7.1 Таблицы MyISAM.

- See section 7 Типы таблиц MySQL. Если задается тип таблицы, который не поддерживается данной версией, то MySQL выберет из возможных типов ближайший к указанному. Например, если задается `TYPE=BDB` и данный дистрибутив MySQL не поддерживает таблиц `BDB`, то вместо этого будет создана таблица `MyISAM`. Другие табличные опции используются для оптимизации характеристик таблицы. Эти опции в большинстве случаев не требуют специальной установки. Данные опции работают с таблицами всех типов, если не указано иное:

Опция	Описание
AUTO_INCREMENT	Следующая величина `AUTO_INCREMENT`, которую следует установить для данной таблицы (`MyISAM`).
AVG_ROW_LENGTH	Приближенное значение средней длины строки для данной таблицы. Имеет смысл устанавливать только для обширных таблиц с записями переменной длины.
CHECKSUM	Следует установить в `1`, чтобы в MySQL поддерживалась проверка контрольной суммы для всех строк (это делает таблицы немного более медленными при обновлении, но позволяет легче находить поврежденные

таблицы) (MyISAM).

COMMENT	Комментарий для данной таблицы длиной 60 символов.
MAX_ROWS	Максимальное число строк, которые планируется хранить в данной таблице.
MIN_ROWS	Минимальное число строк, которые планируется хранить в данной таблице.
PACK_KEYS	Следует установить в 1 для получения меньшего индекса. Обычно это замедляет обновление и ускоряет чтение (MyISAM, ISAM). Установка в 0 отключит уплотнение ключей. При установке в DEFAULT (MySQL 4.0) обработчик таблиц будет уплотнять только длинные столбцы CHAR/VARCHAR.
PASSWORD	Шифрует файл `.frm' с помощью пароля. Эта опция не функционирует в стандартной версии MySQL.
DELAY_KEY_WRITE	Установка в 1 задерживает операции обновления таблицы ключей, пока не закроется указанная таблица (MyISAM).
ROW_FORMAT	Определяет, каким образом должны храниться строки. В настоящее время эта опция работает только с таблицами MyISAM, которые поддерживают форматы строк DYNAMIC и FIXED. See section 7.1.2 Форматы таблиц MyISAM.

- При использовании таблиц MyISAM MySQL вычисляет выражение max_rows * avg_row_length, чтобы определить, насколько велика будет результирующая таблица. Если не задана ни одна из вышеупомянутых опций, то максимальный размер таблицы будет составлять 4Гб (или 2Гб если данная операционная система поддерживает только таблицы величиной до 2Гб). Это делается для того, чтобы, если нет реальной необходимости в больших файлах, ограничить размеры указателей, что позволит сделать индексы меньше и быстрее. Если опция PACK_KEYS не используется, то по умолчанию уплотняются только строки, но не числа. При использовании PACK_KEYS=1 числа тоже будут уплотняться. При уплотнении двоичных числовых ключей MySQL будет использовать сжатие префиксов. Это означает, что выгода от этого будет значительной только в случае большого количества одинаковых чисел. При сжатии префиксов для каждого ключа требуется один дополнительный байт, в котором указано, сколько байтов предыдущего ключа являются такими же, как и для следующего (следует учитывать, что указатель на строку хранится в порядке "старший-байт-в-начале" сразу после ключа - чтобы улучшить компрессию). Это означает, что при наличии нескольких одинаковых ключей в двух строках записи все последующие ``аналогичные" ключи будут занимать только по 2 байта (включая указатель строки). Сравним: в обычном случае для хранения последующих ключей требуется размер_хранения_ключа + размер_указателя (обычно 4) байтов. С другой стороны, если все ключи абсолютно разные, каждый ключ будет занимать на 1 байт больше, если данный ключ не может иметь величину NULL (в этом случае уплотненный ключ будет храниться в том же байте, который используется для указания, что ключ равен NULL).
- Если после команды CREATE указывается команда SELECT, то MySQL создаст новые поля для всех элементов в данной команде SELECT. Например:

```
mysql> CREATE TABLE test (a INT NOT NULL AUTO_INCREMENT,
        PRIMARY KEY (a), KEY(b))
        TYPE=MyISAM SELECT b,c FROM test2;
```

Эта команда создаст таблицу MyISAM с тремя столбцами a, b и c. Отметим, что столбцы из команды SELECT присоединяются к таблице справа, а не перекрывают ее. Рассмотрим следующий пример:

```
mysql> SELECT * FROM foo;
+---+
| n |
+---+
| 1 |
+---+

mysql> CREATE TABLE bar (m INT) SELECT n FROM foo;
Query OK, 1 row affected (0.02 sec)
Records: 1 Duplicates: 0 Warnings: 0

mysql> SELECT * FROM bar;
+------+---+
| m    | n |
+------+---+
| NULL | 1 |
+------+---+
1 row in set (0.00 sec)
```

Каждая строка в таблице `foo` вносится в таблицу `bar` со своим значением из `foo`, при этом в новые столбцы в таблице `bar` записываются величины, заданные по умолчанию. Команда `CREATE TABLE ... SELECT` не создает автоматически каких-либо индексов. Это сделано преднамеренно, чтобы команда была настолько гибкой, насколько возможно. Чтобы иметь индексы в созданной таблице, необходимо указать их перед данной командой `SELECT`:

```
mysql> CREATE TABLE bar (UNIQUE (n)) SELECT n FROM foo;
```

Если возникает ошибка при копировании данных в таблицу, то они будут автоматически удалены. Чтобы обеспечить возможность использовать для восстановления таблиц журнал обновлений/двоичный журнал, в MySQL во время выполнения команды `CREATE TABLE ... SELECT` не разрешены параллельные вставки.

- Воспользовавшись опцией `RAID_TYPE`, можно разбить файл данных `MyISAM` на участки с тем, чтобы преодолеть 2Гб/4Гб лимит файловой системы под управлением ОС, не поддерживающих большие файлы. Разбиение не касается файла индексов. Следует учесть, что для файловых систем, которые поддерживают большие файлы, эта опция не рекомендуется! Для получения более высокой скорости ввода-вывода можно разместить RAID-директории на различных физических дисках. `RAID_TYPE` будет работать под любой операционной системой, если конфигурация MySQL выполнена с параметром `--with-raid`. В настоящее время для опции `RAID_TYPE` возможен только параметр `STRIPED` (`1` и `RAID0` являются псевдонимами для него). Если указывается `RAID_TYPE=STRIPED` для таблицы `MyISAM`, то `MyISAM` создаст поддиректории `RAID_CHUNKS` с именами `` `00' ``, `` `01' ``, `` `02' `` в директории базы данных. В каждой из этих директорий `MyISAM` создаст файл `` `table_name.MYD' ``. При записи данных в файл данных обработчик RAID установит соответствие первых `RAID_CHUNKSIZE*1024` байтов первому упомянутому файлу, следующих `RAID_CHUNKSIZE*1024` байтов - следующему файлу и так далее.
- Опция `UNION` применяется, если необходимо использовать совокупность идентичных таблиц как одну таблицу. Она работает только с таблицами `MERGE`. иметь привилегии `SELECT`, `UPDATE` и `DELETE`. Все сопоставляемые таблицы должны принадлежать той же базе данных, что и таблица `MERGE`.
- Для внесения данных в таблицу `MERGE` необходимо указать с помощью `INSERT_METHOD`, в какую таблицу данная строка должна быть внесена.. Эта опция была введена в MySQL 4.0.0.

- В созданной таблице ключ PRIMARY будет помещен первым, за ним все ключи UNIQUE и затем простые ключи. Это помогает оптимизатору MySQL определять приоритеты используемых ключей, а также более быстро определять сдублированные ключи UNIQUE.
- Используя опции DATA DIRECTORY="каталог" или INDEX DIRECTORY="каталог", можно указать, где обработчик таблицы должен помещать свои табличные и индексные файлы. Следует учитывать, что указываемый параметр directory должен представлять собой полный путь к требуемому каталогу (а не относительный путь). Данные опции работают только для таблиц MyISAM в версии MySQL 4.0, если при этом не используется опция --skip-symlink.

Синтаксис оператора ALTER TABLE

```
ALTER [IGNORE] TABLE tbl_name alter_spec [, alter_spec ...]

alter_specification:
        ADD [COLUMN] create_definition [FIRST | AFTER column_name ]
  или   ADD [COLUMN] (create_definition, create_definition,...)
  или   ADD INDEX [index_name] (index_col_name,...)
  или   ADD PRIMARY KEY (index_col_name,...)
  или   ADD UNIQUE [index_name] (index_col_name,...)
  или   ADD FULLTEXT [index_name] (index_col_name,...)
  или   ADD [CONSTRAINT symbol] FOREIGN KEY index_name (index_col_name,...)
            [reference_definition]
  или   ALTER [COLUMN] col_name {SET DEFAULT literal | DROP DEFAULT}
  или   CHANGE [COLUMN] old_col_name create_definition
               [FIRST | AFTER column_name]
  или   MODIFY [COLUMN] create_definition [FIRST | AFTER column_name]
  или   DROP [COLUMN] col_name
  или   DROP PRIMARY KEY
  или   DROP INDEX index_name
  или   DISABLE KEYS
  или   ENABLE KEYS
  или   RENAME [TO] new_tbl_name
  или   ORDER BY col
  или   table_options
```

Оператор ALTER TABLE обеспечивает возможность изменять структуру существующей таблицы. Например, можно добавлять или удалять столбцы, создавать или уничтожать индексы или переименовывать столбцы либо саму таблицу. Можно также изменять комментарий для таблицы и ее тип.

Если оператор ALTER TABLE используется для изменения определения типа столбца, но DESCRIBE tbl_name показывает, что столбец не изменился, то, возможно, MySQL игнорирует данную модификацию по одной из причин, описанных в разделе. Например, при попытке изменить столбец VARCHAR на CHAR MySQL будет продолжать использовать VARCHAR, если данная таблица содержит другие столбцы с переменной длиной.

Оператор ALTER TABLE во время работы создает временную копию исходной таблицы. Требуемое изменение выполняется на копии, затем исходная таблица удаляется, а новая переименовывается. Так делается для того, чтобы в новую таблицу автоматически попадали все обновления кроме неудавшихся. Во время выполнения ALTER TABLE исходная таблица доступна для чтения другими клиентами. Операции обновления и записи в этой таблице приостанавливаются, пока не будет готова новая таблица.

Следует отметить, что при использовании любой другой опции для ALTER TABLE кроме RENAME, MySQL всегда будет создавать временную таблицу, даже если данные, строго говоря, и не нуждаются в копировании (например, при изменении имени столбца). Мы планируем исправить это в будущем, однако, поскольку ALTER TABLE выполняется не так часто, мы (разработчики MySQL) не считаем эту задачу первоочередной. Для таблиц MyISAM можно увеличить скорость воссоздания индексной части (что является наиболее медленной частью в процессе восстановления таблицы) путем установки переменной myisam_sort_buffer_size достаточно большого значения.

- Для использования оператора ALTER TABLE необходимы привилегии ALTER, INSERT и CREATE для данной таблицы.
- Опция IGNORE является расширением MySQL по отношению к ANSI SQL92. Она управляет работой ALTER TABLE при наличии дубликатов уникальных ключей в новой таблице. Если опция IGNORE не задана, то для данной копии процесс прерывается и происходит откат назад. Если IGNORE указывается, тогда для строк с дубликатами уникальных ключей только первая строка используется, а остальные удаляются.
- Можно запустить несколько выражений ADD, ALTER, DROP и CHANGE в одной команде ALTER TABLE. Это является расширением MySQL по отношению к ANSI SQL92, где допускается только одно выражение из упомянутых в одной команде ALTER TABLE.
- Опции CHANGE col_name, DROP col_name и DROP INDEX также являются расширениями MySQL по отношению к ANSI SQL92.
- Опция MODIFY представляет собой расширение Oracle для команды ALTER TABLE.
- Необязательное слово COLUMN представляет собой ``белый шум'' и может быть опущено.
- При использовании ALTER TABLE имя_таблицы RENAME TO новое_имя без каких-либо других опций MySQL просто переименовывает файлы, соответствующие заданной таблице. В этом случае нет необходимости создавать временную таблицу. .
- В выражении create_definition для ADD и CHANGE используется тот же синтаксис, что и для CREATE TABLE. Следует учитывать, что этот синтаксис включает имя столбца, а не просто его тип.
- Столбец можно переименовывать, используя выражение CHANGE имя_столбца create_definition. Чтобы сделать это, необходимо указать старое и новое имена столбца и его тип в настоящее время. Например, чтобы переименовать столбец INTEGER из a в b, можно сделать следующее:
- ```
  mysql> ALTER TABLE t1 CHANGE a b INTEGER;
  ```

  При изменении типа столбца, но не его имени синтаксис выражения CHANGE все равно требует указания обоих имен столбца, даже если они одинаковы. Например:

  ```
  mysql> ALTER TABLE t1 CHANGE b b BIGINT NOT NULL;
  ```

  Однако начиная с версии MySQL 3.22.16a можно также использовать выражение MODIFY для изменения типа столбца без переименовывания его:

  ```
  mysql> ALTER TABLE t1 MODIFY b BIGINT NOT NULL;
  ```

- При использовании CHANGE или MODIFY для того, чтобы уменьшить длину столбца, по части которого построен индекс (например, индекс по первым 10 символам столбца VARCHAR), нельзя сделать столбец короче, чем число проиндексированных символов.
- При изменении типа столбца с использованием CHANGE или MODIFY MySQL пытается преобразовать данные в новый тип как можно корректнее.
  ```

- В версии MySQL 3.22 и более поздних можно использовать `FIRST` или `ADD ... AFTER имя_столбца` для добавления столбца на заданную позицию внутри табличной строки. По умолчанию столбец добавляется в конце. Начиная с версии MySQL 4.0.1, можно также использовать ключевые слова `FIRST` и `AFTER` в опциях `CHANGE` или `MODIFY`.
- Опция `ALTER COLUMN` задает для столбца новое значение по умолчанию или удаляет старое. Если старое значение по умолчанию удаляется и данный столбец может принимать значение `NULL`, то новое значение по умолчанию будет `NULL`. Если столбец не может быть `NULL`, то MySQL назначает значение по умолчанию .
- Опция `DROP INDEX` удаляет индекс. Это является расширением MySQL по отношению к ANSI SQL92.
- Если столбцы удаляются из таблицы, то эти столбцы удаляются также и из любого индекса, в который они входят как часть. Если все столбцы, составляющие индекс, удаляются, то данный индекс также удаляется.
- Если таблица содержит только один столбец, то этот столбец не может быть удален. Вместо этого можно удалить данную таблицу, используя команду `DROP TABLE`.
- Опция `DROP PRIMARY KEY` удаляет первичный индекс. Если такого индекса в данной таблице не существует, то удаляется первый индекс `UNIQUE` в этой таблице. (MySQL отмечает первый уникальный ключ `UNIQUE` как первичный ключ `PRIMARY KEY`, если никакой другой первичный ключ `PRIMARY KEY` не был явно указан). При добавлении `UNIQUE INDEX` или `PRIMARY KEY` в таблицу они хранятся перед остальными неуникальными ключами, чтобы можно было определить дублирующиеся ключи как можно раньше.
- Опция `ORDER BY` позволяет создавать новую таблицу со строками, размещенными в заданном порядке. Следует учитывать, что созданная таблица не будет сохранять этот порядок строк после операций вставки и удаления. В некоторых случаях такая возможность может облегчить операцию сортировки в MySQL, если таблица имеет такое расположение столбцов, которое вы хотели бы иметь в дальнейшем. Эта опция в основном полезна, если заранее известен определенный порядок, в котором преимущественно будут запрашиваться строки. Использование данной опции после значительных преобразований таблицы дает возможность получить более высокую производительность.
- При использовании команды `ALTER TABLE` для таблиц `MyISAM` все неуникальные индексы создаются в отдельном пакете (подобно `REPAIR`). Благодаря этому команда `ALTER TABLE` при наличии нескольких индексов будет работать быстрее.
- Начиная с MySQL 4.0, вышеуказанная возможность может быть активизирована явным образом. Команда `ALTER TABLE ... DISABLE KEYS` блокирует в MySQL обновление неуникальных индексов для таблиц `MyISAM`. После этого можно применить команду `ALTER TABLE ... ENABLE KEYS` для воссоздания недостающих индексов. Так как MySQL делает это с помощью специального алгоритма, который намного быстрее в сравнении со вставкой ключей один за другим, блокировка ключей может дать существенное ускорение на больших массивах вставок.
- Применяя функцию C API `mysql_info()`, можно определить, сколько записей было скопировано, а также (при использовании `IGNORE`) - сколько записей было удалено из-за дублирования значений уникальных ключей.
- Выражения `FOREIGN KEY`, `CHECK` и `REFERENCES` фактически ничего не делают. Они введены только из соображений совместимости, чтобы облегчить перенос кода с других серверов SQL и запуск приложений, создающих таблицы со ссылками.
- Ниже приводятся примеры, показывающие некоторые случаи употребления команды `ALTER TABLE`. Пример начинается с таблицы `t1`, которая создается следующим образом:

```
mysql> CREATE TABLE t1 (a INTEGER,b CHAR(10));
```

Для того чтобы переименовать таблицу из t1 в t2:

```
mysql> ALTER TABLE t1 RENAME t2;
```

Для того чтобы изменить тип столбца с INTEGER на TINYINT NOT NULL (оставляя имя прежним) и изменить тип столбца b с CHAR(10) на CHAR(20) с переименованием его с b на c:

```
mysql> ALTER TABLE t2 MODIFY a TINYINT NOT NULL, CHANGE b c CHAR(20);
```

Для того чтобы добавить новый столбец TIMESTAMP с именем d:

```
mysql> ALTER TABLE t2 ADD d TIMESTAMP;
```

Для того чтобы добавить индекс к столбцу d и сделать столбец a первичным ключом:

```
mysql> ALTER TABLE t2 ADD INDEX (d), ADD PRIMARY KEY (a);
```

Для того чтобы удалить столбец c:

```
mysql> ALTER TABLE t2 DROP COLUMN c;
```

Для того чтобы добавить новый числовой столбец AUTO_INCREMENT с именем c:

```
mysql> ALTER TABLE t2 ADD c INT UNSIGNED NOT NULL AUTO_INCREMENT,
        ADD INDEX (c);
```

Заметьте, что столбец c индексируется, так как столбцы AUTO_INCREMENT должны быть индексированы, кроме того, столбец c объявляется как NOT NULL, поскольку индексированные столбцы не могут быть NULL.

При добавлении столбца AUTO_INCREMENT значения этого столбца автоматически заполняются последовательными номерами (при добавлении записей). Первый номер последовательности можно установить путем выполнения команды SET INSERT_ID=# перед ALTER TABLE или использования табличной опции AUTO_INCREMENT = #.

2. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 и создать базу данных с разработанной физической моделью. Продемонстрировать умение добавить и удалить столбец командой alter table

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— скрипт создания таблиц базы данных на SQL;

— скрипт изменения структуры таблиц базы данных на SQL.

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №3 Заполнение таблиц и модификация данных

1. Теоретическая часть

В данной работе рассматриваются следующие операторы языка манипулирования данными (DML):

Insert – вставки данных;

Update – изменения данных;

Delete – удаление данных

Merge – слияние данных таблиц (действие 3предыдущих команд в одной)

Синтаксис оператора `INSERT`

Выделяют 3 различных синтаксисса

1) вставляет строки в соответствии с точно указанными в команде значениями.(форма со списком из нескольких значений поддерживается в версии MySQL 3.22.5 и более поздних)

```
INSERT [IGNORE]
    [INTO] tbl_name [(col_name,...)]
    VALUES (expression,...),(...),...
```

2) вставляет строки, выбранные из другой таблицы или таблиц

```
INSERT      [INTO] tbl_name [(col_name,...)]
    SELECT ...
```

3)синтаксис вставки строк, который поддерживается в версии MySQL 3.22.10 и более поздних

```
INSERT  [IGNORE] [INTO] tbl_name
    SET col_name=expression, col_name=expression, ...
```

В каждом из них после слова INSERT может стоять один из 3 модификаторов:

- LOW_PRIORITY- если таблицу использует другой процесс, вставка будет отложена до её освобождения
- DELAYED использовалось до версии 5.6, аналог LOW_PRIORITY
- HIGH_PRIORITY- в противоположность LOW_PRIORITY – немедленная вставка вне зависимости от занятости таблицы

HIGH_PRIORITY и LOW_PRIORITY работают только на таблицах, где блокировка уровня таблицы (MyISAM, MEMORY и MERGE). Следует отметить, что указатель `LOW_PRIORITY` обычно не используется с таблицами MyISAM, поскольку при его указании становятся невозможными параллельные вставки.

`tbl_name` задает таблицу, в которую должны быть внесены строки. Столбцы, для которых заданы величины в команде, указываются в списке имен столбцов или в части `SET`:

- Если не указан список столбцов для `INSERT ... VALUES` или `INSERT ... SELECT`, то величины для всех столбцов должны быть определены в списке `VALUES()` или в результате работы `SELECT`. Если порядок столбцов в таблице неизвестен, для его получения можно использовать `DESCRIBE tbl_name`.
- Любой столбец, для которого явно не указано значение, будет установлен в свое значение по умолчанию.
- Выражение `expression` может относится к любому столбцу, который ранее был внесен в список значений. Например, можно указать следующее:
- ```
  mysql> INSERT INTO tbl_name (col1,col2) VALUES(15,col1*2);
  ```

  Но нельзя указать:

  ```
  mysql> INSERT INTO tbl_name (col1,col2) VALUES(col2*2,15);
  ```

- Если в команде `INSERT` со строками, имеющими много значений, указывается ключевое слово `IGNORE`, то все строки, имеющие дублирующиеся ключи `PRIMARY` или `UNIQUE` в этой таблице, будут проигнорированы и не будут внесены.

Предупреждения возникают при выполнении любого из следующих условий:

- Внесение `NULL` в столбец, который был объявлен, как `NOT NULL`. Данный столбец устанавливается в значение, заданное по умолчанию.
- Установка числового столбца в значение, лежащее за пределами его допустимого диапазона. Данная величина усекается до соответствующей конечной точки этого диапазона.
- Занесение в числовой столбец такой величины, как `'10.34 a'`. Конечные данные удаляются и вносится только оставшаяся числовая часть. Если величина вовсе не имеет смысла как число, то столбец устанавливается в `0`.
- Внесение в столбцы типа `CHAR`, `VARCHAR`, `TEXT` или `BLOB` строки, превосходящей максимальную длину столбца. Данная величина усекается до максимальной длины столбца.
- Внесение в столбец даты или времени строки, недопустимой для данного типа столбца. Этот столбец устанавливается в нулевую величину, соответствующую данному типу.

Синтаксис оператора `UPDATE`

```
UPDATE [LOW_PRIORITY] [IGNORE] tbl_name
    SET col_name1=expr1 [, col_name2=expr2, ...]
    [WHERE where_definition]
    [LIMIT #]
```

Оператор `UPDATE` обновляет столбцы в соответствии с их новыми значениями в строках существующей таблицы. В выражении `SET` указывается, какие именно столбцы следует модифицировать и какие величины должны быть в них установлены. В выражении `WHERE`, если оно присутствует, задается, какие строки подлежат обновлению. В остальных случаях обновляются все строки. Если задано выражение `ORDER BY`, то строки будут обновляться в указанном в нем порядке.

Если указывается ключевое слово `LOW_PRIORITY`, то выполнение данной команды `UPDATE` задерживается до тех пор, пока другие клиенты не завершат чтение этой таблицы.

Если указывается ключевое слово `IGNORE`, то команда обновления не будет прервана, даже если при обновлении возникнет ошибка дублирования ключей. Строки, из-за которых возникают конфликтные ситуации, обновлены не будут.
```

Если доступ к столбцу из указанного выражения осуществляется по аргументу `tbl_name`, то команда `UPDATE` использует для этого столбца его текущее значение. Например, следующая команда устанавливает столбец `age` в значение, на единицу большее его текущей величины:

```
mysql> UPDATE persondata SET age=age+1;
```

Значения команда `UPDATE` присваивает слева направо. Например, следующая команда дублирует столбец `age`, затем инкрементирует его:

```
mysql> UPDATE persondata SET age=age*2, age=age+1;
```

Если столбец устанавливается в его текущее значение, то MySQL замечает это и не обновляет его.

Команда `UPDATE` возвращает количество фактически измененных строк. В версии MySQL 3.22 и более поздних функция C API `mysql_info()` возвращает количество строк, которые были найдены и обновлены, и количество предупреждений, имевших место при выполнении `UPDATE`.

В версии MySQL 3.23 можно использовать `LIMIT #`, чтобы убедиться, что было изменено только заданное количество строк.

Синтаксис оператора `DELETE`

```
DELETE [LOW_PRIORITY | QUICK] FROM table_name
       [WHERE where_definition]
       [ORDER BY ...]
       [LIMIT rows]
```

или

```
DELETE [LOW_PRIORITY | QUICK] table_name[.*] [,table_name[.*] ...]
       FROM table-references
       [WHERE where_definition]
```

оили

```
DELETE [LOW_PRIORITY | QUICK]
       FROM table_name[.*], [table_name[.*] ...]
       USING table-references
       [WHERE where_definition]
```

Оператор `DELETE` удаляет из таблицы `table_name` строки, удовлетворяющие заданным в `where_definition` условиям, и возвращает число удаленных записей.

Если оператор `DELETE` запускается без определения `WHERE`, то удаляются все строки. При работе в режиме `AUTOCOMMIT` это будет аналогично использованию оператора `TRUNCATE`. В MySQL 3.23 оператор `DELETE` без определения `WHERE` возвратит ноль как число удаленных записей.

Если действительно необходимо знать число удаленных записей при удалении всех строк, и если допустимы потери в скорости, то можно использовать команду `DELETE` в следующей форме:

```
mysql> DELETE FROM table_name WHERE 1>0;
```

Следует учитывать, что эта форма работает намного медленнее, чем `DELETE FROM table_name` без выражения `WHERE`, поскольку строки удаляются поочередно по одной.

Если указано ключевое слово LOW_PRIORITY, выполнение данной команды DELETE будет задержано до тех пор, пока другие клиенты не завершат чтение этой таблицы.

Если задан параметр QUICK, то обработчик таблицы при выполнении удаления не будет объединять индексы - в некоторых случаях это может ускорить данную операцию.

В таблицах MyISAM удаленные записи сохраняются в связанном списке, а последующие операции INSERT повторно используют места, где располагались удаленные записи. Чтобы возвратить неиспользуемое пространство и уменьшить размер файлов, можно применить команду OPTIMIZE TABLE или утилиту myisamchk для реорганизации таблиц. Команда OPTIMIZE TABLE проще, но утилита myisamchk работает быстрее.

Первый из числа приведенных в начале данного раздела многотабличный формат команды DELETE поддерживается, начиная с MySQL 4.0.0. Второй многотабличный формат поддерживается, начиная с MySQL 4.0.2.

Идея заключается в том, что удаляются только совпадающие строки из таблиц, перечисленных перед выражениями FROM или USING. Это позволяет удалять единовременно строки из нескольких таблиц, а также использовать для поиска дополнительные таблицы.

Символы .* после имен таблиц требуются только для совместимости с Access:

```
DELETE t1,t2 FROM t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id
```

или

```
DELETE FROM t1,t2 USING t1,t2,t3 WHERE t1.id=t2.id AND t2.id=t3.id
```

В предыдущем случае просто удалены совпадающие строки из таблиц t1 и t2.

Выражение ORDER BY и использование нескольких таблиц в команде DELETE поддерживается в MySQL 4.0.

Если применяется выражение ORDER BY, то строки будут удалены в указанном порядке. В действительности это выражение полезно только в сочетании с LIMIT. Например:

```
DELETE FROM somelog
        WHERE user = 'jcole'
        ORDER BY timestamp
        LIMIT 1
```

Данный оператор удалит самую старую запись (по timestamp), в которой строка соответствует указанной в выражении WHERE.

Специфическая для MySQL опция LIMIT для команды DELETE указывает серверу максимальное количество строк, которые следует удалить до возврата управления клиенту. Эта опция может использоваться для гарантии того, что данная команда DELETE не потребует слишком много времени для выполнения. Можно просто повторять команду DELETE до тех пор, пока количество удаленных строк меньше, чем величина LIMIT.

Слияние данных (оператор MERGE)

Оператор MERGE в стандарте с SQL:2003, улучшен в SQL:2008, удаление введено в оператор с SQL:2011)

Оператор MERGE не поддерживается mySQL, поэтому синтаксис при выполнении работы на mysql привести надо, а запускать запрос -нет

Выполняет операции вставки, обновления или удаления для целевой таблицы на основе результатов соединения с исходной таблицей. Например, можно синхронизировать две таблицы путем вставки, обновления или удаления строк в одной таблице на основании отличий, найденных в другой таблице.

```
MERGE
   [ TOP ( expression ) [ PERCENT ] ]
   [ INTO ] <target_table> [ WITH ( <merge_hint> ) ] [ [ AS ] table_alias ]
   USING <table_source>
   ON <merge_search_condition>
   [ WHEN MATCHED [ AND <clause_search_condition> ]
      THEN <merge_matched> ] [ ...n ]
   [ WHEN NOT MATCHED [ BY TARGET ] [ AND <clause_search_condition> ]
      THEN <merge_not_matched> ]
   [ WHEN NOT MATCHED BY SOURCE [ AND <clause_search_condition> ]
      THEN <merge_matched> ] [ ...n ]
   [ <output_clause> ]
   [ OPTION ( <query_hint> [ ,...n ] ) ]
;
```

Аргументы

TOP (expression) [PERCENT]

Указывает количество или процент строк, которые подпадают под эту операцию. Аргумент expression может быть либо числом, либо процентной долей строк. Строки, на которые ссылается выражение TOP, не расположены в определенном порядке.

Предложение TOP применяется после соединения всей исходной таблицы и всей целевой таблицы и удаления соединенных строк, которые не рассматриваются как предназначенные для выполнения операций вставки, обновления или удаления. Предложение TOP дополнительно сокращает количество соединенных строк до указанного значения, а затем к оставшимся соединенным строкам применяются операции вставки, обновления или удаления без учета порядка. Иными словами, порядок, в котором строки подвергаются операциям, определенным в предложениях WHEN, не задан. Например, указание значения TOP (10) затрагивает 10 строк. Из них 7 могут быть обновлены и 3 вставлены или 1 может быть удалена, 5 обновлено и 4 вставлено и т. д.

Инструкция MERGE выполняет полный просмотр исходной и целевой таблиц, поэтому при использовании предложения TOP для изменения большой таблицы путем создания нескольких пакетов производительность ввода-вывода может снизиться. В этом случае необходимо обеспечить, чтобы во всех подряд идущих пакетах осуществлялась обработка новых строк.

target_table

Таблица или представление, с которыми выполняется сопоставление строк данных из таблицы <table_source> по условию <clause_search_condition>. Параметр target_table является целевым объектом любых операций вставки, обновления или удаления, указанных предложениями WHEN инструкции MERGE.

USING <table_source>

Указывается источник данных, который сопоставляется со строками данных в таблице target_table на основе условия <merge_search condition>. Результат этого совпадения обуславливает действия, которые выполняются предложениями WHEN инструкции MERGE. Аргумент <table_source> может быть удаленной таблицей или производной таблицей, которая обращается к удаленным таблицам.

ON <merge_search_condition>

Указываются условия, при которых таблица <table_source> соединяется с таблицей target_table для сопоставления. Необходимо указать столбцы целевой таблицы, которые сравниваются с соответствующим столбцом исходной таблицы.

WHEN MATCHED THEN <merge_matched>

Указывается, что все строки target_table, которые соответствуют строкам, возвращенным <table_source> ON <merge_search_condition>, и удовлетворяют дополнительным условиям поиска, обновляются или удаляются в соответствии с предложением <merge_matched>.

Инструкция MERGE может иметь не больше двух предложений WHEN MATCHED. Если указаны два предложения, первое предложение должно сопровождаться предложением AND <search_condition>. Для любой строки второе предложение WHEN MATCHED применяется только в тех случаях, если не применяется первое. Если имеются два предложения WHEN MATCHED, одно должно указывать действие UPDATE, а другое — действие DELETE. Если действие UPDATE указано в предложении <merge_matched> и более одной строки в <table_source>соответствует строке в target_table на основе <merge_search_condition>, то SQL Server возвращает ошибку. Инструкцию MERGE нельзя использовать для обновления одной строки более одного раза, а также использовать для обновления и удаления одной и той же строки.

WHEN NOT MATCHED [BY TARGET] THEN <merge_not_matched>

Указывает, что строка вставлена в таблицу target_table для каждой строки, возвращенной выражением <table_source> ON <merge_search_condition>, которая не соответствует строке в таблице target_table, но удовлетворяет дополнительному условию поиска (при наличии такового). Значения для вставки указываются с помощью предложения

<merge_not_matched>. Инструкция MERGE может иметь только одно предложение WHEN **NOT** MATCHED.

WHEN NOT MATCHED BY SOURCE THEN <merge_matched>

Указывается, что все строки target_table, которые не соответствуют строкам, возвращенным <table_source> ON <merge_search_condition>, и удовлетворяют дополнительным условиям поиска, обновляются или удаляются в соответствии с предложением <merge_matched>.

Инструкция MERGE может иметь не более двух предложений WHEN NOT MATCHED BY SOURCE. Если указаны два предложения, то первое предложение должно сопровождаться предложением AND <clause_search_condition>. Для любой выбранной строки второе предложение WHEN NOT MATCHED BY SOURCE применяется только в тех случаях, если не применяется первое. Если имеется два предложения WHEN NOT MATCHED BY SOURCE, то одно должно указывать действие UPDATE, а другое — действие DELETE. В условии <clause_search_condition> можно ссылаться только на столбцы целевой таблицы.

<merge_matched>

Указывает действие обновления или удаления, которое применяется ко всем строкам target_table, не соответствующим строкам, которые возвращаются <table_source> ON <merge_search_condition> и удовлетворяют каким-либо дополнительным условиям поиска.

UPDATE SET <set_clause>

Указывается список имен столбцов или переменных, которые необходимо обновить в целевой таблице, и значений, которые необходимо использовать для их обновления.

<merge_not_matched>

Указываются значения для вставки в целевую таблицу.

(column_list)

Список, состоящий из одного или нескольких столбцов целевой таблицы, в которые вставляются данные. Столбцы необходимо указывать в виде однокомпонентного имени, так как в противном случае инструкция MERGE возвращает ошибку. Список column_list должен быть заключен в круглые скобки, а его элементы должны разделяться запятыми.

VALUES (values_list)

Список с разделителями-запятыми констант, переменных или выражений, которые возвращают значения для вставки в целевую таблицу. Выражения не могут содержать инструкцию EXECUTE.

DEFAULT VALUES

Заполняет вставленную строку значениями по умолчанию, определенными для каждого столбца.

<search condition>

Указываются условия поиска, используемые для указания <merge_search_condition> или <clause_search_condition>.

Пример сольем таблицу студентов и восстановившихся студентов (результат в таблице Student)

Поле Action может иметь 3 значения :

‘New’ – добавить строку со студентом в таблицу Student

‘Mod’ – изменить строку со студентом в таблице Student (например другая группа)

‘Del’ – удалить студента из таблицы Student

```
MERGE INTO Student AS S
USING resume_study_student AS R
ON S.id_student = R.id_student
WHEN MATCHED AND
R.Action = 'Mod'
THEN UPDATE
SET  id_gr=R.id_gr, surname= R.surname, name=R.name, patronym =R.patronym
WHEN MATCHED AND
R.Action = 'Del'
THEN DELETE
WHEN NOT MATCHED AND
I.Action = 'New'
THEN INSERT
VALUES (R.id_student,R.id_gr, R.surname, R.name, R.patronym)
```

2. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 Выполнить вставку тестовых данных в таблицы, созданные в ходе выполнения лабораторной работы 2. В строках, вставляемых в таблицы, должны быть данные как удовлетворяющие, так и не удовлетворяющие условиям запросов, приведенных в варианте задания. Необходимо привести свои пример использования оператором ***update*** и ***delete*** и ***merge***

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— наборы данных, содержащихся в таблицах БД;

— Таблицы тестовых данных вида:

Текст запроса	данные удовлетворяющие условиям запросов	данные не удовлетворяющие условиям запросов
а. станции в названии которых есть слово «площадь»	Таблица1 (станция) Площадь Ленина	Таблица1 (станция) Московские ворота

— примеры использования insert, update, delete и merge;

— скрипт полного заполнения базы (удобно после исползовать для быстрого развертывания БД на другой машине).

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №4 Разработка SQL запросов: виды соединений и шаблоны

1. Теоретическая часть

Оператор выборки в языке SQL

Назначение оператора SELECT состоит в выборке и отображении данных одной или более таблиц базы данных. Это исключительно мощный оператор, способный выполнять действия, эквивалентные операторам реляционной алгебры выборки, проекции и соединения, причем в пределах единственной выполняемой команды. Оператор SELECT является чаще всего используемой командой языка SQL. Общий формат оператора SELECT имеет следующий вид:

SELECT [DISTINCT | ALL] .{ *| *[columnExpression [*AS *newName]] [, ...] }*
FROM TableName [alias] [,..]
[WHERE condition]
[GROUP BY *columnList*] [HAVING condition]
[ORDER BY *columnList]*

Слово distinct позволяет убрать дублирующиеся строки из результата запроса.

Здесь параметр *columnExpression* представляет собой имя столбца или выражение из нескольких имен.

columnExpression может включать только следующие типы элементов:

• имена столбцов;

• агрегирующие функции;

• константы;

• выражения, включающие комбинации перечисленных выше элементов.

В противоположность списку столбцов * обозначает возврат всех столбцов всех таблиц запроса.

Параметр *TableName* является именем существующей в базе данных таблицы (или представления), к которой необходимо получить доступ. Необязательный параметр alias — это сокращение, псевдоним, устанавливаемое для имени таблицы *TableName.* Обработка элементов оператора SELECT выполняется в следующей последовательности.

• FROM. Определяются имена используемой таблицы или нескольких таблиц, перечисленные через запятую. Также в качестве источника данных в этом разделе могут быть таблицы связанные различными видами соединения, представления и другие запросы (смотри запросы с подзапросами)

• WHERE. Выполняется фильтрация строк объекта в соответствии с заданными условиями.

• GROUP BY. Образуются группы строк, имеющих одно и то же значение в указанном столбце (использовать только с агрегатными функциями).

• HAVING. Фильтруются группы строк объекта в соответствии с указанным условием , относящимся к результату агрегатной функции.

• SELECT. Устанавливается, какие столбцы должны присутствовать в выходных данных.

• ORDER BY. Определяется упорядоченность результатов выполнения оператора.

При этом каждому полю в списке сортировки может быть приписано ASC (по возрастанию- это параметр сортировки поля по умолчанию) или DESC (по убыванию)

Порядок конструкций в операторе SELECT *не может* быть изменен. Только две конструкции оператора — SELECT и FROM — являются обязательными, все остальные конструкции могут быть опущены.

Условия в конструкции WHERE
(применимы как к оператору SELECT, так и UPDATE или INSERT)

В приведенных выше примерах в результате выполнения операторов SELECT выбирались все строки указанной таблицы. Однако очень часто требуется тем или иным образом ограничить набор строк, помещаемых в результирующую таблицу запроса. Это достигается с помощью указания в запросе конструкции WHERE. Она состоит из ключевого слова WHERE, за которым следует перечень условий поиска, определяющих те строки, которые должны быть выбраны при выполнении запроса. Существует пять основных типов условий поиска (или *предикатов,* если пользоваться терминологией ISO).

• Сравнение. Сравниваются результаты вычисления одного выражения с результатами вычисления другого выражения. (<,>,=,<>)

• Диапазон. Проверяется, попадает ли результат вычисления выражения в заданный диапазон значений. (имя_поля BETWEEN знач_1 AND знач_2)

• Принадлежность к множеству. Проверяется, принадлежит ли результат вычисления выражения к заданному множеству значений. (имя_поля in (знач_1,знач_2…, знач_n))

• Значение NULL. Проверяется, содержит ли данный столбец NULL (неопределенное значение) (имя_поля IS NULL).

• Соответствие шаблону. Проверяется, отвечает ли некоторое строковое значение заданному шаблону. (имя_поля LIKE шаблон)

В

• %. Символ процента представляет любую последовательность из нуля или более символов (поэтому часто именуется также *подстановочным символом).* (В ACCESS вместо него используется *, но при вызове из программы через ADO или DAO должен быть все равно %*)*

• _. Символ подчеркивания представляет любой отдельный символ. . (В ACCESS вместо него используется ?, но при вызове из программы через ADO или DAO должно быть все равно *_)*

Все остальные символы в шаблоне представляют сами себя.

Пример

• address LIKE 'Ш%'. Этот шаблон означает, что первый символ значения обязательно должен быть символом Ш, а все остальные символы не представляют интереса и не проверяются.

• address LIKE 'Ш_ _ _ _'. Этот шаблон означает, что значение должно иметь длину, равную строго четырем символам, причем первым символом обязательно должен быть символ ' Ш ' . (пробелы между подчеркиваниями только для визуализации –их нет в шаблоне)

• address LIKE ' %ич'. Этот шаблон определяет любую последовательностьсимволов длиной не менее двух символов, причем последними символами обязательно должен быть символы ич. (поиск мужчин по отчеству)

• address LIKE '%слово%'. Этот шаблон означает, что нас интересует

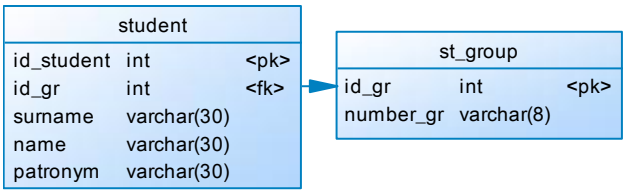
любая последовательность символов, включающая подстроку «слово»;

• address NOT LIKE ' Ш%'. Этот шаблон указывает, что требуются любые строки, которые не начинаются с символа Ш.

Если требуемая строка должна включать также служебный символ, обычно применяемый в качестве символа подстановки, то следует определить с помощью конструкции ESCAPE "маскирующий" символ, который указывает, что следующий за ним символ больше не имнет специального значения, и поместить его перед символом подстановки. Например, для проверки значений на соответствие литеральной строке 45%' можно воспользоваться таким предикатом:LIKE '15#%' ESCAPE '#'

Пример

Представим себе таблицы группа и студент.



Выберем всех студентов группы Z4432K

select id_student, student.id_gr, surname, name, patronym from student, st_group where student.id_gr =st_group.id_gr and number_gr='Z4432K'

Виды соединений в языке SQL

Виды соединений в SQL:

- Декартово произведение CROSS JOIN
- Внутреннее соединение INNER JOIN (часто просто JOIN)
- Внешние соединения:
- Левое соединение LEFT JOIN
- Правое соединение RIGHT JOIN
- Полное (внешнее) соединение OUTER JOIN

Соединение является подмножеством более общей комбинации данных двух таблиц, называемой *декартовым произведением* . Декартово произведение двух таблиц представляет собой другую таблицу, состоящую из всех возможных пар строк, входящих в состав обеих таблиц. Набор столбцов результирующей таблицы представляет собой все столбцы первой таблицы, за которыми следуют все столбцы второй таблицы. Если ввести запрос к двум таблицам без задания конструкции WHERE, результат выполнения запроса в среде SQL будет представлять собой декартово произведение этих таблиц. Кроме того, стандарт ISO предусматривает специальный формат оператора SELECT, позволяющий вычислить декартово произведение двух таблиц:

SELECT [DISTINCT. | ALL] {* | *columnList}*
FROM *tableName1* CROSS JOIN *tableName2*

При выполнении операции соединения данные из двух таблиц комбинируются с образованием пар связанных строк, в которых значения сопоставляемых столбцов являются одинаковыми. Если одно из значений в сопоставляемом столбце одной таблицы не совпадает ни с одним из значений в сопоставляемом столбце другой таблицы, то соответствующая строка удаляется из результирующей таблицы. Именно это правило применялось во всех рассмотренных выше примерах соединения таблиц. Стандартом ISO предусмотрен и другой набор операторов соединений, называемых *внешними соединениями*. Во внешнем соединении в результирующую таблицу помещаются также строки, не удовлетворяющие условию соединения.

Внутреннее соединение используется для связи таблиц по совпадающим значениям столбцов. Внешние соединения нужны в случае, когда значениям в одной таблице не всегда будут иметь соответствующие значения в другой (т.е. внешний ключ – пуст (NULL))

Свойства соединений:

- Внутреннее соединение симметрично, т.е

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* INNER JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* INNER JOIN *tableName1*

- Полное внешнее соединение симметрично, т.е.

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* OUTER JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* OUTER JOIN *tableName1*

- Левое и правое соединения симметричны друг по отношению к другу, т.е.

SELECT *tableName1.*, tableName2.*,* FROM *tableName1* LEFT JOIN *tableName2*
Эквивалентно
SELECT *tableName1.*, tableName2.*,* FROM *tableName2* RIGHT JOIN *tableName1*

Для примеров различных соединений используем таблицы группа и студент

student

id_student	id_gr	surname	name	patronym
1	1	Петров	Петр	Петрович
2	2	Иванов	Иван	Иванович
3	*NULL*	Пупкин	Василий	Федорович

st_group

id_gr	number_gr
1	Z4431K
2	Z5432K
3	B5433

Select number_gr, id_student, surname, name, patronym from st_group inner join student on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group INNER JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович

Select number_gr, id_student, surname, name, patronym from st_group left join student on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group LEFT JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
B5433	*NULL*	*NULL*	*NULL*	*NULL*

Select number_gr, id_student, surname, name, patronym from st_group right join student on student.id_gr=st_group.id_gr

Ему эквивалентен запрос Select number_gr, id_student, surname, name, patronym from student left join st_group on student.id_gr=st_group.id_gr

Результат такого запросы будет таким

st_group RIGHT JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
NULL	3	Пупкин	Василий	Федорович

Select number_gr, id_student, surname, name, patronym from st_group outer join student on student.id_gr=st_group.id_gr

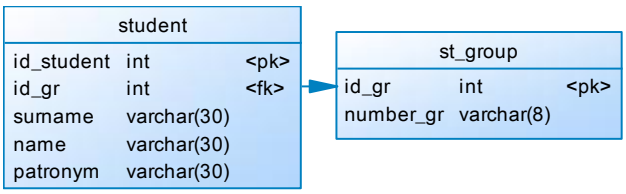
Результат такого запросы будет таким

st_group OUTER JOIN student

number_gr	id_student	surname	name	patronym
Z4431K	1	Петров	Петр	Петрович
Z5432K	2	Иванов	Иван	Иванович
NULL	3	Пупкин	Василий	Федорович
B5433	*NULL*	*NULL*	*NULL*	*NULL*

Это можно использовать для нахождения групп без студентов

Пример



Группы без студентов

```
Select number_gr, id_student, surname, name, patronym FROM
st_group LEFT JOIN student ON student.id_gr=st_group.id_gr
WHERE student.id_student is NULL;
```

```
Select number_gr, id_student, surname, name, patronym FROM
student RIGHT JOIN st_group ON student.id_gr=st_group.id_gr
WHERE student.id_student is NULL;
```

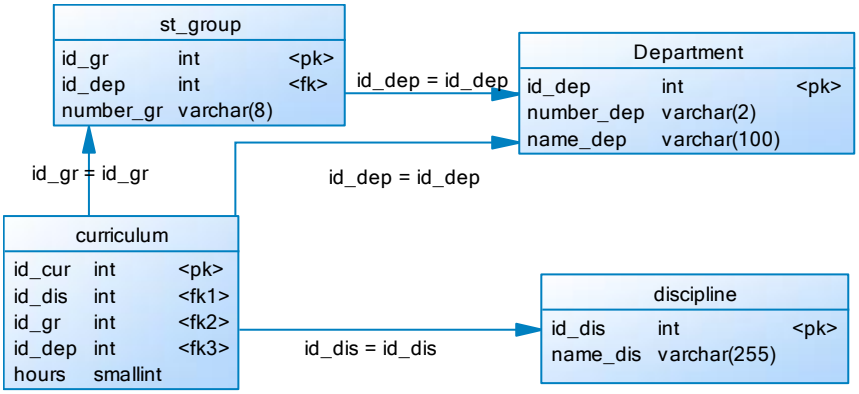
Запросы с использованием псевдонимов

Необходимость в таких запросах возникает когда одна и та же таблица должна использоваься в разных качествах (кафедра, ведущая дисциплину у группы и кафедра, по которой группа выпускается) или когда нужно чтобы некоторый атрибут принимал 2 значения одновременно (группа, которая изучает и Дифференциальные уравнения и Объектно-ориентированное программирование)

Общий вид псевдонима на таблицу выглядит select … from таблица as псевдоним. AS иногда можно опустить.

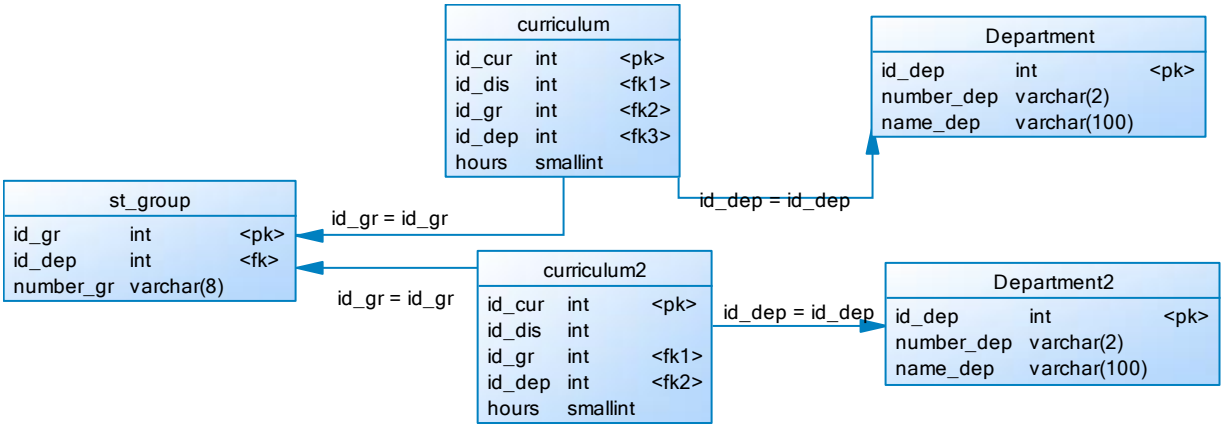
Пример

Схема (кафедра)



Написать запрос: Группы, у которых ведет дисциплины и 43 и 41 кафедра.

Для наглядности



Select distinct st_group.id_gr, number_gr from st_group
inner join curriculum on curriculum.id_gr= st_group.id_gr
inner join Department on Department.id_dep=curriculum.id_dep
inner join curriculum **as curriculum2** on **curriculum2**.id_gr= st_group.id_gr
inner join Department as **Department2** on **Department2**.id_dep=**curriculum2**.id_dep
where Department.number_dep=’43’ and Department2.number_dep=’41’

2. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 и 2 реализовать запросы а) .. в), указанные в варианте задания. Все запросы должны не содержать вложенных запросов или агрегатных функций. (Используйте псевдонимы)

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— текст запросов на SQL;

— наборы данных, возвращаемые запросами.

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №5 Разработка SQL запросов: запросы с подзапросами

1. Теоретическая часть

Агрегатные функции в операторе выборки языка SQL

Стандарт ISO содержит определение следующих пяти *агрегирующих функций:*
• COUNT — возвращает количество значений в указанном столбце;
• SUM — возвращает сумму значений в указанном столбце;
• AVG — возвращает усредненное значение в указанном столбце;
• MIN — возвращает минимальное значение в указанном столбце;
• MAX — возвращает максимальное значение в указанном столбце.

.Все эти функции оперируют со значениями в единственном столбце таблицы и возвращают единственное значение. Функции COUNT, MIN и MAX применимы как к числовым, так и к нечисловым полям, тогда как функции SUM и AVG могут использоваться только в случае числовых полей. За исключением COUNT (*) , при вычислении результатов любых функций сначала исключаются все пустые значения, после чего требуемая операция применяется только к оставшимся непустым значениям столбца. Вариант COUNT (*) является особым случаем использования функции COUNT — его назначение состоит в подсчете всех строк в таблице, независимо от того, содержатся там пустые, повторяющиеся или любые другие значения. Если до применения агрегирующей функции необходимо исключить повторяющиеся значения, следует перед именем столбца в определении функции поместить ключевое слово DISTINCT. Стандарт ISO допускает использование ключевого слова ALL с целью явного указания того, что исключение повторяющихся значений не требуется, хотя это ключевое слово подразумевается по умолчанию, если никакие иные определители не заданы. Ключевое слово DISTINCT не имеет смысла для функций MIN и MAX. Однако его использование может оказывать влияние на результаты выполнения функций SUM и AVG, поэтому следует заранее обдумать, должно ли оно присутствовать в каждом конкретном случае. Кроме того, ключевое слово DISTINCT в каждом запросе может быть указано не более одного раза. Следует отметить, что агрегирующие функции могут использоваться только в списке выборки SELECT и в конструкции HAVING Во всех других случаях применение этих функций недопустимо. Если список выборки SELECT содержит агрегирующую функцию, а в тексте запроса отсутствует конструкция GROUP BY, обеспечивающая объединение данных в группы, то ни один из элементов списка выборки SELECT не может включать каких-либо ссылок на столбцы, за исключением случая, когда этот столбец используется как параметр агрегирующей функции.__

5.3.4. Группирование результатов (конструкция GROUP BY)

Приведенные выше примеры сводных данных подобны итоговым строкам, обычно размещаемым в конце отчетов. В итогах все детальные данные отчета сжимаются в одну обобщающую строку. Однако очень часто в отчетах требуется формировать и промежуточные итоги. Для этой цели в операторе SELECT может указываться конструкция GROUP BY. Запрос, в котором присутствует конструкция GROUP BY, называется *группирующим запросом,* поскольку в нем группируются данные, полученные в результате выполнения операции SELECT, после чего для каждой отдельной группы создается единственная итоговая строка. Столбцы, перечисленные в конструкции GROUP BY, называются *группируемыми столбцами.* Стандарт ISO требует, чтобы конструкции SELECT и GROUP BY были тесно связаны между собой. При использовании в операторе SELECT конструкции GROUP BY каждый элемент списка в списке выборки SELECT должен иметь *единственное значение для всей группы.*

Все имена столбцов, приведенные в списке выборки SELECT, должны присутствовать и в конструкции GROUP BY, за исключением случаев, когда имя столбца используется только в агрегирующей функции. Противоположное утверждение не всегда справедливо — в конструкции GROUP BY могут присутствовать имена столбцов, отсутствующие в списке выборки SELECT. Если совместно с конструкцией GROUP BY используется конструкция WHERE, то она обрабатывается в первую очередь, а группированию подвергаются только те строки, которые удовлетворяют условию поиска. Стандартом ISO определено, что при проведении группирования все отсутствующие значения рассматриваются как равные. Если две строки таблицы в одном и том же группируемом столбце содержат значения NULL и идентичные значения во всех остальных непустых группируемых столбцах, они помещаются в одну и ту же группу.

конструкция HAVING

Конструкция HAVING предназначена для использования совместно с конструкцией GROUP BY для задания ограничений, указываемых с целью отбора тех *групп,* которые будут помещены в результирующую таблицу запроса. Хотя кон- струкции HAVING и WHERE имеют сходный синтаксис, их назначение различно.

В основном HAVING действует так же, как выражение WHEREв операторе SELECT. Тем не менее, выражение WHERE применяется к столбцам и выражениям для отдельной строки, в то время как оператор HAVING применяетсяк результату действия команды GROUP BY (агрегатной функции). Стандарт ISO требует, чтобы имена столбцов, применяемые в конструкции HAVING, обязательно присутствовали в списке элементов GROUP BY или применялись в агрегирующих функциях. На практике условия поиска в конструкции HAVING всегда включают, по меньшей мере, одну агрегирующую функцию; в противном случае эти условия поиска должны быть помещены в конструкцию WHERE и применены для отбора отдельных строк. (Помните, что агрегирующие функции не могут использоваться в конструкции WHERE.)

Конструкция HAVING не является необходимой частью языка SQL —- любой запрос, написанный с использованием конструкции HAVING, может быть пред- ставлен в ином виде, без ее применения.

Приведем пример использования агрегетных функций и группировки.

Допустим существует таблица Student_Uni

Student_uni		
id_student	int	<pk>
surname	varchar(30)	
name	varchar(30)	
patronym	varchar(30)	
Form_study	varchar(1)	
Faculty	varchar(1)	
department	varchar(2)	

,

Где From_study- форма обучения («О», «В» «З»), Faculty- факультет, department- кафедра.

Если мы хотим посчитать всех студентов вуза, необходимо написать запрос

SELECT count (id_student) as all_st from Student_Uni

, тогда в результате мы получим одно число равное общему количеству студентов.

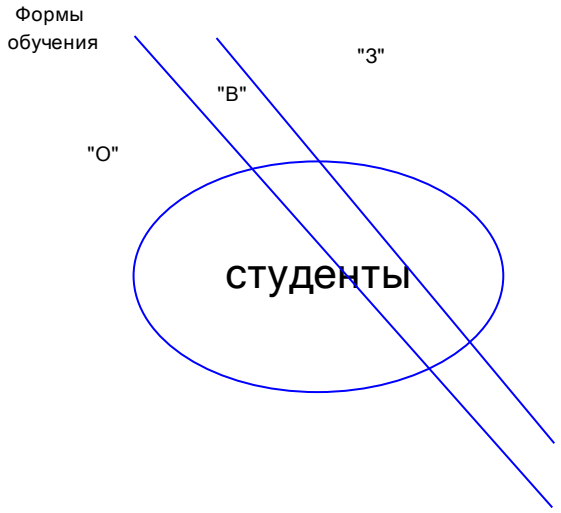
Если мы хотим узнать сколько учится студентов на каждой форме обучения надо написать

SELECT count (id_student) as all_st, Form_study from Student_Uni GROUP BY Form_study

И получим в результате, что-то вроде

all_st	Form_study
400	O
20	B
200	З

На самом деле сгруппировав по форме обучения, мы сделали равенство форм обучения критерием отнесения студента к той или иной группе, в которой потом будет производиться подсчет. Т.е. разбили все множество студентов по форме обучения



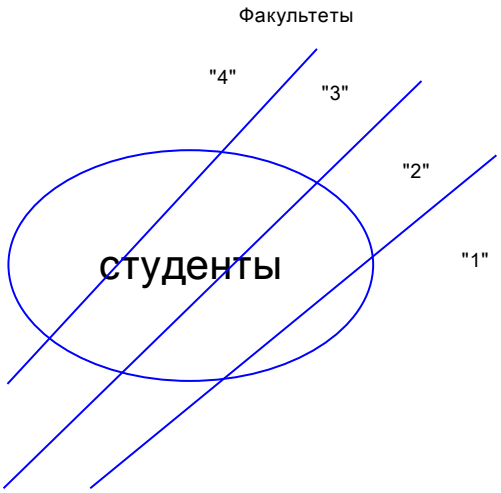
Если мы захотим узнать численность студентов на факультетах, запрос будет выглядеть:

SELECT count (id_student) as all_st, Faculty from Student_Uni GROUP BY Faculty

Допустим в вузе только 4 факультета, тогда мы получим результат типа

all_st	Faculty
10	1

0	
10 0	2
20 0	3
22 0	4

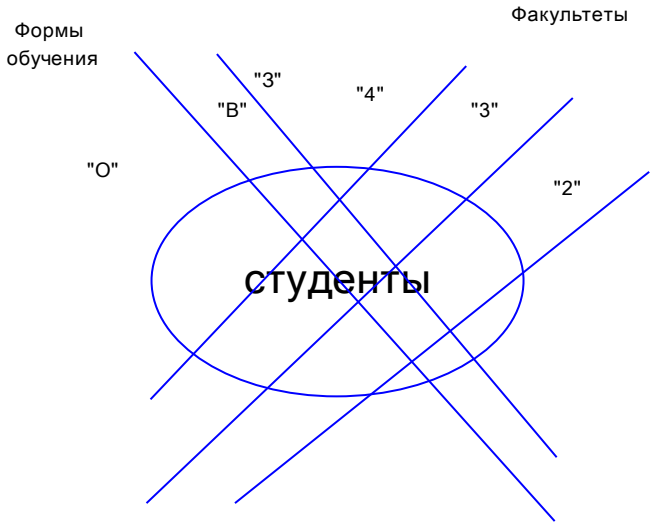


Ячейка уже объединяется по факультету.

А если мы сгруппируем и по факультету и по форме обучения

SELECT count (id_student) as all_st, Faculty, Form_study from Student_Uni GROUP BY Faculty, Form_study

То получим ячейку у которой одинаковы оба эти значения



И результат (числа не вписаны- это общий вид)

All_st	Faculty	Form_study
	1	О
	1	В
	1	З
	2	О
	2	В
	2	З
	3	О
	3	В
	3	З
	4	О
	4	В
	4	З

Хотя строк может быть меньше, если не на всех факультетах есть все формы обучения!!!

Каждый раз добавляя в группировку поля мы уменьшаем ячейку, поэтому например запрос типа

SELECT count (id_student) as all_st from Student_Uni GROUP BY id_student

Выдаст и кучу строк с единицами (размер ячейки подсчета студентов- один студент)

Поэтому обратите внимание !

Если у Вас группировка и агрегатная функция заданы для одного поля, то это ,вероятно, ошибка

Объединение, пересечение, разность запросов в языке SQL

В языке SQL можно использовать обычные операции над множествами — объединение (union), пересечение (intersection) и разность (difference), — позволяющие комбинировать результаты выполнения двух и более запросов в единую результирующую таблицу.

• *Объединением* двух таблиц A и B называется таблица, содержащая все строки, которые имеются в первой таблице (A), во второй таблице (B) или в обеих этих таблицах одновременно,

• *Пересечением,* двух таблиц называется таблица, содержащая все строки, присутствующие в обеих исходных таблицах одновременно.

• *Разностью* двух таблиц A и B называется таблица, содержащая все строки, которые присутствуют в таблице A, но отсутствуют в таблице B.

Все эти операции над множествами графически представлены на рис. 5.2. На таблицы, которые могут комбинироваться с помощью операций над множествами, накладываются определенные ограничения. Самое важное из них состоит в том, что таблицы должны быть *совместимы, по соединению* — т.е. они должны иметь одну и ту же структуру. Это означает, что таблицы должны иметь одинаковое количество столбцов, причем в соответствующих столбцах должны размещаться данные одного и того же типа и длины. Обязанность убедиться в том, что значения данных соответствующих столбцов принадлежат одному и тому же домену, возлагается на пользователя.

Три операции над множествами, предусмотренные стандартом ISO, носят название UNION, INTERSECT и EXCEPT. В каждом случае формат конструкции с операцией над множествами должен быть следующим:

operator .[ALL] [CORRESPONDING [BY {columnl [, . . .]}]]

При указании конструкции CORRES POND ING BY операция над множествами выполняется для указанных столбцов. Если задано только ключевое слово CORRESPONDING, а конструкция BY отсутствует, операция над множествами выполняется для столбцов, которые являются общими для обеих таблиц. Если указано ключевое слово ALL, результирующая таблица может содержать повторяющиеся строки.

Одни диалекты языка SQL не поддерживают операций INTERSECT и EXCEPT, а в других вместо ключевого слова EXCEPT используется ключевое слово MINUS.

Пример выберем имена и фамилии преподавателей или студентов с непустым отчеством. (логическое или)

```
(SELECT surname, name
FROM student
WHERE patronym IS NOT NULL)
UNION
(SELECT surname, name
FROM teacher
WHERE patronym IS NOT NULL);
```

Или

```
(SELECT *
FROM student
WHERE patronym IS NOT NULL)
UNION CORRESPONDING BY surname, name
```

(SELECT *
FROM teacher
WHERE patronym IS NOT NULL);

Выберем фамилии и имена и отчества преподавателей , которые еще учатся (в магистратуре)(логическое И)(считаем, что полных тезок нет)

(SELECT surname, name, patronym
FROM student)
INTERSECT
(SELECT surname, name, patronym
FROM teacher);
Или
(SELECT *
FROM student)
INTERSECT CORRESPONDING BY surname, name, patronym
(SELECT *
FROM teacher);

Выберем фамилии и имена и отчества студентов , которые не преподают (в магистратуре) (студенты минус преподаватели)(считаем, что полных тезок нет)

(SELECT surname, name, patronym
FROM student)
EXCEPT
(SELECT surname, name, patronym
FROM teacher);
Или
(SELECT *
FROM student)
EXCEPT CORRESPONDING BY surname, name, patronym
(SELECT *
FROM teacher);

Запросы с подзапросами в языке SQL

Здесь мы обсудим использование законченных операторов SELECT, внедренных в тело другого оператора SELECT. *Внешний* (второй) оператор SELECT использует результат выполнения *внутреннего* (первого) оператора для определения содержания окончательного результата всей операции. Внутренние запросы могут находиться в конструкциях WHERE и HAVING внешнего оператора SELECT — в этом случае они получают название *подзапросов,* или *вложенных запросов.* Кроме того, внутренние операторы SELECT могут использоваться в операторах INSERT, UPDATE и DELETE .

Также возможно нахождение подзапроса в конструкции FROM , где он должен получить псевдоним после скобок и с ним можно работать как с любой таблицей.

Существуют три типа подзапросов.

• *Скалярный подзапрос* возвращает значение, выбираемое из пересечения одного столбца с одной строкой, т.е. единственное значение. В принципе скалярный подзапрос может использоваться везде, где требуется указать единственное значение.

• *Строковый подзапрос* возвращает значения нескольких столбцов таблицы, но в виде единственной строки. Строковый подзапрос может использоваться везде, где применяется конструктор строковых значений, — обычно это предикаты.

• *Табличный подзапрос* возвращает значения одного или нескольких столбцов таблицы, размещенные в более чем одной строке. Табличный подзапрос может использоваться везде, где допускается указывать таблицу, например как операнд предиката IN.

Пример скалярного подзапроса

Вывести группу, в которой учится столько же человек, как и в группе Z4431

```
Select st_group.* from st_group left join student on student.id_gr=st_group.id_gr
Group by student.id_gr
Having count (id_student)=
(select count(*) from st_group gr
  left join student st on st.id_gr=gr.id_gr where  number_gr=’ Z4431’)
```

Left join, т.к в группе может не быть студентов

Табличный подзапрос

Вывести всех студентов из групп ,где учится Иванов

```
Select student.* from student where id_gr in
```

(select st_group.id_gr from from st_group inner join student on student.id_gr=st_group.id_gr where student.surname=’Иванов’)

К подзапросам применяются следующие правила и ограничения.

1. В подзапросах не должна использоваться конструкция ORDER BY, хотя она может присутствовать во внешнем операторе SELECT.

2. Список выборки SELECT подзапроса должен состоять из имен отдельных столбцов или составленных из них выражений, за исключением случая, когда в подзапросе используется ключевое слово EXISTS.

3. По умолчанию имена столбцов в подзапросе относятся к таблице, имя которой указано в конструкции FROM подзапроса. Однако разрешается ссылаться и на столбцы таблицы, указанной в конструкции FROM внешнего запроса, для чего используются уточненные имена столбцов (как описано ниже).

4. Если подзапрос является одним из двух операндов, участвующих в операции сравнения, то подзапрос должен указываться в правой части этой операции. Например, приведенный ниже вариант записи запроса является некорректным, поскольку подзапрос размещен в левой части операции сравнения со значением столбца salary.

SELECT staffNo, fName, IName, position, salary

FROM Staff

WHERE (SELECT AVG(salary) FROM Staff) < salary;

Ключевые слова ANY и ALL

Ключевые слова ANY и ALL могут использоваться с подзапросами, возвращающими один столбец чисел. Если подзапросу будет предшествовать ключевое слово ALL, условие сравнения считается выполненным только в том случае, если оно выполняется для всех значений в результирующем столбце подзапроса. Если тексту подзапроса предшествует ключевое слово ANY, то условие сравнения будет считаться выполненным, если оно удовлетворяется хотя бы для какого-либо (одного или нескольких) значения в результирующем столбце подзапроса. Если в результате выполнения подзапроса будет получено пустое значение, то для ключевого слова ALL условие сравнения будет считаться выполненным, а для ключевого слова ANY — невыполненным. Согласно стандарту ISO дополнительно можно использовать ключевое слово SOME, являющееся синонимом ключевого слова ANY.

Пример

Найдите всех работников, чья зарплата превышает зарплату хотя бы одного *работника отделения компании под номером 'вооз '.*

SELECT staffNo, fName, IName, position, salary

FROM Staff

WHERE salary > SOME(SELECT salary

FROM Staff

WHERE branchNo = 'B003');

Далее, *коррелированный* подзапрос – это особый вид подзапроса (табличного, однострочного или скалярного), а именно такой, в котором есть ссылка на некоторую «внешнюю» таблицу. В следующем примере заключенное в скобки выражение после ключевого слова IN и есть коррелированный подзапрос, потому что включает ссылку на внешнюю таблицу

S (запрос звучит так: «Получить названия поставщиков, которые поставляют деталь P1»).

SELECT DISTINCT S.SNAME

FROM S

WHERE ‘P1’ IN (SELECT PNO

FROM SP

WHERE SP.SNO = S.SNO)

Экзистенциальные запросы в языке SQL

Ключевые слова EXISTS и NOT EXISTS предназначены для использования только совместно с подзапросами или управляющими конструкциями. Результат их обработки представляет собой логическое значение TRUE или FALSE. Для ключевого слова EXISTS результат равен TRUE в том и только в том случае, если в возвращаемой подзапросом результирующей таблице присутствует хотя бы одна строка. Если результирующая таблица подзапроса пуста, результатом обработки ключевого слова EXISTS будет значение FALSE. Для ключевого слова NOT EXISTS используются правила обработки, обратные по отношению к ключевому слову EXISTS. Поскольку по ключевым словам EXISTS и NOT EXISTS проверяется лишь наличие строк в результирующей таблице подзапроса, то эта таблица может содержать произвольное количество столбцов.

Например, группа, где есть студенты, будет получена запросом

```
    Select st_group.* from st_group where
exists (select id_student from student where
student.id_gr=st_group.id_gr)
```

Часто условие NOT EXISTS используется для реализации разности.

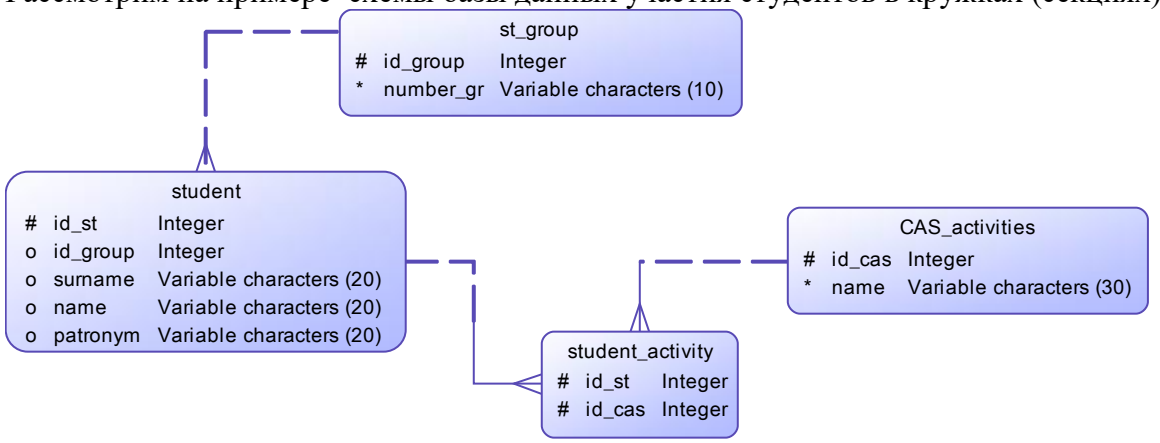
Например: Группа без студентов может быть найдена запросом

```
    Select st_group.* from st_group where
not exists (select id_student from student where
student.id_gr=st_group.id_gr)
```

Обратите внимание, что в подзапросе сквозная видимость

Одно из назначений NOT EXISTS – реализация реляционного деления (так называемые запросы на «все»).

Рассмотрим на примере схемы базы данных участия студентов в кружках (секциях).



На схеме CAS activities(от"Creativity, Action, Service") это секции.

Запрос звучит так:

Студенты, которые посещают все кружки.

Для самой простой по смыслу реализации такого запроса посмотрим на него с точки зрения реляционного деления. В данном случаемы должны поделить студентов на кружки, то есть студент- это делимое, а все кружки - делитель.

такой запрос можно разделить на 3 части- запроса:

A
NOT EXISTS
(B
NOT EXISTS (
C))

При этом каждая часть имеет свое назначение.

Части А и С отвечают за связку между делимым и делителем (студент и все кружки)

Запросы А и С как правило похожи с точностью до псевдонимов (сквозная видимость- псевдонимы обязательны), однако в С идет дополнительная связка с А по ключу делимого (в данном случае студента) и связка С и В по ключу делителя (в данном случае кружка)

Запрос В задает делитель.

Такой запрос будет выглядеть:

```
Select student.* from student
inner join student_activity on student.id_st=student_activity.id_st
where not exists
(select * from CAS_activities
where not exists
(Select * from student as st
inner join student_activity as st_a on st.id_st=st_a.id_st
where st.id_st=student.id_st
and st_a.id_cas= CAS_activities.id_cas))
```

В данном случае можно даже сократить запрос, убрав из первой части привязкук клуам, а из самой вложенной таблицу студент, так как в student_activity уже будут все нужные данные.

Тогда запрос будет выглядеть так:

```
Select student.* from student
where not exists
(select * from CAS_activities
where not exists
```

```
(Select * from student student_activity as st_a
where student.id_st=st_a.id_st
and st_a.id_cas= CAS_activities.id_cas))
```

Если запрос звучит например

Студенты, которые посещают все кружки, связанные с математикой

То меняется здесь только делитель. Вместо «все кружки» он будет «все кружки, связанные с математикой»

Тогда запрос изменится так

```
Select student.* from student
where not exists
(select * from CAS_activities
where name like '%математ%'
and not exists
(Select * from student student_activity as st_a
where student.id_st=st_a.id_st
and st_a.id_cas= CAS_activities.id_cas))
```

2. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 реализовать запросы г) .. ж), указанные в варианте задания.

Один из запросов на максимум/минимум реализовать с помощью директивы all

Запрос на «все» (реляционное деление) реализовать с помощью 2 not exists

Запросы на разность реализовать в 3 вариантах: Not in, except (MySQL не поддерживает, поэтому только синтаксис), с использованием левого/правого соединения

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— текст запросов на SQL;

— наборы данных, возвращаемые запросами.

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №6 Хранимые процедуры

1. Теоретическая часть

2. Теоретическая часть

Хранимые процедуры представляют собой набор команд SQL, которые могут компилироваться и храниться на сервере. Таким образом, вместо того, чтобы хранить часто используемый запрос, клиенты могут ссылаться на соответствующую хранимую процедуру. Это обеспечивает лучшую производительность, поскольку данный запрос должен анализироваться только однажды и уменьшается трафик между сервером и клиентом. Концептуальный уровень можно также повысить за счет создания на сервере библиотеки функций.

Триггер представляет собой хранимую процедуру, которая активизируется при наступлении определенного события. Например, можно задать хранимую процедуру, которая срабатывает каждый раз при удалении записи из транзакционной таблицы - таким образом, обеспечивается автоматическое удаление соответствующего заказчика из таблицы заказчиков, когда все его транзакции удаляются.

Хранимые программы (процедуры и функции) поддерживаются в MySQL 5.0. Хранимые процедуры – набор SQL-выражений, который может быть сохранен на сервере. Как только это сделано, клиенту уже не нужно повторно передавать запрос, а требуется просто вызвать хранимую программу.

Это может быть полезным тогда, когда:

- многочисленные приложения клиента написаны в разных языках или работают на других платформах, но нужно использовать ту же базу данных операций
- безопасность на 1 месте

Хранимые процедуры и функции (подпрограммы) могут обеспечить лучшую производительность потому, что меньше информации требуется для пересылки между клиентом и сервером. Выбор увеличивает нагрузку на сервер БД, но снижает затраты на стороне клиента. Используйте это, если много клиентских машин (таких как Веб-серверы) обслуживаются одной или несколькими БД.

Хранимые подпрограммы также позволяют вам использовать библиотеки функций, хранимые в БД сервера. Эта возможность представлена для многих современных языков программирования, которые позволяют вызывать их непосредственно (например, используя классы).

MySQL следует в синтаксисе за SQL:2003 для хранимых процедур, который уже используется в IBM's DB2.

Хранимые процедуры требуют наличия таблицы `proc` в базе `mysql`. Эта таблица обычно создается во время установки сервера БД. При создании, модификации, удалении хранимых подпрограмм сервер манипулирует с таблицей `mysql.proc`

Начиная с MySQL 5.0.3 требуются следующие привилегии:

CREATE ROUTINE для создания хранимых процедур

ALTER ROUTINE необходимы для изменения или удаления процедур. Эта привилегия автоматически назначается создателю процедуры (функции)

EXECUTE привилегия потребуется для выполнения подпрограммы. Тем не менее, автоматически назначается создателю процедуры (функции). Также, по умолчанию, `SQL SECURITY` параметр для подпрограммы `DEFINER` , который разрешает пользователям, имеющим доступ к БД вызывать подпрограммы, ассоциированные с этой БД.

Синтаксис хранимых процедур и функций

Хранимая подпрограмма представляет собой процедуру или функцию. Хранимые подпрограммы создаются с помощью выражений `CREATE PROCEDURE` или `CREATE FUNCTION`. Хранимая подпрограмма вызывается, используя выражение `CALL` , причем только возвращающие значение переменные используются в качестве выходных. Функция может быть вызвана подобно любой другой функции и может возвращать скалярную величину. Хранимые подпрограммы могут вызывать другие хранимые подпрограммы.

Начиная с MySQL 5.0.1, загруженная процедура или функция связана с конкретной базой данных. Это имеет некоторые последствия:

- Когда подпрограмма вызывается, то подразумевается, что надо произвести вызов `USE db_name` (и отменить использование базы, когда подпрограмма завершилась, и база больше не потребуется)

- Вы можете указывать обычные имена процедур с именем базы данных. Это может быть использовано, чтобы ссылаться на подпрограмму, которая - не в текущей базе данных. Например, для выполнения хранимой процедуры `p` или функции `f` которые связаны с БД `test`, вы можете сказать интерпретатору команд так: `CALL test.p()` или `test.f()`.
- Когда база данных удалена, все загруженные подпрограммы связанные с ней тоже удаляются. В MySQL 5.0.0, загруженные подпрограммы - глобальные и не связанны с базой данных. Они наследуют по умолчанию базу данных из вызывающего оператора. Если `USE db_name` выполнено в пределах подпрограммы, оригинальная текущая БД будет восстановлена после выхода из подпрограммы (Например текущая БД `db_11`, делаем вызов подпрограммы, использующей `db_22`, после выхода из подпрограммы остается текущей `db_11`)

MySQL поддерживает полностью расширения, которые разрешают юзать обычные `SELECT` выражения (без использования курсоров или локальных переменных) внутри хранимых процедур. Результирующий набор, возвращенный от запроса, а просто отправляется напрямую клиенту. Множественный `SELECT` запрос генерирует множество результирующих наборов, поэтому клиент должен использовать библиотеку, поддерживающую множественные результирующие наборы.

CREATE PROCEDURE – создать хранимую процедуру.

CREATE FUNCTION – создать хранимую функцию.

Синтаксис:

```
CREATE PROCEDURE имя_процедуры ([параметр_процедуры[,...]])
[характеристёика ...] тело_подпрограммы

CREATE FUNCTION имя_функции ([параметр_функции[,...]])
RETURNS тип
[характеристика ...] тело_подпрограммы

параметр_процедуры:
 [ IN | OUT | INOUT ] имя_параметра тип
параметр_функции:
 имя_параметра тип

тип:
 Любой тип данных MySQL

характеристика:
 LANGUAGE SQL
 | [NOT] DETERMINISTIC
 | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
 | SQL SECURITY { DEFINER | INVOKER }
 | COMMENT 'string'

тело_подпрограммы:
 Правильное  SQL выражение.
```

Рассмотрим все на практике.

Сначала создадим хранимую процедуру следующим запросом:

```
CREATE PROCEDURE `my_proc`(OUT t INTEGER(11))
    NOT DETERMINISTIC
    SQL SECURITY INVOKER
    COMMENT ''
BEGIN
select val1+val2 into 't' from `my` LIMIT 0,1;
END;
```

Применение выражения `LIMIT` в этом запросе сделано из соображений того, что не любой клиент способен принять многострочный результирующий набор.

После этого вызовем ее:

```
CALL my_proc(@a);
SELECT @a;
```

Для отделения внутреннего запроса от внешнего всегда используют разделитель отличный от обычно (для задания используют команду **DELIMITER** <строка/символ>)

Вот еще один пример с учетом всех требований.

```
delimiter //
CREATE PROCEDURE simpleproc (OUT param1 INT)
 BEGIN
SELECT COUNT(*) INTO param1 FROM t;
END;
//

delimiter ;

CALL simpleproc(@a);
SELECT @a;
```

```
+------+
|  @a  |
+------+
|  3   |
+------+
1 row in set (0.00 sec)
```

Готово.

Получение значения первичного ключа для дальнейшей вставки

Если это неавтоинкрементный ключ, то

```
 SELECT ifnull(max(id)+1,0) INTO new_id FROM my_table;
INSERT INTO my_table(id,col1, col2,...) VALUES (new_id,'val1', 'val2'...);
```

Представим mysql `INSERT` в одной из таблиц, а таблица имеет столбец `item_id`, который установлен в `autoincrement` и `primary key`.

```
INSERT INTO table_name (col1, col2,...) VALUES ('val1', 'val2'...);
set new_id=(SELECT LAST_INSERT_ID());
```

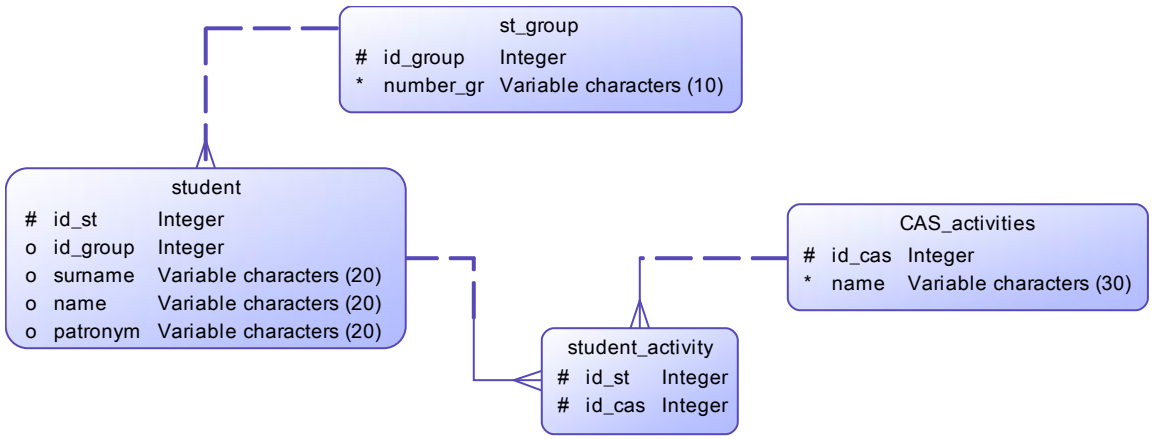
Это вернет вам значение `PRIMARY KEY` последней введенной строки:

Идентификатор, который был сгенерирован, поддерживается на сервере для каждого подключения. Это означает, что значение, возвращаемое функцией для данного клиента, является первым значением AUTO_INCREMENT, сгенерированным для самого последнего оператора, влияющим на столбец AUTO_INCREMENT этого клиента.

Вам также нужно иметь в виду, что это будет работать, только если `Last_INSERT_ID()` будет запущен после вашего запроса `Insert`. Это запрос возвращает идентификатор, вставленный в схему. Вы не можете получить конкретный последний вставленный идентификатор таблицы.

Наиболее часто

Примеры выполнены для базы данных со схемой



Процедура вставки с пополнением справочника.

Вставляется информация о студенте, если указанный номер группы отсутствует в БД, запись добавляется в таблицу с перечнем групп(справочник групп). Для **не** автоинкрементной таблицы группа

```
  delimiter //
CREATE PROCEDURE ins_stud (gr_num varchar(8),name_ varchar(15),
surname_ varchar(20),patronym_ varchar(25))
BEGIN
declare id_gr_new int;
declare id_st_new int;
if exists(select * from st_group where num_gr=gr_num)
  then select id_gr into id_gr_new from st_group where num_gr=gr_num;
  else   begin
  set id_gr_new=( select ifnull(max(id_gr)+1,0) from st_group );
  INSERT INTO st_group(id_gr,num_gr) VALUES (id_gr_new,gr_num);
        end;
  end if;
  set id_st_new=( select ifnull(max(id_st)+1,0) from student);
insert into student (id_st, surname, name,patronym,id_gr)
VALUES (id_st_new,surname_,name_,patronym_,id_gr_new);
END;//
delimiter ;
```

Процедура удаления с очисткой справочника

Удаляется информация о студенте, если в его группе нет больше студентов, запись удаляется из таблицы с перечнем групп.

```
 delimiter //
create procedure del_student_clear_gr(id_st_del int)
BEGIN
declare id_gr_del int;
select id_gr into id_gr_del from student where id_st=id_st_del;
delete from student where id_st=id_st_del;
if not exists(select * from student where id_gr=id_gr_del)
then delete from st_group where id_gr=id_gr_del;
end if;
END;//
delimiter ;
```

Процедура каскадного удаления.

Перед удалением записи о группе удаляются записи обо всех студентах этой группы и их кружках

```
 delimiter //
create procedure del_group_cascade ( id_gr_del int)
BEGIN
delete from student_activity where id_st in
       (select id_st from student where id_gr= id_gr_del);
delete from student where id_gr= id_gr_del;
 delete from st_group where id_gr=id_gr_del;
END;//
delimiter ;
```

Процедура вычисления и возврат значения агрегатной функции

Возврат количества групп

```
delimiter ;
use uni;
delimiter //
create procedure count_groups (out cnt_gr int)
BEGIN
select ifnull(count(id_gr),0) into cnt_gr from st_group;
END;//
delimiter ;

call count_groups( @cnt);
select @cnt;
```

```
delimiter //
create function count_groups1() returns int DETERMINISTIC
BEGIN
declare cnt_gr int DEFAULT 0;
set cnt_gr=(select ifnull(count(id_gr),0)  from st_group);
return cnt_gr;
END;//
delimiter ;
```

```
select count_groups1();
```

Формирование статистики во временной таблице

```
 delimiter //
create procedure cas_statistics ()
BEGIN
create temporary table if not exists cas_stat
(
id_stat int auto_increment primary key,
id_st int,
count_cas int,
count_cas_avg double default 0,
diff_cnt_avg double default 0
);
insert into cas_stat (id_st,count_cas)
select student.id_st, count(id_cas) as count_cas from student
left join student_activity on student.id_st= student_activity.id_st group by student.id_st;

update cas_stat set count_cas_avg=
(select avg(count_cas) from
 (select student.id_st, count(id_cas) as count_cas from student
left join student_activity on student.id_st= student_activity.id_st group by student.id_st)q);

update cas_stat set diff_cnt_avg=count_cas-count_cas_avg;

select * from cas_stat;
select avg(diff_cnt_avg*diff_cnt_avg) from cas_stat;

drop table cas_stat;
END;//
delimiter ;
```

3. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 , создать в БД ХП, реализующие:
— вставку с пополнением справочников (вставляется информация о студенте, если указанный номер группы отсутствует в БД, запись добавляется в таблицу с перечнем групп) (получаем ссылку на внешний ключ по значению данного из родительской таблицы);
— удаление с очисткой справочников – удаление всех зависимых данных(удаляется информация о студенте, если в его группе нет больше студентов, запись удаляется из таблицы с перечнем групп);
— каскадное удаление (перед удалением записи о группе удаляются записи обо всех студентах этой группы);
— вычисление и возврат значения агрегатной функции (т.к. агркгатная функция дает единственный результат);
— формирование статистики во временной таблице (например, для рассматриваемой БД — для каждого факультета: количество групп, количество обучающихся студентов, ко-личество изучаемых дисциплин, количество сданных дисциплин, средний балл по факульте-ту).

4. Содержание отчета

Содержание отчета:

— физическую модель БД.

—назначение, тексты ХП и их вызовов;

— наборы данных, возвращаемые ХП.

5. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №7 Триггеры. Обеспечение активной целостности данных базы данных

1. Теоретическая часть

Триггер — это хранимая процедура, которая не вызывается непосредственно, а исполняется при наступлении определенного события (вставка, удаление, обновление строки). Поддержка триггеров в MySQL началась с версии 5.0.2. MySQL использует стандарт ANSI SQL:2003 для процедур и других функций.

Назначение триггеров (наиболее частое):

- поддержание активно ссылочной целостности

- Вычисление/поддержание в актуальном состоянии вычислимых (производных) атрибутов (полей)
- логирование (запись) изменений

При этом необходимо помнить, что если вы собираетесь поддерживать в актуальном состоянии вычислимые поля, то триггеры должны обрабатывать все три события, иначе данные будут некорректными.

Триггеры могут быть двух уровней:
(i) уровня строки[1] (***for each row***) — запускаются для каждой строки таблицы, затрону-той изменением;
(ii) уровня оператора (for each statement) — запускаются для каждой из инструкций ***insert***, ***update***, ***delete***, применяемой к таблице.

Триггеры MYSQL

Синтаксис создания триггера
CREATE TRIGGER trigger_name trigger_time trigger_event
ON tbl_name FOR EACH ROW trigger_stmt

trigger_name — название триггера

trigger_time — Время срабатывания триггера. BEFORE — перед событием. AFTER — после события.

trigger_event — Событие:

insert — событие возбуждается операторами insert, data load, replace

update — событие возбуждается оператором update

delete — событие возбуждается операторами delete, replace.

Операторы DROP TABLE и TRUNCATE не активируют выполнение триггера

tbl_name — название таблицы

trigger_stmt выражение, которое выполняется при активации триггера

В течение выполнения триггера существуют две строки NEW и OLD, повторяющие структуру таблицы на которую поставлен триггер(в других диалектах SQL таблицы ***inserted*** и ***deleted)***. NEW содержит новые версии данных (вставленные оператором ***insert*** или измененные оператором ***update***). OLD содержит старые версии данных (удаленные оператором ***delete*** или подлежащие изменению оператором ***update***). Ссылка на указанные таблицы производится так же, как на основные таблицы БД.

Замечание: в настоящее время триггеры не активизированы каскадными действиями внешнего ключа.

Также может понадобиться поменять разделитель MySQL при создании триггеров. Оригинальный разделитель MySQL - это ; , но так как мы будем использовать разделитель для добавленных запросов, то может понадобиться явно указать разделитель, чтобы создавать запросы из командной строки.

Чтобы изменить разделитель, нужно выполнить команду перед командой триггера:

```
DELIMITER //
```

А после команды триггера надо ввести:

```
DELIMITER ;
```

Пример триггера, реализующего ссылочную целостность (каскадное изменение)

Создадим простой триггер, который при удалении корзины будет удалять все элементы корзины, которые имеют такой же cart_id:

```
CREATE TRIGGER `tutorial`.`before_delete_carts`
BEFORE DELETE ON `trigger_carts` FOR EACH ROW
BEGIN
DELETE FROM trigger_cart_items WHERE OLD.cart_id = cart_id;
END
```

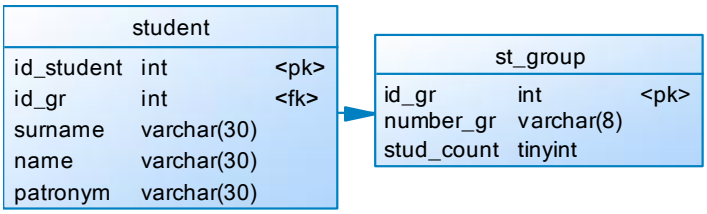
Пример триггера для вычисленных полей

Имеется простой пример, который связывает триггер с таблицей для инструкций `INSERT`. Это действует как сумматор, чтобы суммировать значения, вставленные в один из столбцов таблицы.

Следующие инструкции создают таблицу и триггер для нее:

```
CREATE TABLE account (acct_num INT, amount DECIMAL(10,2));
CREATE TRIGGER ins_sum BEFORE INSERT ON account
FOR EACH ROW SET @sum = @sum + NEW.amount;
```

Еще один пример с вычислимым столбцом для таблиц student и st_group




```
delimiter //
Create trigger my_trigger
 after update on student
FOR EACH ROW
Begin
Update st_group
Set stud_count=stud_count-1
where id_gr =OLD.id_gr;
Update st_group
Set stud_count=stud_count+1
where id_gr =NEW.id_gr;
End//
delimiter ;
```

Пример триггера логирования событий

```
— Удаляем триггер
DROP TRIGGER `update_test`;
— Создадим еще одну таблицу,
— в которой будут храниться резервные копии строк из таблицы test
CREATE TABLE `testing`.`backup` (
`id` INT( 11 ) UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
`row_id` INT( 11 ) UNSIGNED NOT NULL,
`content` TEXT NOT NULL
) ENGINE = MYISAM
— триггеры
DELIMITER |
CREATE TRIGGER `update_test` before update ON `test`
FOR EACH ROW BEGIN
  INSERT INTO backup Set row_id = OLD.id, content = OLD.content;
END;

CREATE TRIGGER `delete_test` before delete ON `test`
FOR EACH ROW BEGIN
  INSERT INTO backup Set row_id = OLD.id, content = OLD.content;
END
```

Пример триггера обеспечения безопасности данных

Необходимо при добавлении записи в табл **user**, пароль преобразовывать в хеш **md5()**, также имя и отчество преобразовывать в инициалы.

```
DELIMITER //
CREATE TRIGGER `test_user_pass` BEFORE INSERT ON `test`.`user`
FOR EACH ROW
BEGIN
SET NEW.name = LEFT(NEW.name,1);
SET NEW.otch = LEFT(NEW.otch,1);
SET NEW.pass = md5(NEW.pass);
END//
DELIMITER ;
```

Синтаксис DROP TRIGGER

```
DROP TRIGGER [schema_name.]trigger_name
```

Это уничтожает триггер. Имя базы данных опционально. Если оно не задано, триггер удаляется из заданной по умолчанию базы данных, Вызов `DROP TRIGGER` был добавлен в MySQL 5.0.2. Использование требует привилегии `SUPER`.

Обратите внимание: До MySQL 5.0.10, имя таблицы требовалось вместо имени схемы (`table_name.trigger_name`).

Кроме того, триггеры, созданные в MySQL 5.0.16 или выше, не могут быть удалены в MySQL 5.0.15 или ниже. Если Вы желаете выполнить такой возврат, Вы также должны в этом случае удалить все триггеры и заново их пересоздать после смены версий.

2. Выполнение лабораторной работы

По аналогии с примерами, приведенными в п. 1 реализовать для своей базы данных триггеры для всех событий (insert,delete, update) до и после. Часть из которых будет обеспечивать ссылочную целостность, остальные могут иметь другое назначение из 3 предложенных. Вычислимые поля можно добавить при необходимости.

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— назначение и тексты триггеров;

— SQL операторы и наборы данных, иллюстрирующие работу триггеров..

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа №8 Проектирование взаимодействия базы данных и приложения

1. Теоретическая часть

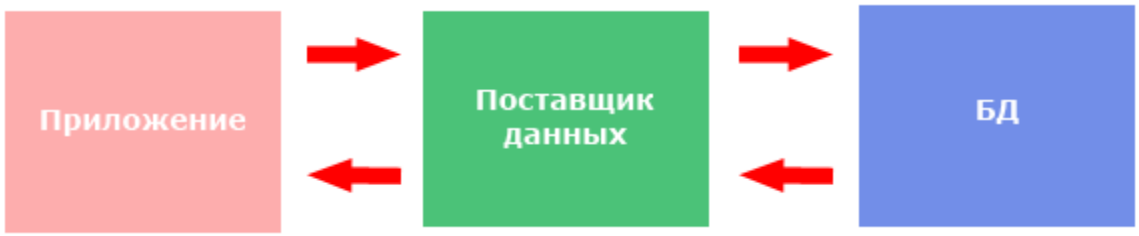
Наиболее часто встречающиеся способы доступа к БД из приложения:

- скриптовые языки со встроенной поддержкой работы с БД (php, например)
- ADO.Net объекты
- ODBC/JDBC объекты
- ORM (Object Relational mapping) привязка классов .Net к таблицам БД и от типов данных CLR в типы SQL.

Рассмотрим пару примеров

1.1ADO.NET

Для работы с базой данных нам потребуется **поставщик данных** (data provider), он обеспечивает подключение к БД, позволяет выполнять команды и получать результаты. По сути это обычный файл (.dll), внутри которого содержаться типы, которые настроены на взаимодействие с какой-то одной конкретной СУБД: MySQL, Oracle, Microsoft SQL Server и так далее.



В Microsoft ADO.NET основное количество поставщиков данных содержится в сборке Sytem.Data.dll, но в этой сборке нет поставщика данных для работы с MySQL.

Скачать его можно с сайта MySQL https://dev.mysql.com/downloads/connector/

Так же для подключения к базе данных потребуется узнать ip адрес сервера

Затем настроим **подключение к базе данных**

1. Создадим и заполним объект MySqlConnectionStringBuilder, который будет хранить следующие значения: имя сервера, где лежит база данных, имя пользователя и пароль для подключения к БД, а так же имя базы данных.

```
MySqlConnectionStringBuilder mysqlCSB;
mysqlCSB = new MySqlConnectionStringBuilder();
mysqlCSB.Server = "ip адрес сервера";
mysqlCSB.Database = "имя БД";
mysqlCSB.UserID = "имя пользователя";
mysqlCSB.Password = "пароль";
```

2. Создадим строку запроса, в ней мы выбираем все комментарии за сегодняшний день.

```
string queryString = @"SELECT comment_author,
comment_date,
comment_content
FROM   wp_comments
WHERE  comment_date >= CURDATE()";
```

Создадим объект DataTable, который будет возвращать наш метод и принимать datagridView.

```
DataTable dt = new DataTable();
```

4. Создадим объект подключения, используя класс MySqlConnection.

```
using(MySqlConnection con = new MySqlConnection())
{
}
```

4.1 Настроим созданный объект, передав в свойство ConnectionString наш созданный ранее объект типа MySqlConnectionStringBuilder.

```
 con.ConnectionString                    =
mysqlCSB.ConnectionString;
```

5. Открываем соединение с базой данных

```
con.Open();
```

6. Создаем объект команду, в конструктор передаем строку запроса и объект подключения

```
MySqlCommand com = new MySqlCommand(queryString, con);
```

7. Выполним метод ExecuteReader, который позволит получить объект чтения данных MySqlDataReader

```
using(MySqlDataReader dr = com.ExecuteReader())
{
//есть записи?
if (dr.HasRows)
{
//заполняем объект DataTable
dt.Load(dr);
}
}
```

8 Осталось поместить полученные данные в datagridView.

```
private void button1_Click(object sender, EventArgs e)
{
dataGridView1.DataSource = GetComments();
}
```

А можно не привязывать datagridView (шаги 7 и 8), а сделать ручную обработку

```
MySqlDataReader reader = command.ExecuteReader();
                while (reader.Read())
                {
                    Console.WriteLine("\t{0}\t{1}\t{2}",
                        reader[0], reader[1], reader[2]);
                }
                reader.Close();
```

1.2 Hibernate

Nhibernate это решение для Объектно Реалицонного мапинга для платформы .NET. Этот фреймворк позволяет делать мапинг Объектно ориентированных моделей к традиционным БД. Его основное преимущество маппирование классов .Net к таблицам БД и от типов данных CLR в типы SQL.

Предположим, что у нас есть база данных с таблицей Employee (работник)

Столбец ID должен быть первичным ключом, и должен быть автоинкрементом.

Теперь запускаем VisualStudio и создаем новое приложение WindowsFormApplication, проект назовем NhibernateBasics. Добави новый класс, назовите его Employee.cs и вставим следующий код.

```
namespace NHibernateBasics
{
    public class Employee
    {
        public virtual int ID { get; set; }
        public virtual string Name { get; set; }
    }
}
```

Nhibernate не нужно специальных интерфейсов для бизнес классов (объекты предметной области). Эти объекты не зависят от механизма загрузки и сохранения данных. Однако требуется чтобы свойства класса были описаны как виртуальные, чтобы они могли создавать Proxi. Из-за отсутствия специфического кода для гибернации, кто то должен взять на себя ответственность за трансляцию из БД в бизнес-объекты и обратно. Эта трансляция может быть выполнена с помощью связывания через XML файл или связывания атрибутов на классы и свойства.

Используем маппинг файл чтобы не захламлять класс бизнес объекта. Добавьте новый XML файл в проект. XML файл будет использоваться как маппинг файл. Имя файла должно быть Employee.hbm.xml. Файл класса .cs и маппинг файл .hbm.xml должны лежать в одной папке и должно быть одинаково названы.

Код xml файла:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.2"
        namespace="NHibernateBasics" assembly="NHibernateBasics">
  <class name="Employee" table="Employee">
    <id name="ID" column="ID">
      <generator class="identity"/>
    </id>
    <property name="Name" column="Name"  />
  </class>
</hibernate-mapping>
```

В свойствах XML файла в проекте необходио выставить свойство Build Action = Embedded Resource.

Добавьте новый конфигурационный файл приложения (app.config). Скопируйте следующий код.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="hibernate-configuration"
        type="NHibernate.Cfg.ConfigurationSectionHandler, NHibernate" />
  </configSections>
  <hibernate-configuration xmlns="urn:nhibernate-configuration-2.2">
    <session-factory>
      <property name="connection.provider">
       NHibernate.Connection.DriverConnectionProvider</property>
      <property name="dialect">NHibernate.Dialect.MsSql2005Dialect</property>
      <property name="query.substitutions">hqlFunction=SQLFUNC</property>
      <property name="connection.driver_class">
       NHibernate.Driver.SqlClientDriver</property>
      <property name="connection.connection_string">
       Data Source=(Local)\SQL2008;Initial Catalog=NHibernateBasics;
       Integrated Security=True</property>
      <property name="show_sql">true</property>
      <mapping assembly="NHibernateBasics" />
    </session-factory>
  </hibernate-configuration>
</configuration>
```

Настройте свойство Connection-string чтобы подключиться к вашей БД. Выставите свойство CATALOG=, значением NHibernateBasics. Выставите свойство mapping assembly=, опять же значением NHibernateBasics.

NHibernte гарантирует нам что две ссылки на объект будут указывать на один и тот же объект если ссылки установлены в одной сессии. Если мы сохраним объекты в одной сессии и загрузим в другой, то два объекта будут разными.

Посмотрим код операций:
Загрузка:

```
using(mySession.BeginTransaction())
{
    // Создаем критерию и загружаем данные
    ICriteria criteria = mySession.CreateCriteria<employee>();
    IList<employee> list = criteria.List<employee>();
    for (int i = 0; i < myFinalObjects.Length; i++)
    {
        myFinalObjects[i] = list[i];
        MessageBox.Show("ID: " + myFinalObjects[i].ID + "
                        Name: " + myFinalObjects[i].Name);
    }
    mySession.Transaction.Commit();
 }
```

Отображение:

```
using (mySession.BeginTransaction())
{
     ICriteria criteria = mySession.CreateCriteria<employee>();
     IList<employee> list = criteria.List<employee>();
     StringBuilder messageString = new StringBuilder();
     // Load and display the data
     foreach (Employee employee in list)
```

```
     {
         messageString.AppendLine("ID: " + employee.ID + " Name: " + employee.Name);
     }
     MessageBox.Show(messageString.ToString());
}
```

Сохранение:

```
using(mySession.BeginTransaction())
{
    // Insert two employees in Database
    mySession.Save(myInitialObjects[0]);
    mySession.Save(myInitialObjects[1]);
    mySession.Transaction.Commit();
}
```

Удаление:

```
using (mySession.BeginTransaction())
{
    // Delete one object from Database
    mySession.Delete(myInitialObjects[0]);
    mySession.Transaction.Commit();
}
```

2. Выполнение лабораторной работы

Создать приложение –форму с 2 способами подсоединения приложения к базе данных и: с двумя способами подсоединения базы данных:

1)Соединение через компоненты ADO.NET или его аналога с помощью строки связи. При этом отображаться должны 2 таблицы/элемента данных (не обязательно в форме таблицы, может быть просто текст): одна через DataGridView ,с подключением источника данных, а другая через чтение результатов запроса SqlCommand в цикле.

2) С использованием технологии ORM (Object-Relational Mapping) с отображение третьей таблицы /элемента данных

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— код приложения с подключением через ADO.net

— код приложения, включая классы, с подключением с использованием технологии ORM

— код xml файла для привязки классов к объектам

—Скриншоты приложения

4. Варианты заданий

Варианты заданий приведены в Приложении 2.

Лабораторная работа № 9 Объектно-реляционные базы данных. Проектирование и создание

1. Теоретическая часть

Наследование

PostgreSQL реализует наследование таблиц, которое может оказаться полезным инструментом для разработчиков базы данных. (Возможность наследования определяется стандартом SQL:1999 и более поздними стандартами и во многих отношениях отличается от возможностей PostgreSQL)

Начнём с примера: предположим мы пытаемся создать модель данных для городов. В каждом штате есть несколько городов, но только одна столица. Мы хотим предоставить возможность быстрого нахождения города-столицы для любого отдельного штата. Всё это можно сделать, создав две таблицы, одну для столиц штата и другую для городов, которые не являются столицами. Однако, что произойдёт, когда мы захотим получить данные о каком-либо городе, в не зависимости от того, является он столицей или нет? Возможность наследования может помочь решить эту проблему. Мы создаём таблиц і `capitals`, которая наследует от таблицы `cities`:

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

В этом случае, таблица `capitals` *наследует* все колонки из родительской таблицы `cities`. В таблице столиц штатов также есть дополнительная колонка с аббревиатурой названия штата — `state`.

В PostgreSQL любая таблица может наследовать от нуля или более других таблиц, а запрос может ссылаться либо на все строки в таблице либо на все строки в таблице вместе с её потомками. Последнее является поведением по умолчанию. Например, следующий запрос ищет имена всех городов, включаяя столицы штатов, которые расположены на высоте свыше 500 футов:

```
SELECT name, altitude
```

```
    FROM cities
    WHERE altitude > 500;
```

С данными для примера, взятыми из учебного руководства PostgreSQL ,этот запрос возвратит:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

С другой стороны, следующий запрос находит все города, которые не являются столицами штатов и которые также расположены на высоте свыше 500ft:

```
SELECT name, altitude
    FROM ONLY cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
```

Здесь слово "ONLY" перед таблицей cities говорит, что запрос должен выполняться только для таблицы cities, а не для таблиц, раположенных ниже cities, в иерархии наследования. Многие из команд, которые мы использовали ранее — `SELECT`, `UPDATE` и `DELETE` — поддерживают нотацию "ONLY".

В некоторых случаях, вы можете захотеть узнать из какой таблицы получена отдельная строка результата. В каждой таблице существует системная колонка с именем `tableoid`, которая расскажет о таблице:

```
SELECT c.tableoid, c.name, c.altiude
FROM cities c
WHERE c.altiude > 500;
```

запрос возвратит:

```
 tableoid |   name    |  altiude
----------+-----------+----------
   139793 | Las Vegas |     2174
   139793 | Mariposa  |     1953
   139798 | Madison   |      845
```

(Если вы попытаетесь воспроизвести этот пример, то предположительно вы получите другие значения OID.) Выполнив объединение с таблицей `pg_class` вы можете увидеть сами имена таблиц:

```
SELECT p.relname, c.name, c.altitude
FROM cities c, pg_class p
WHERE c.altiude > 500 and c.tableoid = p.oid;
```

запрос возвратит:

```
 relname  |   name    | altitude
----------+-----------+----------
 cities   | Las Vegas |     2174
```

```
 cities   | Mariposa  |     1953
 capitals | Madison   |      845
```

Наследование не распространяется автоматически на данные из команд `INSERT` или `COPY` на другие таблицы в иерархии наследования. В следующем примере, выполнение `INSERT` вызовет ошибку:

```
INSERT INTO cities (name, population, altitude, state)
VALUES ('New York', NULL, NULL, 'NY');
```

Мы могли бы надеяться, что данные как-нибудь будут перенаправлены в таблицу `capitals`, но этого не произойдёт: `INSERT` всегда вставляет данные точно в указанную таблицу. В некоторых случах, возможно перенаправить вставляемые данные, с помощью правила (см Chapter 34). Однако, это не поможет в данном выше случае, потому что таблица `cities` не содержит колонки `state` и таким образом команда будет отвергнута, перед тем как к ней можно будет применить правило.

Для таблиц внутри иерархии наследования, могут быть заданы ограничения целостности check. Все ограничения check для родительской таблицы автоматически наследуются всеми её потомками. Однако, другие типы ограничений целостности не наследуются.

Таблица может наследовать более чем от одной родительской таблицы и в этом случае она будет содержать суммарный список колонок из родительских таблиц. К этому списку добавляются любые колонки, задаваемые в самой таблице-потомке. Если в нескольких родительских таблицах, встречаются колонки с одинаковым именем или такая колонка есть в родительской таблице и в определении таблицы-потомка, то эти колонки "сливаются" таким образом, что только одна такая колонка будет в таблице-потомке. При слиянии, колонки должны иметь одинаковый тип, иначе будет выдано сообщение об ошибке. Колонка после слияния получает копию всех ограничений целостности check от каждой слитой колонки.

Наследование таблицы задаётся, используя команду *CREATE TABLE*, с ключевым словом `INHERITS`. Однако, похожая команда `CREATE TABLE AS` не позволяет указывать наследование.

В качестве альтернативы, для уже созданной таблицы можно задать новую родительскую таблицу с помощью *ALTER TABLE*, используя подформу `INHERITS`. Чтобы выполнить эту команду, новая таблица-потомок уже должна включать колонки с тем же именем и типом, что и родительская таблица. Она также должна включать ограничения целостности check с тем же именем и выражением check как и в родительской таблице. Похожим образом, связь наследования может быть удалена из таблицы-потомка, используя `ALTER TABLE` с подформой `NO INHERIT`.

Подходящий способ создания новой совместимой таблицы для таблицы-потомка состоит в использовании опции `LIKE` в команде `CREATE TABLE`. Такая команда создаёт таблицу с теми же колонками и с теми же типами (смотрите однако замечание про предостережения ниже). В качестве альтернативы, совместимую таблицу можно создать если сперва создать новую таблицу-потом с помощью `CREATE TABLE`, а затем удалить связь наследования через `ALTER TABLE`.

Родительская таблица не может быть удалена пока существует хотя бы одна таблица-потомок. Если вы хотите удалить таблицу и всех её потомков, то наиболее простой способ состоит в удалении родительской таблицы с опцией `CASCADE`. Если колонки в таблицах-потомках наследуются из родительских таблиц, то их нельзя удалить или изменить.

Любые изменения, выполненные командой *ALTER TABLE* в описании колонок и ограничениях целостности check, будут распространяться вниз по иерархии наследования. Команда `ALTER TABLE`

следует тем же самым правилами по слиянию дублирующихся колонок, что применяются во время выполнения команды `CREATE TABLE`

Пользовательские типы

Составные типы

Составной тип представляет структуру табличной строки или записи; по сути это просто список имён полей и соответствующих типов данных. PostgreSQL позволяет использовать составные типы во многом так же, как и простые типы. Например, в определении таблицы можно объявить столбец составного типа.

Ниже приведены два простых примера определения составных типов:

```
CREATE TYPE complex AS (
r double precision,
i double precision
);
CREATE TYPE inventory_item AS (
name text,
supplier_id integer,
price numeric
);
```

Определив такие типы, мы можем использовать их в таблицах:

```
CREATE TABLE on_hand (
item inventory_item,
count integer
);
```

Объявление перечислений

Тип перечислений создаются с помощью команды CREATE TYPE, например так:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');
```

Созданные типы enum можно использовать в определениях таблиц и функций, как и любые другие:

```
CREATE TYPE mood AS ENUM ('sad', 'ok', 'happy');

CREATE TABLE person (
name text,
current_mood mood
);
INSERT INTO person VALUES ('Moe', 'happy');
SELECT * FROM person WHERE current_mood = 'happy';
```

Все типы перечислений считаются уникальными и поэтому значения разных типов нельзя сравнивать.

2. Выполнение лабораторной работы

Спроектировать физическую модель базы данных, находящуюся в третьей нормальной форме и включающей наследование и хотя бы один пользовательский тип в соответствии с заданным вариантом. Написать соответствующий скрипт создания базы данных

Варианты заданий приведены в Приложении 3.

3. Содержание отчета

Содержание отчета:

— Текст задания.

— физическую модель БД.

— Скрипт создания базы данных

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 10 Объектно-реляционные базы данных. Манипуляция данными и пользовательские операторы

1. Теоретическая часть

Пользовательские операторы

CREATE OPERATOR — создать оператор

Синтаксис

```
CREATE OPERATOR имя (
    PROCEDURE = имя_функции
    [, LEFTARG = тип_слева ] [, RIGHTARG = тип_справа ]
    [, COMMUTATOR = коммут_оператор ] [, NEGATOR = обратный_оператор ]
    [, RESTRICT = процедура_ограничения ] [, JOIN = процедура_соединения ]
    [, HASHES ] [, MERGES ]
)
```

Описание

`CREATE OPERATOR` определяет новый оператор, *имя*. Владельцем оператора становится пользователь, его создавший. Если указано имя схемы, оператор создаётся в ней, в противном случае — в текущей схеме.

Имя оператора образует последовательность из нескольких символов (не более чем `NAMEDATALEN`-1, по умолчанию 63) из следующего списка:

+ - * / < > = ~ ! @ # % ^ & | ` ?

Однако выбор имени ограничен ещё следующими условиями:

- Сочетания символов `--` и `/*` не могут присутствовать в имени оператора, так как они будут обозначать начало комментария.
- Многосимвольное имя оператора не может заканчиваться знаком `+` или `-`, если только оно не содержит также один из этих символов:

  ~ ! @ # % ^ & | ` ?

  Например, `@-` — допустимое имя оператора, а `*-` — нет. Благодаря этому ограничению, PostgreSQL может разбирать корректные SQL-запросы без пробелов между компонентами.

- Использование `=>` в качестве имени оператора считается устаревшим и может быть вовсе запрещено в будущих выпусках.

Оператор `!=` отображается в `<>` при вводе, так что эти два имени всегда равнозначны.

Необходимо определить либо `LEFTARG`, либо `RIGHTARG`, а для бинарных операторов оба аргумента. Для правых унарных операторов должен быть определён только `LEFTARG`, а для левых унарных — только `RIGHTARG`.

Процедура *`имя_функции`* должна быть уже определена с помощью `CREATE FUNCTION` и иметь соответствующее число аргументов (один или два) указанных типов.

Другие предложения определяют дополнительные характеристики оптимизации.

Чтобы создать оператор, необходимо иметь право `USAGE` для типов аргументов и результата, а также право `EXECUTE` для нижележащей функции. Если указывается коммутативный или обратный оператор, нужно быть его владельцем.

Параметры

`имя`

Имя определяемого оператора. Допустимые в нём символы перечислены ниже. Указанное имя может быть дополнено схемой, например так: `CREATE OPERATOR myschema.+ (...)`. Если схема не указана, оператор создаётся в текущей схеме. При этом два оператора в одной схеме могут иметь одно имя, если они работают с разными типами данных. Такое определение операторов называется *перегрузкой*.

`имя_функции`

Функция, реализующая этот оператор.

`тип_слева`

Тип данных левого операнда оператора, если он есть. Этот параметр опускается для левых унарных операторов.

`тип_справа`

Тип данных правого операнда оператора, если он есть. Этот параметр опускается для правых унарных операторов.

коммут_оператор

Оператор, коммутирующий для данного.

обратный_оператор

Оператор, обратный для данного.

процедура_ограничения

Функция оценки избирательности ограничения для данного оператора.

процедура_соединения

Функция оценки избирательности соединения для этого оператора.

HASHES

Показывает, что этот оператор поддерживает соединение по хешу.

MERGES

Показывает, что этот оператор поддерживает соединение слиянием.

Чтобы задать имя оператора с указанием схемы в *коммут_оператор* или другом дополнительном аргументе, применяется синтаксис OPERATOR(), например:

```
COMMUTATOR = OPERATOR(myschema.===) ,
```

Пользовательские операторы пример

Любой оператор представляет собой «синтаксический сахар» для вызова нижележащей функции, выполняющей реальную работу; поэтому прежде чем вы сможете создать оператор, необходимо создать нижележащую функцию. Однако оператор — *не исключительно* синтаксический сахар, так как он несёт и дополнительную информацию, помогающую планировщику запросов оптимизировать запросы с этим оператором. Рассмотрению этой дополнительной информации будет посвящён следующий раздел.

Postgres поддерживает левые унарные, правые унарные и бинарные операторы. Операторы могут быть перегружены; то есть одно имя оператора могут иметь различные операторы с разным количеством и типами операндов. Когда выполняется запрос, система определяет, какой оператор вызвать, по количеству и типам предоставленных операндов.

В следующем примере создаётся оператор сложения двух комплексных чисел. Предполагается, что мы уже создали определение типа `complex`. Сначала нам нужна функция, собственно выполняющая операцию, а затем мы сможем определить оператор:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS 'имя_файла', 'complex_add'
    LANGUAGE C IMMUTABLE STRICT;

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Теперь мы можем выполнить такой запрос:

```
SELECT (a + b) AS c FROM test_complex;

        c
-----------------
 (5.2,6.05)
 (133.42,144.95)
```

Мы продемонстрировали создание бинарного оператора. Чтобы создать унарный оператор, просто опустите `leftarg` (для левого унарного) или `rightarg` (для правого унарного). Обязательными в `CREATE OPERATOR` являются только предложение `procedure` и объявления аргументов. Предложение `commutator`, добавленное в данном примере, представляет необязательную подсказку для оптимизатора запросов. Подробнее о `commutator` и других подсказках для оптимизатора рассказывается в следующем разделе.

Функции для перечислений

Для типов перечислений предусмотрено несколько функций, которые позволяют сделать код чище, не «зашивая» в нём конкретные значения перечисления. Эти функции перечислены в таблице ниже. В этих примерах подразумевается, что перечисление создано так:

```
CREATE TYPE rainbow AS ENUM ('red', 'orange', 'yellow', 'green',
      'blue', 'purple');
```

Функция	Описание	Пример	Результат примера
`enum_first(anyenum)`	Возвращает первое значение заданного перечисления	`enum_first( null::rainbow)`	`red`
`enum_last(anyenum)`	Возвращает последнее значение заданного перечисления	`enum_last( null::rainbow)`	`purple`
`enum_range(anyenum)`	Возвращает все значения заданного перечисления в упорядоченном массиве	`enum_range( null::rainbow)`	`{red,orange,yellow, green,blue,purple}`
`enum_range(anyenum, anyenum)`	Возвращает набор значений, лежащих между двумя заданными, в виде упорядоченного массива. Эти значения должны принадлежать одному перечислению. Если первый параметр равен NULL, функция возвращает первое значение перечисления, а если NULL второй — последнее.	`enum_range( 'orange'::rainbow, 'green'::rainbow)`	`{orange,yellow, green}`
		`enum_range(NULL, 'green'::rainbow)`	`{red,orange,yellow, green}`
		`enum_range( 'orange'::rainbow, NULL)`	`{orange,yellow, green,blue,purple}`

Пользовательские агрегатные функции

Агрегатные функции в PostgreSQL определяются в терминах *значений состояния* и *функций перехода состояния*. То есть агрегатная функция работает со значением состояния, которое меняется при обработке каждой последующей строки. Чтобы определить агрегатную функцию, нужно выбрать тип данных для значения состояния, начальное значение состояния и функцию перехода состояния. Функция перехода состояния принимает предыдущее значение состояния и входное агрегируемое значение для текущей строки и возвращает новое значение состояния. Также можно указать *функцию завершения*, на случай, если ожидаемый результат агрегатной функции отличается от данных, которые сохраняются в изменяющемся значении состояния. Функция завершения принимает конечное значение состояния и возвращает то, что она хочет вернуть в виде результата агрегирования. В принципе, функции перехода и завершения представляют собой просто обычные функции, которые также могут применяться вне контекста агрегирования. (На практике для большей производительности часто создаются специализированные функции перехода, которые работают, только когда вызываются при агрегировании.)

Таким образом, помимо типов данных аргументов и результата, с которыми имеет дело пользователь агрегатной функции, есть также тип данных внутреннего состояния, который может отличаться от этих типов.

Если мы определяем агрегат, не использующий функцию завершения, наш агрегат будет вычислять бегущее значение функции по столбцам каждой строки. Примером такой агрегатной функции является `sum`. Вычисление `sum` начинается с нуля, а затем к накапливаемой сумме всегда прибавляется значение из текущей строки. Например, если мы хотим сделать агрегатную функцию

sum для комплексных чисел, нам потребуется только функция сложения для такого типа данных. Такая агрегатная функция может быть определена так:

```
CREATE AGGREGATE sum (complex)
(
    sfunc = complex_add,
    stype = complex,
    initcond = '(0,0)'
);
```

Использовать её можно будет так:

```
SELECT sum(a) FROM test_complex;

   sum
-----------
 (34,53.9)
```

(Заметьте, что мы задействуем перегрузку функций: в системе есть несколько агрегатных функций с именем sum, но PostgreSQL может определить, какая именно из них применима к столбцу типа complex.)

Определённая выше функция sum вернёт ноль (начальное значение состояния), если в наборе данных не окажется значений, отличных от NULL. У нас может возникнуть желание вернуть NULL в этом случае — стандарт SQL требует, чтобы sum работала так. Мы можем добиться этого, просто опустив фразу initcond, так что начальным значением состояния будет NULL. Обычно это будет означать, что в sfunc придётся проверять входное значение состояния на NULL. Но для sum и некоторых других простых агрегатных функций, как max и min, достаточно вставить в переменную состояния первое входное значение не NULL, а затем начать применять функцию перехода со следующего значения не NULL. PostgreSQL сделает это автоматически, если начальное значение состояние равно NULL и функция перехода помечена как «strict» (то есть не должна вызываться для аргументов NULL).

Ещё одна особенность поведения по умолчанию «строгой» функции перехода — предыдущее значение состояния остаётся без изменений, когда встречается значение NULL. Другими словами, значения NULL игнорируются. Если вам нужно другое поведение для входных значений NULL, не объявляйте свою функцию перехода строгой (strict); вместо этого, проверьте в ней поступающие значения на NULL и обработайте их, как требуется.

Функция avg (вычисляющая среднее арифметическое) представляет собой более сложный пример агрегатной функции. Ей необходимы два компонента текущего состояния: сумма входных значений и их количество. Окончательный результат получается как частное этих величин. При реализации этой функции для значения состояния обычно используется массив. Например, встроенная реализация avg(float8) выглядит так:

```
CREATE AGGREGATE avg (float8)
(
    sfunc = float8_accum,
    stype = float8[],
    finalfunc = float8_avg,
    initcond = '{0,0,0}'
);
```

Примечание

Функция `float8_accum` принимает массив из трёх, а не двух элементов, так как в дополнение к количеству и сумме значений она подсчитывает ещё сумму их квадратов. Это сделано для того, чтобы её можно было применять для `avg` и для некоторых других агрегатных функций.

Вызовы агрегатных функций SQL допускают указания `DISTINCT` и `ORDER BY`, которые определяют, какие строки и в каком порядке будут поступать в функцию перехода агрегата. Это реализовано на заднем плане и непосредственно не затрагивает функции, поддерживающие работу агрегата.

CREATE AGGREGATE

CREATE AGGREGATE — создать агрегатную функцию

Синтаксис

```
CREATE AGGREGATE имя ( [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [ ,
... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
)

CREATE AGGREGATE имя ( [ [ режим_аргумента ] [ имя_аргумента ] тип_данных_аргумента [
, ... ] ]
                        ORDER BY [ режим_аргумента ] [ имя_аргумента ]
тип_данных_аргумента [ , ... ] ) (
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , INITCOND = начальное_условие ]
    [ , PARALLEL = { SAFE | RESTRICTED | UNSAFE } ]
    [ , HYPOTHETICAL ]
)

или старый синтаксис

CREATE AGGREGATE имя (
    BASETYPE = базовый_тип,
```

```
    SFUNC = функция_состояния,
    STYPE = тип_данных_состояния
    [ , SSPACE = размер_данных_состояния ]
    [ , FINALFUNC = функция_завершения ]
    [ , FINALFUNC_EXTRA ]
    [ , COMBINEFUNC = комбинирующая_функция ]
    [ , SERIALFUNC = функция_сериализации ]
    [ , DESERIALFUNC = функция_десериализации ]
    [ , INITCOND = начальное_условие ]
    [ , MSFUNC = функция_состояния_движ ]
    [ , MINVFUNC = обратная_функция_движ ]
    [ , MSTYPE = тип_данных_состояния_движ ]
    [ , MSSPACE = размер_данных_состояния_движ ]
    [ , MFINALFUNC = функция_завершения_движ ]
    [ , MFINALFUNC_EXTRA ]
    [ , MINITCOND = начальное_условие_движ ]
    [ , SORTOP = оператор_сортировки ]
)
```

Описание

`CREATE AGGREGATE` создаёт новую агрегатную функцию. Некоторое количество базовых и часто используемых агрегатных функций включено в дистрибутив. Но если нужно адаптировать их к новым типам или создать недостающие агрегатные функции, это можно сделать с помощью команды `CREATE AGGREGATE`.

Если указывается имя схемы (например, `CREATE AGGREGATE myschema.myagg ...`), агрегатная функция создаётся в указанной схеме. В противном случае она создаётся в текущей схеме.

Агрегатная функция идентифицируется по имени и типам входных данных. Две агрегатных функции в одной схеме могут иметь одно имя, только если они работают с разными типами данных. Имя и тип(ы) входных данных агрегата не могут совпадать с именем и типами данных любой другой обычной функции в той же схеме. Это же правило действует при перегрузке имён обычных функций (см. CREATE FUNCTION).

Простую агрегатную функцию образуют одна или две обычные функции: функция перехода состояния *функция_состояния* и необязательная функция окончательного вычисления *функция_завершения*. Они используются следующим образом:

```
функция_состояния( внутреннее-состояние, следующие-значения-данных ) ---> следующее-
внутреннее-состояние
функция_завершения( внутреннее-состояние ) ---> агрегатное_значение
```

PostgreSQL создаёт временную переменную типа *тип_данных_состояния* для хранения текущего внутреннего состояния агрегата. Затем для каждой поступающей строки вычисляются значения аргументов агрегата и вызывается функция перехода состояния с текущим значением состояния и полученными аргументами; эта функция вычисляет следующее внутреннее состояние. Когда таким образом будут обработаны все строки, вызывается завершающая функция, которая должна вычислить возвращаемое значение агрегата. Если функция завершения отсутствует, просто возвращается конечное значение состояния.

Агрегатная функция может определить начальное условие, то есть начальное значение для внутренней переменной состояния. Это значение задаётся и сохраняется в базе данных в виде

строки типа `text`, но оно должно быть допустимым внешним представлением константы типа данных переменной состояния. По умолчанию начальным значением состояния считается NULL.

Если функция перехода состояния объявлена как «strict» (строгая), её нельзя вызывать с входными значениями NULL. В этом случае агрегатная функция выполняется следующим образом. Строки со значениями NULL игнорируются (функция перехода не вызывается и предыдущее значение состояния не меняется) и если начальное состояние равно NULL, то в первой же строке, в которой все входные значения не NULL, первый аргумент заменяет значение состояния, а функция перехода вызывается для каждой последующей строки, в которой все входные значения не NULL. Это поведение удобно для реализации таких агрегатных функций, как `max`. Заметьте, что такое поведение возможно, только если *`тип_данных_состояния`* совпадает с первым *`типом_данных_аргумента`*. Если же эти типы различаются, необходимо задать начальное условие не NULL или использовать нестрогую функцию перехода состояния.

Если функция перехода состояния не является строгой, она вызывается безусловно для каждой поступающей строки и должна сама обрабатывать вводимые значения и переменную состояния, равные NULL. Это позволяет разработчику агрегатной функции полностью управлять тем, как она воспринимает значения NULL.

Если функция завершения объявлена как «strict» (строгая), она не будет вызвана при конечном значении состояния, равном NULL; вместо этого автоматически возвращается результат NULL. (Разумеется, это вполне нормальное поведение для строгих функций.) Когда функция завершения вызывается, она в любом случае может возвратить значение NULL. Например, функция завершения для `avg` возвращает NULL, если определяет, что было обработано ноль строк.

Иногда бывает полезно объявить функцию завершения как принимающую не только состояние, но и дополнительные параметры, соответствующие входным данным агрегата. В основном это имеет смысл для полиморфных функций завершения, которым может быть недостаточно знать тип данных только переменной состояния, чтобы вывести тип результата. Эти дополнительные параметры всегда передаются как NULL (так что функция завершения не должна быть строгой, когда применяется `FINALFUNC_EXTRA`), но в остальном это обычные параметры. Функция завершения может выяснить фактические типы аргументов в текущем вызове, воспользовавшись системным вызовом `get_fn_expr_argtype`.

Агрегатная функция может дополнительно поддерживать *режим движущегося агрегата*. Для этого режима требуются параметры `MSFUNC`, `MINVFUNC` и `MSTYPE`, а также могут задаваться `MSSPACE`, `MFINALFUNC`, `MFINALFUNC_EXTRA` и `MINITCOND`. За исключением `MINVFUNC`, эти параметры работают как соответствующие параметры простого агрегата без начальной буквы `M`; они определяют отдельную реализацию агрегата, включающую функцию обратного перехода.

Если в список параметров добавлено указание `ORDER BY`, создаётся особый типа агрегата, называемый *сортирующим агрегатом*; с указанием `HYPOTHETICAL` создаётся *гипотезирующий агрегат*. Эти агрегаты работают с группами отсортированных значений и зависят от порядка сортировки, поэтому определение порядка сортировки входных данных является неотъемлемой частью их вызова. Кроме того, они могут иметь *непосредственные* аргументы, которые вычисляются единожды для всей процедуры агрегирования, а не для каждой поступающей строки. Гипотезирующие агрегаты представляют собой подкласс сортирующих агрегатов, в которых непосредственные аргументы должны совпадать, по количеству и типам данных, с агрегируемыми аргументами. Это позволяет добавить значения этих непосредственных аргументов в набор агрегируемых строк в качестве дополнительной «гипотетической» строки.

Агрегатная функция может дополнительно поддерживать *частичное агрегирование*. Для этого требуется задать параметр COMBINEFUNC. Если в качестве *типа_данных_состояния* выбран internal, обычно уместно также задать SERIALFUNC и DESERIALFUNC, чтобы было возможно параллельное агрегирование. Заметьте, что для параллельного агрегирования агрегатная функция также должна быть помечена как PARALLEL SAFE (безопасная для распараллеливания).

2. Выполнение лабораторной работы

1. Выполнить вставку тестовых данных в таблицы, созданные в ходе выполнения лабораторной работы 9.

Сделать запрос выборки с условием к таблицам предку и потомку.

Придумать и создать пользовательский оператор для своей предметной области

Придумать и создать пользовательскую агрегатную функцию для своей предметной области

3. Содержание отчета

— Текст задания.

— физическую модель БД.

— наборы данных, содержащихся в таблицах БД

— текст запросов на SQL;

— код операторов

— пример выполнения операторов

— наборы данных, возвращаемые запросами.

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 11 Разработка документной базы данных

1. Теоретическая часть

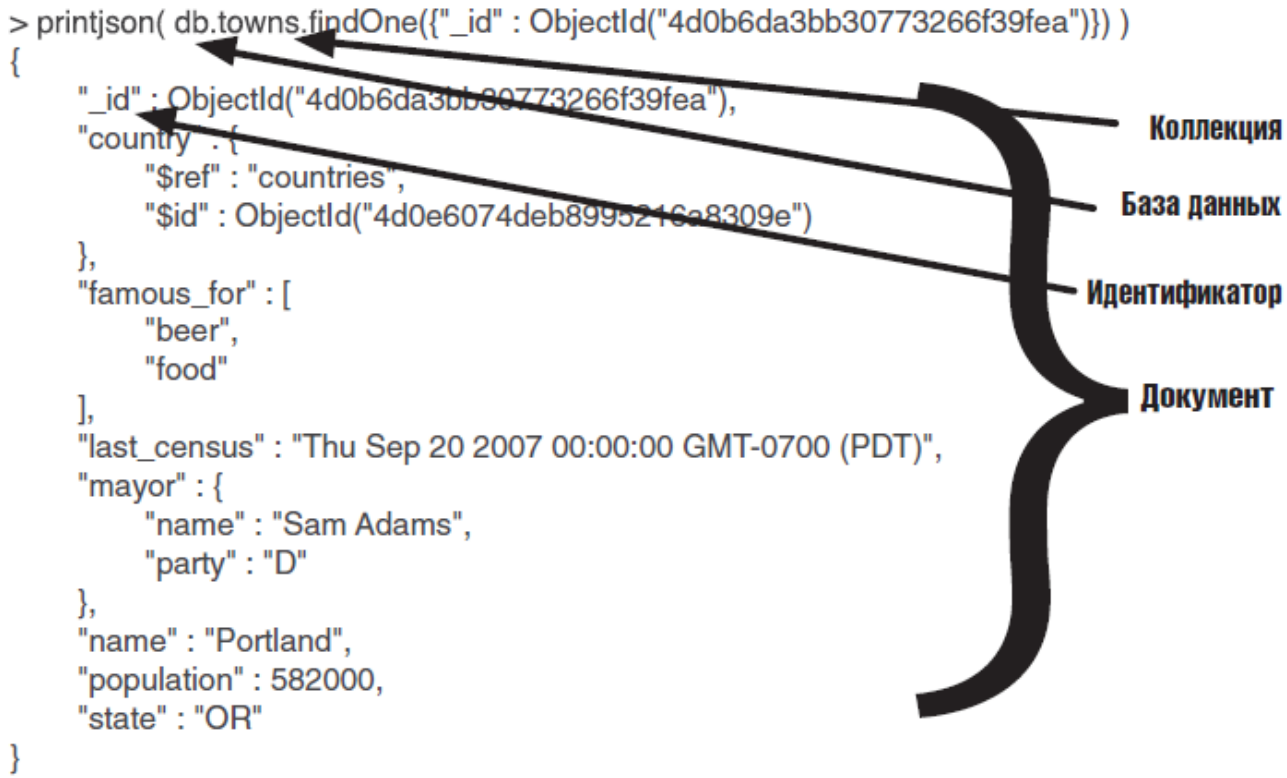
Создание БД

MongoDB реализует новый подход к построению баз данных, где нет таблиц, схем, запросов SQL, внешних ключей и многих других вещей, которые присущи объектно-реляционным базам данных.

В отличие от реляционных баз данных MongoDB предлагает документо-ориентированную модель данных, благодаря чему MongoDB работает быстрее, обладает лучшей масштабируемостью, ее легче использовать.

Mongo – хранилище JSON-документов (хотя, строго говоря, данные хранятся в двоичном варианте JSON, который называется BSON).

Документ Mongo можно уподобить строке реляционной таблицы без схемы, в которой допускается произвольная глубина вложенности значений. Рисунок ниже поможет понять, как выглядит JSON-документ.



Документы вместо строк

Если реляционные базы данных хранят строки, то MongoDB хранит документы. В отличие от строк документы могут хранить сложную по структуре информацию. Документ можно представить как хранилище ключей и значений.

Ключ представляет простую метку, с которым ассоциировано определенный кусок данных.

Однако при всех различиях есть одна особенность, которая сближает MongoDB и реляционные базы данных. В реляционных СУБД встречается такое понятие как первичный ключ. Это понятие описывает некий столбец, который имеет уникальные значения. В MongoDB для каждого документа имеется уникальный идентификатор, который называется `_id`. И если явным образом не указать его значение, то MongoDB автоматически сгенерирует для него значение.

Каждому ключу сопоставляется определенное значение. Но здесь также надо учитывать одну особенность: если в реляционных базах есть четко очерченная структура, где есть поля, и если какое-то поле не имеет значение, ему (в зависимости от настроек конкретной бд) можно присвоить значение `NULL`. В MongoDB все иначе. Если какому-то ключу не сопоставлено значение, то этот ключ просто опускается в документе и не употребляется.

Коллекции

Если в традиционном мире SQL есть таблицы, то в мире MongoDB есть коллекции. И если в реляционных БД таблицы хранят однотипные жестко структурированные объекты, то в коллекции могут содержать самые разные объекты, имеющие различную структуру и различный набор свойств.

Репликация

Система хранения данных в MongoDB представляет набор реплик. В этом наборе есть основной узел, а также может быть набор вторичных узлов. Все вторичные узлы сохраняют целостность и

автоматически обновляются вместе с обновлением главного узла. И если основной узел по каким-то причинам выходит из строя, то один из вторичных узлов становится главным.

Простота в использовании

Отсутствие жесткой схемы базы данных и в связи с этим потребности при малейшем изменении концепции хранения данных пересоздавать эту схему значительно облегчают работу с базами данных MongoDB и дальнейшим их масштабированием. Кроме того, экономится время разработчиков. Им больше не надо думать о пересоздании базы данных и тратить время на построение сложных запросов.

GridFS

Одной из проблем при работе с любыми системами баз данных является сохранение данных большого размера. Можно сохранять данные в файлах, используя различные языки программирования. Некоторые СУБД предлагают специальные типы данных для хранения бинарных данных в БД (например, BLOB в MySQL).

В отличие от реляционных СУБД MongoDB позволяет сохранять различные документы с различным набором данных, однако при этом размер документа ограничивается 16 мб. Но MongoDB предлагает решение - специальную технологию **GridFS**, которая позволяет хранить данные по размеру больше, чем 16 мб.

Система GridFS состоит из двух коллекций. В первой коллекции, которая называется `files`, хранятся имена файлов, а также их метаданные, например, размер. А в другой коллекции, которая называется `chunks`, в виде небольших сегментов хранятся данные файлов, обычно сегментами по 256 кб.

Для тестирования GridFS можно использовать специальную утилиту **mongofiles**, которая идет в пакете mongodb.

Создание базы

Чтобы создать новую базу данных book, выполните в окне терминала

показанную ниже команду. Она запускает интерактивный интерфейс,

устроенный по образцу MySQL.

```
$ mongo book
```

Для начала наберите слово help. Сейчас текущей является база данных book, но команда show dbs покажет другие базы, а команда use позволит сменить текущую базу. Для создания коллекции в Mongo достаточно просто добавить в нее первую запись. Поскольку в Mongo нет схем, то заранее ничего определять не надо. Более того, даже сама база данных book физически не существует, пока мы не добавим в нее первый документ. Следующий код создает коллекцию towns и вставляет в нее данные:

```
> db.towns.insert({
name: “New York”,
population: 22200000,
last_census: ISODate(“2009-07-31”),
```

```
famous_for: [ “statue of liberty”, “food” ],
mayor : {
name : “Michael Bloomberg”,
party : “I”
}
})
```

Выше мы упоминали, что документы хранятся в формате JSON (точнее, BSON), поэтому в таком формате мы их и добавляем. Фигурные скобки {...} обозначают объект (аналог ассоциативного массива или хеш-таблицы), содержащий ключи и значения, а квадратные скобки [...] – линейный массив. Значения могут быть вложенными, причем глубина вложенности не ограничена. Команда show collections позволит убедиться, что коллекция действительно существует.

```
> show collections
system.indexes
towns
```

Коллекция towns создана нами, а коллекция system.indexes существует всегда. Просмотреть содержимое коллекции позволяет команда fi nd(). Для удобства чтения мы отформатировали вывод, поскольку обычно выводится сплошная строка.

```
> db.towns.find()
{
“_id” : ObjectId(“4d0ad975bb30773266f39fe3”),
“name” : “New York”,
“population”: 22200000,
“last_census”: “Fri Jul 31 2009 00:00:00 GMT-0700 (PDT)”,
“famous_for” : [ “statue of liberty”, “food” ],
“mayor” : { “name” : “Michael Bloomberg”, “party” : “I” }
        }
```

JavaScript

Родным языком Mongo является JavaScript , который применяется и в сложных MapReduce-запросах, и для простейшего получения справки:

```
> db.help()
> db.towns.help()
```

Эти команды выводят список доступных для данного объекта функций. db представляет собой JavaScript-объект, который содержит информацию о текущей базе данных. db.x – JavaScript-объект, представляющий коллекцию с именем x. Сами команды – обычные JavaScript-функции.

```
> typeof db
object
> typeof db.towns
        object

> typeof db.towns.insert
function
```

Чтобы ознакомиться с исходным кодом функции, вызовите ее без параметров и без скобок Давайте добавим еще несколько документов в коллекцию towns, для чего напишем собственную JavaScript-функцию.

mongo/insert_city.js

```
function insertCity(
name, population, last_census,
famous_for, mayor_info
) {
```

```
db.towns.insert({
name:name,
population:population,
last_census: ISODate(last_census),
famous_for:famous_for,
mayor : mayor_info
});
}
```

Можете просто скопировать этот код в оболочку, а затем вызвать его.

```
insertCity(“Punxsutawney”, 6200, ‘2008-31-01’,
[“phil the groundhog”], { name : “Jim Wehrle” }
)
```

2. Выполнение лабораторной работы

Несмотря на то что считается что NoSQL базы данных не имеет схемы на самом деле схема есть Просто она написано неявно или не описано вообще тем не менее когда планируется использовать базу данных необходимо продумать формат представления данных, каким бы он ни был и задокументировать его. для хранения данных в базе данных mongodb используется формат JSON , точнее его бинарное представление BSON.

Необходимо спроектировать структуру json файла , соответствующую предметной области по варианту задания .

3. Содержание отчета

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Лабораторная работа № 12 Манипулирование данными в документной базе данных

1. Теоретическая часть

Обновление данных

Метод save

Как и другие системы управления базами данных MongoDB предоставляет возможность обновления данных. Наиболее простым для использования является метод `save`. В качестве параметра этот метод принимает документ.

В этот документ в качестве поля можно передать параметр `_id`. Если метод находит документ с таким значением _id, то документ обновляется. Если же с подобным _id нет документов, то документ вставляется.

Если параметр _id не указан, то документ вставляется, а параметр _id генерируется автоматически как при обычном добавлении через функцию insert:

```
> db.users.save({name: "Eugene", age : 29, languages: ["english", "german", "spanish"]})
```

В качестве результата функция возвращает объект WriteResult. Например, при успешном сохранении мы получим:

```
WriteResult({"nInserted" : 1 })
```

update

Более детальную настройку при обновлении предлагает функция update. Она принимает три параметра:

- `query`: принимает запрос на выборку документа, который надо обновить
- `objNew`: представляет документ с новой информацией, который заместит старый при обновлении
- `options`: определяет дополнительные параметры при обновлении документов. Может принимать два аргумента: `upsert` и `multi`.

  Если параметр `upsert` имеет значение `true`, что mongodb будет обновлять документ, если он найден, и создавать новый, если такого документа нет. Если же он имеет значение `false`, то mongodb не будет создавать новый документ, если запрос на выборку не найдет ни одного документа.

Параметр `multi` указывает, должен ли обновляться первый элемент в выборке (используется по умолчанию, если данный параметр не указан) или же должны обновляться все документы в выборке.

Например:

```
> db.users.update({name : "Tom"}, {name: "Tom", age : 25}, {upsert:
true})
```

Теперь документ, найденный запросом `{name : "Tom"}`, будет перезаписан документом `{"name": "Tom", "age" :"25"}`.

Функция `update()` также возвращает объект WriteResult. Например:

```
WriteResult({"nMatched" : 1, "nUpserted": 0,
"nModified": 1})
```

В данном случае результат говорит нам о том, что найден один документ, удовлетворяющий условию, и один документ был обновлен.

Обновление отдельного поля

Часто не требуется обновлять весь документ, а только значение одного из его ключей. Для этого применяется оператор $set. Если документ не содержит обновляемое поле, то оно создается.

```
> db.users.update({name : "Tom", age: 29}, {$set: {age :
30}})
```

Если обновляемого поля в документе нет, до оно добавляется:

```
> db.users.update({name : "Tom", age: 29}, {$set: {salary :
300}})
```

В данном случае обновлялся только один документ, первый в выборке. Указав значение `multi:true`, мы можем обновить все документы выборки:

```
> db.users.update({name : "Tom"}, {$set: {name: "Tom", age : 25}},
{multi:true})
```

Для простого увеличения значения числового поля на определенное количество единиц применяется оператор $inc. Если документ не содержит обновляемое поле, то оно создается. Данный оператор применим только к числовым значениям.

```
> db.users.update({name : "Tom"}, {$inc:
{age:2}})
```

Удаление поля

Для удаления отдельного ключа используется оператор $unset:

```
> db.users.update({name : "Tom"}, {$unset: {salary:
1}})
```

Если вдруг подобного ключа в документе не существует, то оператор не оказывает никакого влияния. Также можно удалять сразу несколько полей:

```
> db.users.update({name : "Tom"}, {$unset: {salary: 1, age:
1}})
```

updateOne и updateMany

Метод updateOne похож на метод update за тем исключением, что он обновляет только один документ.

```
> db.users.updateOne({name : "Tom", age: 29}, {$set: {salary :
360}})
```

Если необходимо обновить все документы, соответствующие некоторому критерию, то применяется метод updateMany():

```
> db.users.updateMany({name : "Tom"}, {$set: {salary :
560}})
```

Обновление массивов

Оператор $push

Оператор $push позволяет добавить еще одно значение к уже существующему. Например, если ключ в качестве значения хранит массив:

```
> db.users.updateOne({name : "Tom"}, {$push: {languages:
"russian"}})
```

Если ключ, для которого мы хотим добавить значение, не представляет массив, то мы получим ошибку `Cannot apply $push/$pushAll modifier to non-array`.

Используя оператор `$each`, можно добавить сразу несколько значений:

```
> db.users.update({name : "Tom"}, {$push: {languages: {$each: ["russian",
"spanish", "italian"]}}})
```

Еще пара операторов позволяет настроить вставку. Оператор $position задает позицию в массиве для вставки элементов, а оператор $slice указывает, сколько элементов оставить в массиве после вставки.

```
> db.users.update({name : "Tom"}, {$push: {languages: {$each: ["german", "spanish", "italian"], $position:1, $slice:5}}})
```

В данном случае элементы `["german", "spanish", "italian"]` будут вставляться в массив languages с 1-го индекса, и после вставки, в массиве останутся только 5 первых элементов.

Оператор $addToSet

Оператор $addToSet подобно оператору $push добавляет объекты в массив. Отличие состоит в том, что $addToSet добавляет данные, если их еще нет в массиве:

```
> db.users.update({name : "Tom"}, {$addToSet: {languages: "russian"}})
```

Удаление элемента из массива

Оператор `$pop` позволяет удалять элемент из массива:

```
> db.users.update({name : "Tom"}, {$pop: {languages: 1}})
```

Указывая для ключа languages значение 1, мы удаляем первый элемент с конца. Чтобы удалить первый элемент сначала массива, надо передать отрицательное значение:

```
> db.users.update({name : "Tom"}, {$pop: {languages: -1}})
```

Несколько иное действие предполагает оператор `$pull`. Он удаляет каждое вхождение элемента в массив. Например, через оператор $push мы можем добавить одно и то же значение в массив несколько раз. И теперь с помощью $pull удалим его:

```
> db.users.update({name : "Tom"}, {$pull: {languages: "english"}})
```

А если мы хотим удалить не одно значение, а сразу несколько, тогда мы можем применить оператор `$pullAll`:

```
> db.users.update({name : "Tom"}, {$pullAll: {languages: ["english", "german", "french"]}})
```

Удаление данных

-

-
-
-

Для удаления документов в MongoDB предусмотрен метод `remove`:

```
> db.users.remove({name :
"Tom"})
```

Метод `remove()` возвращает объект WriteResult. При успешном удалении одного документа результат будет следующим:

```
WriteResult({"nRemoved"
: 1})
```

В итоге все найденные документы с name=Tom будут удалены. Причем, как и в случае с find, мы можем задавать условия выборки для удаления различными способами (в виде регулярных выражений, в виде условных конструкций и т.д.):

```
> db.users.remove({name :
/T\w+/i})
> db.users.remove({age: {$lt :
30}})
```

Метод `remove` также может принимать второй необязательный параметр булевого типа, который указывает, надо удалять один элемент или все элементы, соответствующие условию. Если этот параметр равен `true`, то удаляется только один элемент. По умолчанию он равен false:

```
> db.users.remove({name : "Tom"},
true)
```

Чтобы удалить разом все документы из коллекции, надо оставить пустым параметр запроса:

```
>
db.users.remove({})
```

Удаление коллекций и баз данных

Мы можем удалять не только документы, но и коллекции и базы данных. Для удаления коллекций используется функция drop:

```
>
db.users.drop()
```

И если удаление коллекции пройдет успешно, то консоль выведет:

```
true
```

Чтобы удалить всю базу данных, надо воспользоваться функцией dropDatabase():

```
> db.dropDatabase()
```

Выборка из БД

Наиболее простой способом получения содержимого БД представляет использование функции `find`. Действие этой функции во многом аналогично обычному запросу `SELECT * FROM Table`, который извлекает все строки. Например, чтобы извлечь все документы из коллекции users, созданной в прошлой теме, мы можем использовать команду `db.users.find()`.

Для вывода документов в более удобном наглядном представлении мы можем добавить вызов метода pretty():

```
> db.users.find().pretty()
```

Однако что, если нам надо получить не все документы, а только те, которые удовлетворяют определенному требованию. Например, мы ранее в базу добавили следующие документы:

```
> db.users.insertOne({"name": "Tom", "age": 28, languages: ["english", "spanish"]})
> db.users.insertOne({"name": "Bill", "age": 32, languages: ["english", "french"]})
> db.users.insertOne({"name": "Tom", "age": 32, languages: ["english", "german"]})
```

Выведем все документы, в которых name=Tom:

```
> db.users.find({name: "Tom"})
```

Такой запрос выведет нам два документа, в которых name=Tom.

Теперь более сложный запрос: нам надо вывести те объекты, у которых name=Tom и одновременно age=32. То есть на языке SQL это могло бы выглядеть так: `SELECT * FROM Table WHERE Name='Tom' AND Age=32`. Данному критерию у нас соответствует последний добавленный объект. Тогда мы можем написать следующий запрос:

```
> db.users.find({name: "Tom", age: 32})
```

Также несложно отыскать по элементу в массиве. Например, следующий запрос выводит все документы, у которых в массиве languages есть english:

```
> db.users.find({languages:
"english"})
```

Усложним запрос и получим те документы, у которых в массиве languages одновременно два языка: "english" и "german":

```
> db.users.find({languages: ["english",
"german"]})
```

Теперь выведем все документы, в которых "english" в массиве languages находится на первом месте:

```
> db.users.find({"languages.0":
"english"})
```

Соответственно если нам надо вывести документы, где english на втором месте (например, `["german", "english"]`), то вместо нуля ставим единицу: `languages.1`.

Проекция

Документ может иметь множество полей, но не все эти поля нам могут быть нужны и важны при запросе. И в этом случае мы можем включить в выборку только нужные поля, использовав проекцию. Например, выведем только порцию информации, например, значения полей "age" у все документов, в которых name=Tom:

```
> db.users.find({name: "Tom"},
{age: 1})
```

Использование единицы в качестве параметра `{age: 1}` указывает, что запрос должен вернуть только содержание свойства age.

И обратная ситуация: мы хотим найти все поля документа кроме свойства age. В этом случае в качестве параметра указываем 0:

```
> db.persons.find({name: "Tom"},
{age: 0})
```

При этом надо учитывать, что даже если мы отметим, что мы хотим получить только поле name, поле _id также будет включено в результирующую выборку. Поэтому, если мы не хотим видеть данное поле в выборке, то надо явным образом указать: `{"_id":0}`

Альтернативно вместо 1 и 0 можно использовать true и false:

```
> db.users.find({name: "Tom"}, {age: true, _id:
false})
```

Если мы не хотим при этом конкретизировать выборку, а хотим вывести все документы, то можно оставить первые фигурные скобки пустыми:

```
> db.users.find({}, {age: 1,
_id: 0})
```

Запрос к вложенным объектам

Предыдущие запросы применялись к простым объектам. Но документы могут быть очень сложными по структуре. Например, добавим в коллекцию persons следующий документ:

```
> db.users.insert({"name": "Alex", "age": 28, company: {"name":"microsoft",
"country":"USA"}})
```

Здесь определяется вложенный объект с ключом company. И чтобы найти все документы, у которых в ключе company вложенное свойство name=microsoft, нам надо использовать оператор точку:

```
> db.users.find({"company.name":
"microsoft"})
```

Использование JavaScript

MongoDB предоставляет замечательную возможность, создавать запросы, используя язык JavaScript. Например, создадим запрос, возвращающий те документы, в которых name=Tom. Для этого сначала объявляется функция:

```
> fn = function() { return
this.name=="Tom"; }
> db.users.find(fn)
```

Этот запрос эквивалентен следующему:

```
>
db.users.find("this.name=='Tom'")
```

Собственно только запросами область применения JavaScript в консоли mongo не ограничена. Например, мы можем создать какую-нибудь функцию и применять ее:

```
> function sqrt(n) { return
n*n; }
> sqrt(5)
25
```

Использование регулярных выражений

Еще одной замечательной возможностью при построении запросов является использование регулярных выражений. Например, найдем все документы, в которых значение ключа name начинается с буквы T:

```
> db.users.find({name:/T\w+/i})
```

Настройка запросов и сортировка

MongoDB представляет ряд функций, которые помогают управлять выборкой из бд. Одна из них - функция limit. Она задает максимально допустимое количество получаемых документов. Количество передается в виде числового параметра. Например, ограничим выборку тремя документами:

```
> db.users.find().limit(3)
```

В данном случае мы получим первые три документа (если в коллекции 3 и больше документов). Но что, если мы хотим произвести выборку не сначала, а пропустив какое-то количество документов? В этом нам поможет функция skip. Например, пропустим первые три записи:

```
> db.users.find().skip(3)
```

MongoDB предоствляет возможности отсортировать полученный из бд набор данных с помощью функции sort. Передавая в эту функцию значения 1 или -1, мы можем указать в каком порядке сортировать: по возрастанию (1) или по убыванию (-1). Во многом эта функция по действию аналогична выражению `ORDER BY` в SQL. Например, сортировка по возрастанию по полю `name`:

```
> db.users.find().sort({name: 1})
```

Ну и в конце надо отметить, что мы можем совмещать все эти функции в одной цепочке:

```
> db.users.find().sort({name: 1}).skip(3).limit(3)
```

Поиск одиночного документа

Если все документы извлекаются функцией `find`, то одиночный документ извлекается функцией `findOne`. Ее действие аналогично тому, как если бы мы использовали функцию `limit(1)`, которая также извлекает первый документ коллекции. А комбинация функций skip и limit извлечет документ по нужному местоположению.

Параметр $natural

Если вдруг нам надо отсортировать ограниченную коллекцию, то мы можем воспользоваться параметром `$natural`. Этот параметр позволяет задать сортировку: документы передаются в том порядке, в каком они были добавлены в коллекцию, либо в обратном порядке.

Например, отберем последние пять документов:

```
> db.users.find().sort({ $natural: -1
}).limit(5)
```

Оператор $slice

$slice является в некотором роде комбинацией функций `limit` и `skip`. Но в отличие от них $slice может работать с массивами.

Оператор `$slice` принимает два параметра. Первый параметр указывает на общее количество возвращаемых документов. Второй параметр необязательный, но если он используется, тогда первый параметр указывает на смещение относительно начала (как функция skip), а второй - на ограничение количества извлекаемых документов.

Например, в каждом документе определен массив languages для хранения языков, на которых говорит человек. Их может быть и 1, и 2, и 3 и более. И допустим, ранее мы добавили следующий объект:

```
> db.users.insert({"name": "Tom", "age": "32", languages: ["english",
"german"]})
```

И мы хотим при выводе документов сделать так, чтобы в выборку попадал только один язык из массива languages, а не весь массив:

```
> db.users.find ({name: "Tom"}, {languages: {$slice :
1}})
```

Данный запрос при извлечении документа оставит в результате только первый язык из массива languages, то есть в данном случае english.

Обратная ситуация: нам надо оставить в массиве также один элемент, но не с начала, а с конца. В этом случае необходимо передать в параметр отрицательное значение:

```
> db.users.find ({name: "Tom"}, {languages: {$slice : -
1}});
```

Теперь в массиве окажется german, так как он первый с конца в добавленном элементе.

Используем сразу два параметра:

```
> db.users.find ({name: "Tom"}, {languages: {$slice : [-1,
1]}});
```

Первый параметр говорит начать выборку элементов с конца (так как отрицательное значение), а второй параметр указывает на количество возвращаемых элементов массива. В итоге в массиве language окажется "german"

Курсоры

Результат выборки, получаемой с помощью функции `find`, называется курсором:

```
> var cursor = db.users.find();
null;
```

Чтобы получить курсор и сразу же не выводить все содержащиеся в нем данные, после метода `find()` добавляет через точку с запятой выражение `null;`

Курсоры инкапсулируют в себе наборы получаемых из бд объектов. Используя синтаксис языка javascript и методы курсоров, мы можем вывести полученные документы на экран и как-то их обработать. Например:

```
> var cursor =
db.users.find();null;
> while(cursor.hasNext()){
... obj = cursor.next();
... print(obj["name"]);
... }
```

Курсор обладает методом hasNext, который показывает при переборе, имеется ли еще в наборе документ. А метод next извлекает текущий документ и перемещает курсор к следующему документу в наборе. В итоге в переменной obj оказывается документ, к полям которого мы можем получить доступ.

Также для перебора документов в курсоре в качестве альтернативы мы можем использовать конструкцию итератора javascript - forEach:

```
> var cursor =
db.users.find()
>
cursor.forEach(function(obj){
... print(obj.name);
... })
```

Чтобы ограничить размер выборки, используется метод `limit`, принимающий количество документов:

```
> var cursor =
db.users.find();null;
null
> cursor.limit(5);null;
null
>
cursor.forEach(function(obj){
... print(obj.name);
... })
```

Используя метод `sort()`, можно отсортировать документы в курсоре:

```
> var cursor =
db.users.find();null;
null
> cursor.sort({name:1});null;
null
>
cursor.forEach(function(obj){
... print(obj.name);
... })
```

Выражение `cursor.sort({name:1})` сортирует документы в курсоре по полю name по возрастанию. Если мы хотим отсортировать по убыванию, то вместо 1 используем -1: `cursor.sort({name:-1})`

И еще один метод `skip()` позволяет пропустить при выборке определенное количество документов:

```
> var cursor =
db.users.find();null;
null
> cursor.skip(2);null;
null
>
cursor.forEach(function(obj){
... print(obj.name);
... })
```

В данном случае пропускаем два документа.

И кроме того, можно объединять все эти методы в цепочки:

```
> var cursor = db.users.find();null;
null
>
```

```
cursor.sort({name:1}).limit(3).skip(2);null;
     null
     > cursor.forEach(function(obj){
     ... print(obj.name);
     ... })
```

2. Выполнение лабораторной работы

Привести пример обновления и удаления данных из базы

Выполнить запросы на выборку по варианту задания

3. Содержание отчета

— Текст задания.

—модель БД.

— текст запросов;

— наборы данных, возвращаемые запросами.

4. Варианты заданий

Варианты заданий приведены в Приложении 3.

Приложение 1 Распределение баллов

Семестр 5

№	Наименование лабораторной	Количество баллов	Предельный № недели сдачи
1.	Разработка физической модели базы данных с учетом декларативной ссылочной целостности	4	2
2.	Создание и модификация базы данных и таблиц базы данных	3	4
3.	Заполнение таблиц и модификация данных	3	6
4.	Разработка SQL запросов: виды соединений и шаблоны	5	8
5.	Разработка SQL запросов: запросы с подзапросами	6	10
6.	Хранимые процедуры	7	13
7.	Триггеры. Обеспечение активной целостности данных базы данных	7	15
8.	Проектирование взаимодействия базы данных и приложения	5	17
	Итого	40	

Семестр 6

№	Наименование лабораторной	Количество баллов	Предельный № недели сдачи
9.	Объектно-реляционные базы данных. Проектирование и создание	10	4
10.	Объектно-реляционные базы данных. Манипуляция данными и пользовательские операторы	15	8
11.	Разработка документной базы данных	10	12
12.	Манипулирование данными в документной базе данных	15	16
	Итого	50	

Приложение 2 Варианты заданий лабораторных 1-8

1. Программа для рисования графов: название вершины, координаты левой верхней точки отображаемой вершины, и её размеры, автор графа

   а. Вершины, название/текст которых содержит слово «отказ», но не зканчивается им
   б. Вершины, у которых нет исходящих ребер
   в. Графы, в которых есть пара вершин, связанных ребрами в обе стороны
   г. Ширина графа в пикселях (от максимальной сумма координаты по горизонтали с шириной отнять минимальную левую координату)
   д. пользователи-авторы графов с максимальных количеством вершин
   е. Вершины, для которых есть исходящие ребра, ведущие ко всем остальным вершинам её графа
   ж. Вершина, у которой нет входящих ребер от вершины о словом «ошибка»

2. Садоводство: участки, владельцы с учетом совместной собственности, линии/номер участка, площадь стоимость постройки, тип построек ,взносы в фонд садоводства

   а. номера участков владельцев с отчеством, заканчивающимся на «ич»
   б. участки, на которых зарегистрировано более 1 типа постройки
   в. тип взносов, которые пока никто не оплатил
   г. Владелец (владельцы) участка максимальной площади
   д. Владельцы участков с максимальным числом типов построек
   е. Владельцы, оплатившие все типы взносов
   ж. Участки, на которых нет бань, но есть туалеты

3. ТСЖ: квартира, владелец, количество комнат, площадь, жилая площадь, этаж, показания счетчиков

   а. квартиры владельцев, отчества которых заканчиваются на «на»
   б. владельцев, у которых есть квартиры с разным количеством комнат
   в. квартиры, в которых не записано ни одного показания счетчиков
   г. владельцы квартир с максимальной жилой площадью
   д. этаж, на котором меньше всего квартир находится в собственности
   е. этаж квартиры, в которых стоят все типы счетчиков
   ж. владелец, у которого нет двухкомнатных квартир, но есть трехкомнатные

4. парк: деревья ,породы, дата высадки, дата обрезки, расположение, аллеи

   а. аллеи, на которых встречаются дубы (дуб в названии породы дерева)
   б. деревья, стоящие на перекрёстке аллей (1 дерево на 2 аллеях)
   в. породы, не высаженные в парке
   г. дерево, которое было посажено позже всех
   д. порода, деревьев которой меньше всего
   е. порода дерева, встречающаяся на всех аллеях заданного парка
   ж. аллея, на которой растут липы, но нет клёнов

5. расписание экзаменов/зачетов: даты экзаменов, дисциплина, преподаватель, группа, аудитория

   а. аудитории, которых проходят экзамены по дисциплинам, начинающимся со слова «интеллект», но это не единственное слово названия
   б. дата, когда в одной аудитории проходит несколько экзаменов
   в. дисциплины, по которым не бывает экзаменов
   г. Преподаватели, принимающие самые последние экзамены
   д. группа, у которой меньше всего экзаменов
   е. дисциплина, по которой есть экзамены у всех групп
   ж. аудитория, в которой не проходят экзамены у группы Z8431

6. поликлиника: прием, пациент, процедура/прием, врач, стоимость

   а. пациенты, приходившие на любые процедуры, связанные с лазером (лазер в любом месте названия)
   б. врач, проводивший одному пациенту и прием и процедуру
   в. процедуры, которые никому не делали
   г. процедуры с наименьшей стоимостью, которые проводили Иванову Ивану Ивановичу
   д. пациент, ходивший на наибольшее количество процедур
   е. врач, проводивший все процедуры
   ж. врач, не принимавший к Иванова Ивана Ивановича, но проводивший ему процедуру.

7. личный кабинет: дисциплина, работа, тип работы, студент, преподаватель, дата сдачи , баллы

   а. преподаватели, за которыми закреплены дисциплины, начинающиеся со слова «систем»
   б. дисциплина, по которой есть несколько типов работ
   в. задание, по которому не прикреплено ни одной работы
   г. студент, сдавший курсовую раньше всех
   д. группа, прикрепившая в этом месяце наибольшее число работ
   е. дисциплина, по которой есть все типы работ (КР, ЛР, практические)
   ж. преподаватель, которому не прикрепляли отчетов по лабораторным работам

8. костюмерная театра: роль, спектакль, название костюма, деталь костюма, размер, автор модели, дата разработки

   а. спектакли, в которых используются костюмы, имеющие в названии слово «принц»
   б. костюм, в котором есть и плащ и штаны
   в. спектакль, на который пока нет костюмов
   г. роль, к которой разрабатывался самый старый из костюмов
   д. автор, разработавший наибольшее число костюмов

е. костюм, в котором есть все типы деталей
ж. автор, не разрабатывавший костюмы к «Золушке», но разрабатывавший к «Мастеру и Маргарите»

9. охраняемые парковки: адрес парковки, машина, владелец, место, рег. номер машины, дата и время заезда, дата и время выезда

а. все парковки, расположенные на проспектах
б. владелец машины, у которого несколько машин разных марок
в. машины, которые не парковались на парковках
г. владелец машин, заезжавший позже всех
д. владелец машины, останавливавшийся на минимальном числе парковок
е. машина, которая стояла на всех парковках Московского района
ж. владелец, не парковавшийся на Невском проспекте, но парковавшийся на Московском

10. спортзал: абонементы, виды спорта, тренеры, дата покупки абонемента, тип абонемента

а. виды спорта, на которые есть абонементы со словом «супер» в названии типа)
б. тренер, который ведет занятия по нескольким различным видам спорта
в. вид спорта, по которому не куплено ни одного абонемента
г. тренеры, ведущие занятия по виду спорта с максимальным числом занятий в абонементе
д. вид спорта, по которому занятия ведет максимальное количество тренеров
е. дата, когда не покупали абонементы на йогу
ж. совокупный абонемент, на все виды латинских танцев.

11. магазин обоев: ширина рулона, название, коллекция, производитель, ширина, материал, цена, текстура материала, партия, дата поставки партии.

Покупая обои, необходимо обратить внимание на то, чтобы обои были одной партии (на этикетках рулонов указана одинаковая комбинация символов). Это гарантия того, что партия обоев изготовлена в один день, из одной краски и рулоны будут одинакового оттенка. Подробности про обои смотри: https://interior-in.ru/steny/oboi-dlya-sten/132-uslovnye-oboznacheniya-na-oboyakh-chto-oni-oznachayut.html
https://dekoriko.ru/nastennye-pokrytiya/oboi/pod-pokrasku/

а. коллекция обоев, начинающаяся со слова «Лес», но это не единственное слово
б. производитель, который производит обои не только из бумаги
в. цвет, обоев которого нет
г. коллекция с самой большой шириной рулона
д. производитель обоев с минимальным количеством коллекций
е. производитель, который производит обои из всех материалов
ж. производитель, у которого нет обоев дешевле 3000 рублей

12. расписание пригородных электричек: Режим движения электрички (ежедневно, выходные, кроме сб и вс), маршрут электрички (перечень всех станций в маршруте), время отправления с каждой станции, время прибытия на каждую станцию

а. все маршруты, проходящие через остановку , в названии которой есть слово километр (21 километр, 36 километр)
б. свежепостроенная станция, к которой пока не ведет ни один маршрут
в. маршрут, проходящий через Шушары и Павловск
г. самая ранняя электричка, прибывающая на Московский вокзал
д. маршруты с наибольшим количеством станций
е. маршрут, на котором нет электричек с режимом отправления по будням, ноесть по выходным.
ж. маршрут, по которому ходят электрички со всеми режимами движения

13. вакансии: название вакансии, организация работодатель, адрес работодателя, диапазон зарплаты, требования к образованию, Обязанности, график работы, требования обязательные, желательные, дата выставления вакансии

а. вакансии, имеющие в названии «SQL», но не заканчивающиеся на него
б. требование, не присутствующее ни в одной вакансии
в. работодатели в Новгороде, выставившие вакансии и программиста JavaScript и системного администратора
г. вакансия с наибольшей зарплатой
д. вакансии с максимальным количеством необязательных требований
е. требование, присутствующее во всех вакансиях на программиста (любые вакансии со словом программист)
ж. вакансии, которые есть в Санкт-Петербурге, но которых нет в Новгороде

14. калькулятор бюджета физического лица: категория дохода (продажа, зарплата), категория расхода (еда, счета за КУ, здоровье …), статьи дохода и расхода, дата расхода/дохода. Категория- более общее понятие чем статья. Например категория- еда, а статьи в ней мясо, рыба, вкусное к чаю, а конкретный расход «печенье «курабье» к чаю 11.09»

а. все расходы категорий, которые относятся к медицине (содержат слово «мед», но не являются мёдом)
б. месяц, в котором были статьи дохода от работы и от аренды (сдачи в аренду)
в. категория доходов, по которой нет статей
г. категория, по которой были наибольшие расходы в прошлом году
д. категория, по которой не было расходов в январе, но были в феврале
е. категория расхода, по которой траты были во всех месяцах 2021 года
ж. месяц, в котором были расходы максимального количества статей

15. Книга контактов: люди, метки (друзья, коллеги, семья), дни рождения, телефон, электронная почта, примечания к человеку или контакту

а. друзья, у которых номер телефона начинается на 8-911
б. Люди (контакты) без электронной почты
в. люди, которые относятся к друзьям и коллегам одновременно
г. самые молодые люди среди контактов
д. месяц, когда есть дни рождения у коллег, но нет у семьи

е. люди, к которым относится максимальное меток
ж. месяц, в котором есть дни рождения у людей со всеми метками

16. вузы для абитуриента: город, вуз, факультеты, направления, направленности, ЕГЭ которые нужно сдать, дата начала/конца приемной кампании.
(Направление -09.03.04 «Программная инженерия», Направленность — его конкретизация «Разработка программно-информационных систем», именно направленность закреплена за кафедрой и соответственно факультетом)

а. направленности, в которых есть слово «интеллект», но оно не последнее
б. факультет, не принимающий ни на одну направленность
в. направление, на которое надо сдавать математику и физику
г. вуз, принимающий на наибольшее количество направлений
д. город, в котором есть все направления
е. вуз, с самым поздним концом приемной кампании
ж. направление, на которое не надо сдавать ЕГЭ по информатике, но надо по физике

17. школьные экскурсии: тип (развлекательная/образовательная), дисциплины к которым имеет отношение образовательная экскурсия, стоимость с человека, список участников, ответственный за проведение учитель

а. экскурсии на заводы (слово «завод» в любом месте названия)
б. экскурсии, относящиеся к биологии и истории
в. учащиеся, которые не ездили за экскурсии
г. учащиеся, которые не ездили в музей истории религии, но ездил в Кунсткамеру
д. учитель, отвечавший за на наибольшее число экскурсий
е. самые дешевые экскурсии
ж. учащийся, который был на всех экскурсиях

18. собачий питомник: собака, владелец, порода, родители, медали с выставок, дата рождения

а. породы, названия которых содержат слово «шпиц», но не заканчивается на него
б. собаки без детей
в. собаки, у родителей которых один и тот же владелец
г. владелец, у которого есть собаки всех пород на букву «к»
д. владелец, у которого есть овчарки, но нет ирландских волкодавов
е. собака, имеющая меньше всего медалей
ж. владельцы самых молодых собак

19. служба доставки: адрес доставки, контактное лицо, стоимость посылки, диапазон желаемого времени доставки. время доставки фактическое, вес посылки, отметка о доставке, фирма отправитель

а. все посылки, отправляемые на улицу, в названии которой есть окончание «на», но это не единственные буквы в нем
б. улица, на которую никогда не доставляли посылки
в. контактное лицо, получавшее посылки на улице и на проспекте
г. фирма, отправившая меньше всего посылок
д. посылка с самым большим весом
е. контактное лицо, получавшее посылки от всех фирм
ж. контактное лицо, никогда не получавшее посылок в марте, но получавшее в январе

20. туристический путеводитель: город, достопримечательность, адрес, тип достопримечательности (памятник, архитектурный комплекс, природный комплекс), дата создания

а. достопримечательности, в которых есть слово «Кремль», но с него название не начинается
б. город без достопримечательностей
в. улица, на которой есть и памятники, и архитектурные комплексы
г. город, в котором нет памятников, но есть музеи
д. улица, на которой больше всего музеев
е. улица с самыми молодыми достопримечательностями
ж. улица, на которой есть достопримечательности всех типов на букву «п»

21. метрополитен: линия, станция, время закрытия, время открытия, адрес выходов, время проезда между станциями, дата открытия станции

а. станции, в названии которых есть окончание «ая», но оно на него не заканчивается
б. станция без выходов
в. пересадочные станции с линии 2 на линию 4
г. линия с самым большим временем проезда
д. линия, на которой собраны все стации, на букву «а»
е. станция с самым поздним закрытием
ж. линия на которой нет перегона меньше 2 минут

22. Медпункт организации: сотрудник организации, данные флюорографии (результат, дата), данные прививок, жалобы сотрудников (прием в медпункте)

а. все сотрудники, мужского пола (отчество, оканчивается на «ич»)
б. сотрудники, никогда не делавшие флюорографию
в. сотрудники организации, делавшие за последний год прививки от столбняка и ковид-19
г. сотрудник, последним сделавший флюорографию в прошлом году
д. сотрудник, у которого сделаны все типы прививок
е. сотрудник с самым большим количеством жалоб
ж. сотрудник, не делавший прививки от кори, но делавший от энцефалита

23. Инвентаризация: ответственное лицо, оборудование , категории имущества (компьютерная техника, канцелярские товары, мебель, освещение, спец. оборудование и т.д.), помещения, дата поставки, дата списания

а. любые помещения, где стоит специализированные столы (оборудование, содержащее «слово», но не начинается с него.
б. Мат. ответственное лицо, за которым пока не числится никакого имущества
в. помещение, в котором есть и компьютерная техника и специальное оборудование
г. имущество, списанное первым в этом году
д. категория оборудования, которая стоит во всех помещениях
е. мат. ответственное лицо, ответственное за наименьшее количество имущества
ж. мат ответственное лицо, ответственное за мебель, но не за компьютерную технику

24. Календарь корпоративных мероприятий: время дата, событие, статус события для участника (обязательное, рекомендуемое, нейтральное), должности и подразделения участников, комнаты проведения

а. Все мероприятия, текущего года, в названии которых есть слово «сессия», но на него с названия не заканчивается
б. Событие, которому пока не назначили участников
в. Комната, в которой проходят 2 мероприятия в один день
г. Участники самых рано заканчивающихся мероприятий
д. Событие, с самым большим количеством не пришедших, но приглашенных на него участников.(участники на мероприятие могут быть приглашены и при этом могут придти/не придти).
е. Мероприятие, в котором обязательно присутствие всех инженеров
ж. Сотрудник, не посещавший мероприятий в июле прошлого года, но посещавший в июле этого.

25. Распределение обязанностей: обязанность, название, частота выполнения, категория обязанности (домашняя, оплата счетов, посещение мероприятий, общественная)

а. Люди, выполняющие обязанности, имеющие в названии слово «ремонт», но не начинающиеся с него
б. Человек без обязанностей
в. Человек, у которого есть обязанности с различной частотой выполнения (например ежедневные и ежемесячные)
г. Ежемесячная обязанность, которая выполняется первой по дате (дню)
д. Категория, обязанности из которой выполняют все, кто вообще имеет обязанности
е. Категория, в которой меньше всего обязанностей
ж. Человек, который не выполняет обязанности из категории «оплата счетов», но выполняет из категории «уборка»

26. Отслеживание задач: задачи, проекты, автор и исполнитель задачи, сроки выполнения()

а. Проекты, в названии которых содержится слово «интеграция», но оно не первое
б. Задача без исполнителей
в. Исполнитель, работающий на 2 различных проекта
г. Задача с самым ранним сроком окончания
д. Проект с самым маленьким количеством задач
е. Человек, ставящий задачи всех приоритетов
ж. Человек, у которого нет незавершенных задач, но есть незавершенные проекты

Приложение 3 Варианты заданий лабораторных 9-12

1. Программа для рисования векторной графики: автор изображения (пользователь), объекты, примитивы, координаты левого верхней точки отображаемой объекта, и её размеры

а. Элемент графики, название/текст которых содержит слово «конец»
б. Элементы изображения, связанные друг с другом
в. Ширина изображения в пикселях (от максимальной сумма координаты по горизонтали с шириной отнять минимальную левую координату)
г. Пользователь- автор, изображений с максимальных количеством элементов
д. Вершина, нет смежных элементов-окружностей

2. Садоводческое товарищество: объекты общего пользования (колонки, помойка, магазин) участки, владельцы с учетом совместной собственности, линии/номер участка, площадь стоимость постройки, тип построек ,взносы в фонд садоводства

а. номера участков владельцев с отчеством, заканчивающимся на «вна»
б. участки в том числе общего пользования, на которых зарегистрировано более1 постройки
в. Владелец (владельцы) участка максимальной площади
г. Владельцы максимального количества участков
д. Участки, на которых нет бань

3. Управляющая компания: квартиры, коммерческие помещения, владелец, количество комнат, площадь, жилая площадь, этаж, показания счетчиков

а. квартиры владельцев, отчества которых заканчиваются на ‘ич’
б. владельцев, у которых есть квартиры на разных этажах
в. владельцы квартир с минимальной жилой площадью
г. этаж, на котором меньше всего квартир находится в собственности
д. владелец, у которого нет трехкомнатных квартир

4. парк: статуи, фонтаны, деревья ,породы, дата высадки, дата обрезки, расположение, аллеи

а. аллеи, на которых встречаются разные виды кленов (клен в названии)
б. аллеи, состоящие из разных пород деревьев
в. дерево, которое было посажено раньше всех
г. порода, деревьев которой меньше всего
д. аллея, деревья на которой не высаживались в прошлом году

5. расписание экзаменов и зачетов: , даты зачетов(диапазон- неделя),даты экзаменов, дисциплина, преподаватель, группа, аудитория

   а. аудитории, которых проходят экзамены по дисциплинам, имеющим в названии слова «базы данных»
   б. аудитории, где в один день проходит несколько экзаменов
   в. дисциплины самого последнего экзамена
   г. аудитории, в которых проходит больше всего экзаменов
   д. преподаватель, не принимающий экзамены у группы 4831

6. детская поликлиника а: разграничен прием здоровых и больных детей датами (днями недели),прием, пациент, процедура/прием, врач, стоимость

   а. пациенты, приходившие на любые процедуры, связанные с электрофорезом
   б. пациент, приходивший к одному врачу и на прием и на процедуру
   в. процедуры с наименьшей стоимостью
   г. пациент, ходивший к наибольшему количеству врачей
   д. пациент, не ходивший на процедуры к Иванову Ивану Ивановичу

7. личный кабинет студента: практики, вкр, дисциплина, работа, тип работы, студент, преподаватель, дата сдачи , баллы

   а. преподаватели, аз которыми закреплены дисциплины, начинающиеся со слова «автоматизирован»
   б. дисциплина, по которой есть и лабораторные и курсовая работа
   в. студент, сдавший курсовую раньше всех
   г. студент, прикрепивший в этом месяце наибольшее число работ
   д. преподаватель, которому не прикрепляли отчетов по курсовому проекту или работе

8. оборудование театра: роль, спектакль, реквизит (для роли /как часть костюма), название костюма, деталь костюма, размер, автор модели, дата разработки (учесть, чтобы в спектаклях идущих одновременно не использовались одинаковые объекты)

   а. спектакли, в которых используются костюмы, имеющие в названии слово шут
   б. костюм, в котором есть и куртка и штаны
   в. автор, разработавший самый старый из костюмов
   г. спектаклю, к которому разработано наибольшее число костюмов
   д. автор, не разрабатывавший костюмы к «Золушке»

9. охраняемые парковки: адрес парковки, машина, тип машины (грузовая, легковая, с прицепом и т.д.), , и места под разные типы машин, под ремонт,владелец, место, рег. номер машины, дата и время заезда, дата и время выезда

а. все парковки, расположенные на линиях (не улицах или проспектах)(улица в адресе содержит «линия»)
б. владелец машины, у которого более одного места под машину
в. владелец машин, заезжавший раньше всех
г. владелец машины, останавливавшийся на минимальном числе парковок
д. владелец, не парковавшийся на Невском проспекте

10. спортзал: абонементы, свободное посещение тренажеров/залов, виды спорта, тренеры, дата покупки абонемента, тип абонемента

а. виды спорта, на которые есть безлимитные абонементы (со словом безлимит в названии типа)
б. вид спорта, по которому занятия ведут различные тренеры
в. тренеры, ведущие занятия по виду спорта с минимальным число занятий в абонементе
г. тренеры, которые ведут занятия по максимальному количеству видов спорта
д. тренеры, не ведущие фитнес

11. магазин обоев и покрытий для стен: типы покрытия (обои, живой камень, краска),ширина рулона, название, коллекция, производитель, ширина, материал, цена, текстура материала, партия, дата поставки партии

а. коллекция обоев, начинающаяся со слова Элегия
б. производитель, который производит как бумажные, так и флизелиновые обои
в. производитель с самой большой шириной рулона
г. производитель обоев с максимальным количеством коллекций
д. производитель, у которого нет обоев дороже 3000 рублей

12. расписание пригородных электричек и поездов дальнего следования: время отправления поезда, маршрут поезда,Режим движения электрички (ежедневно, выходные, кроме сб и вс), маршрут электрички (перечень всех станций в маршруте), время отправления с каждой станции, время прибытия на каждую станцию

а. все маршруты, проходящие через остановку , в названии которой есть слово село (царское село, детское село)
б. маршрут, проходящий через Купчино и Шушары
в. самая поздняя электричка, прибывающая на витебский вокзал
г. маршруты с наибольшим количеством электричек
д. маршрут, на котором нет электричек с режимом отправления по выходным.

13. вакансии: волонтерские позиции,название вакансии, организация работодатель, адрес работодателя, диапазон зарплаты, требования к образованию, Обязанности, график работы, требования обязательные, желательные, дата выставления вакансии

а. вакансии, имеющие в названии SQL
б. работодатели в Санкт-Петербурге, выставившие вакансии программиста и системного администратора
в. вакансия с наименьшей зарплатой

г. вакансии с минимальным количеством обязательных требований
д. вакансии, в которых нет требования к опыту работы

14. калькулятор бюджета физического лица: пользователь калькулятора (разные уровни доступа), категория дохода (продажа, зарплата), категория расхода (еда, счета за КУ, здоровье …), статьи дохода и расхода, дата расхода/дохода

а. все расходы категорий, которые относятся к спорту (содержат слово спорт)
б. месяц, в котором были разные статьи дохода
в. категория, по которой наибольшие расходы в текущем году
г. категория, по которой не было расходов в январе
д. месяц, в котором были траты максимального количества статей

15. Книга контактов: организации с их контактами и временем работы, люди, метки (друзья, коллеги, семья), дни рождения, телефон, электронная почта, примечания к человеку или контакту

а. друзья, у которых номер телефона начинается на 8-911,
б. люди, которые относятся к соседям и коллегам одновременно
в. самые старые люди среди контактов
г. месяц, когда есть дни рождения у соседей, но нет у семьи
д. метки, к которым относится максимальное количество людей

16. вузы и колледжи для абитуриента: учебные заведения , разделенные по уровням образования, город, вуз, факультеты, направления, направленности, ЕГЭ которые нужно сдать, дата начала приемной кампании.
(Направление -09.03.04 «Программная инженерия», Направленность — его конкретизация «Разработка программно-информационных систем», именно направленность закреплена за кафедрой и соответственно факультетом)

а. направленности, в которых есть слово автоматизированный
б. направление, на которое надо сдавать историю и обществознание
в. факультет вуза, принимающий на наименьшее количество направлений
г. вуз, с самым ранним началом приемной компании
д. направление, на которое не надо сдавать ЕГЭ по информатике

17. школьные экскурсии: тип (развлекательная/образовательная), дисциплины к которым имеет отношение образовательная экскурсия, стоимость с человека, список участников, ответственный за проведение учитель, категории участников экскурсии (участвует, организует, ассистирует при проведении (старшие классы у младших или родители))

а. экскурсии в музеи (слово музей в любом месте названия)
б. экскурсии, относящиеся к литературе и истории
в. учащиеся, которые не ездили в музей истории религии
г. экскурсия, собравшая наибольшее число участников
д. самые дорогие экскурсии

18. питомник собак и кошек: животное, владелец, порода, родители, медали с выставок, дата рождения

а. породы, названия которых содержат слово «шпиц», но не начинается с него
б. собаки, у родителей которых один и тот же владелец
в. владелец, у которого есть йоркширские терьеры, но нет мастиффов
г. порода, собак которой меньше всего
д. владельцы самых старых собак

19. служба доставки с учетом разницы физических и юридических лиц: адрес доставки, контактное лицо, стоимость посылки, диапазон желаемого времени доставки. время доставки фактическое, вес посылки, отметка о доставке, фирма отправитель (у юр. лица есть контактное лицо (человек))

а. все посылки, отправляемые на улицу, в названии которой есть окончание «ая», но это не единственные буквы в нем
б. улица, на которую никогда не доставляли посылки
в. контактное лицо, получавшее посылки по разным адресам
г. посылка с самым маленьким весом
д. контактное лицо, никогда не получавшее посылок в январе

20. туристический путеводитель с учетом не только «адресных» достопримечательностей, но и природных: местность, город, достопримечательность, адрес, тип достопримечательности (памятник, архитектурный комплекс, природный комплекс), дата создания

а. достопримечательности, в которых есть слово «мать», но с него название не начинается
б. улица, на которой есть и памятники, и архитектурные комплексы
в. город, в котором нет природных комплексов
г. улица, на которой больше всего памятников
д. города с самыми старыми достопримечательностями

21. метрополитен и трамвай: линия, станция, время закрытия, время открытия, адрес выходов, время проезда между станциями, дата открытия станции

а. станции, в названии которых есть слово «площадь», но оно на него не заканчивается
б. пересадочные станции с линии 1 на линию 2
в. линия с самым большим временем проезда
г. станция с самым ранним открытием
д. линия на которой нет перегона больше 3 минут

22. Медпункт вуза: студенты, сотрудники вуза, данные флюорографии (результат, дата), данные прививок, жалобы сотрудников (прием в медпункте)

а. все сотрудники, женского пола (отчество, оканчивается на «на»)

б. сотрудники организации, делавшие за последний год прививки от столбняка и кори
в. сотрудник, последним сделавший флюорографию
г. сотрудник с самым большим количеством жалоб
д. сотрудник, не делавший прививки от энцефалита

23. Инвентаризация: ответственное лицо, оборудование , категории имущества (компьютерная техника, канцелярские товары (могут не иметь помещения, а приписаны быть к отделу, не списываются), мебель, освещение, спец. оборудование и т.д.), помещения, дата поставки, дата списания

а. любые помещения, где стоит компьютерная мебель (категория мебель и слово «комп»)
б. материально ответственное лицо, у которого подотчетна и компьютерная техника и специальное оборудование
в. имущество, списанное последним
г. мат. ответственное лицо, ответственное за наибольшее количество имущества
д. помещение, в котором нет компьютерной техники

24. Календарь корпоративных мероприятий: время дата, событие, статус события для участника (обязательное, рекомендуемое, нейтральное), должности и подразделения участников, комнаты проведения, отдельный учет сотрудника как организатора.

а. Все мероприятия, текущего года, в названии которых есть слово конференция, но с него с названия не начинаются

б. Мероприятия, в которых участвуют сотрудники «подразделения 1» и «подразделения 2»
в. Участники самых поздно заканчивающихся мероприятий
г. Событие, с самым большим количеством пришедших на него участников.(участники на мероприятие могут быть приглашены и при этом могут придти/не придти).
д. Сотрудник, не посещавший мероприятий в июле прошлого года

25. Распределение обязанностей: обязанность, название, частота выполнения, категория обязанности (домашняя, оплата счетов, посещение мероприятий, общественная), разные статусы людей (выполнение и назначение обязанностей)

а. Люди, выполняющие обязанности, имеющие в названии слово уборка, но не заканчивающиеся на него
б. Обязанность, которую выполняет более одного человека
в. Ежемесячная обязанность, которая выполняется первой по дате (дню)
г. Категория, в которой меньше всего обязанностей
д. Человек, который не выполняет обязанности из категории «оплата счетов»

26. Отслеживание задач: задачи, проекты, автор и исполнитель задачи, сроки выполнения(разные статусы людей (выполнение и назначение задач), некоторые не могут назначать)

а. Задачи, в названии которых содержится слово «интеграция», но оно не последнее
б. Задача, относящаяся к 2 различным проектам
в. Задача с самым поздним сроком окончания
г. Проект с самым большим количеством задач
д. Человек, у которого нет незавершенных задач