Содержание

Ограничения целостности.

Типы данных

1. Классы → Виды → Типы → Варианты.
2. Базовые типы данных.
    2.5.1. Тип GUID.
    2.5.2. Тип Counter.
3. Пользовательский тип данных.
4. Значения по умолчанию.
5. Виртуальные атрибуты.
6. Базовые и виртуальные отношения.

Отсутствующие данные.

1. Пустые значения (Empty-ззначения).
2. Неопределенные значения (Null-значения).
3. Null-значения и общее правило вычисления выражений.
4. Null-значения и логические операции.
5. Null-значения и проверка условий.

Данные в mySQL.

понятие о транзакции.
Типология архитектур доступа к записям и индексация таблиц.

Ограничения целостности.

Кратко. Ограничения целостности в базах данных, назначение, доменная целостность, сущностная целостность, ссылочная целостность, декларативная и процедурная целостность, перехват ошибок при нарушениях целостности.

Целостность (от англ. integrity – нетронутость, неприкосновенность, сохранность, целостность) – понимается как правильность данных в любой момент времени. Но эта цель может быть достигнута лишь в определенных пределах: СУБД не может контролировать правильность каждого отдельного значения, вводимого в базу данных (хотя каждое значение можно проверить на правдоподобность).

Например, нельзя обнаружить, что вводимое значение 5 (представляющее номер дня недели) в действительности должно быть равно 3. С другой стороны, значение 9 явно будет ошибочным и СУБД должна его отвергнуть. Однако для этого ей следует сообщить, что номера дней недели должны принадлежать набору (1,2,3,4,5,6,7).

Поддержание целостности базы данных может рассматриваться как защита данных от попадания в базу недопустимым данным (например, предупредить ошибки пользователей при вводе данных), от неверных изменений или разрушений (не путать с незаконными изменениями и разрушениями, являющимися проблемой безопасности).

Современные СУБД имеют ряд средств для обеспечения поддержания целостности (так же, как и средств обеспечения поддержания безопасности).

Что такое ограничения целостности в реляционной БД.

Все ограничения целостности можно разделить на несколько больших категорий:

1. Прикладная целостность (приложение).
2. Доменная целостность (атрибут-столбец).
3. Сущностная целостность (отношение-таблица).
4. Ссылочная целостность (связь-ключ).
5. Транзакционная целостность (база).

Кроме того ограничения можно классифицировать как:

- **Средства декларативного характера** создаются как составные части объектов при их определении в базе данных (например, условие на значение при определении таблицы в базе данных).
- **Средства процедурного характера** (триггеры и хранимые процедуры) реализуются как отдельные программные модули. В общем случае процедурные ограничения более функциональны, но менее экономны с точки зрения ресурсов и наоборот.

Наличие развитой системы ограничений целостности во многом определяет зрелость базы данных. Обычно проще сразу позаботиться о том, чтобы в базу данных не попадали неверные значения, чем потом их убирать из базы данных.

1. Прикладная целостность (приложение).

Это ограничения в клиентском приложении, разработчики должны позаботиться о том, чтобы ошибки, возникающие при нарушениях целостности, перехватывались клиентским приложением.

2. Доменная целостность (атрибут-столбец).

Они отвечают за то, чтобы в соответствующем поле базы данных были допустимые значения. Например, фамилия, как правило, должна состоять из букв, а почтовый индекс - из цифр.

В базах данных такая целостность обычно обеспечивается типом данных (доменом), условиями на значение, запретом пустых значений, триггерами и хранимыми процедурами, а также ключами.

Подробнее типы данных будут рассмотрены дальше.

3. Сущностная целостность (отношение-таблица).

Главная задача здесь - сделать так, чтобы данные об одной сущности не попали в базу данных два раза.

В базах данных такая целостность обеспечивается ограничением уникальности записи и первичным ключом.

Объект реального мира представляется в реляционной базе данных как кортеж некоторого отношения.

Требование целостности сущностей заключается в следующем: каждый кортеж любого отношения должен отличатся от любого другого кортежа этого отношения (т.е. любое отношение должно обладать первичным ключом).

Вполне очевидно, что если данное требование не соблюдается (т.е. кортежи в рамках одного отношения не уникальны), то в базе данных может хранится противоречивая информация об одном и том же объекте.

Поддержание целостности сущностей обеспечивается средствами системы управления базой данных (СУБД). Это осуществляется с помощью двух ограничений:

- при добавлении записей в таблицу проверяется уникальность их первичных ключей;
- не позволяется изменение значений атрибутов, входящих в первичный ключ.

4. Ссылочная целостность (связь-ключ).

Обеспечивается системой первичных и внешних ключей. Например, при помощи этих средств можно гарантировать, что у нас не будет заказов, оформленных на покупателей, которых нет в базе данных.

Сложные объекты реального мира представляются в реляционной базе данных в виде кортежей нескольких нормализованных отношений, связанных между собой. При этом:

- Связи между данными отношениями описываются в терминах функциональных зависимостей.
- Для отражения функциональных зависимостей между кортежами разных отношений используется дублирование первичного ключа одного отношения (родительского) в другое (дочернее).
- Атрибуты, представляющие собой копии ключей родительских отношений, называются внешними ключами.

Требование целостности по ссылкам состоит в следующем: для каждого значения внешнего ключа, появляющегося в дочернем отношении, в родительском отношении должен найтись кортеж с таким же значением первичного ключа.

Пусть, например, даны отношения ОТДЕЛ (N_ОТДЕЛА, ИМЯ_ОТДЕЛА) и СОТРУДНИК (N_СОТРУДНИКА, N_ОТДЕЛА, ИМЯ_СОТРУДНИКА), в которых хранятся сведения о работниках предприятия и подразделениях, где они работают.

Отношение ОТДЕЛ в данной паре является родительским, поэтому его первичный ключ *"N_отдела"* присутствует в дочернем отношении СОТРУДНИК.

Требование целостности по ссылкам означает здесь, что в таблице СОТРУДНИК не может присутствовать кортеж со значением атрибута *"N_отдела"*, которое не встречается в таблице ОТДЕЛ. Если такое значение в отношении ОТДЕЛ отсутствует, значение внешнего ключа в отношении СОТРУДНИК считается неопределенным.

Как правило, поддержание целостности ссылок также возлагается на систему управления базой данных. Например, она может не позволить пользователю добавить запись, содержащую внешний ключ с несуществующим (неопределенным) значением.

5. Транзакционная целостность (база).

Для обеспечения целостности в случае ограничений на базу данных, а не какие-либо отдельные операции, служит аппарат транзакций.

Понятие о транзакции будет рассмотрено далее.

Типы данных.

1. Классы → Виды → Типы → Варианты.

В СУБД по умолчанию ввели несколько наиболее распространенных типов данных, каждый из которых принадлежит какому-то из перечисленных ниже классов и видов данных.

Типы данных делятся на **два класса** базовые и пользовательские.

Виды данных:

1) числовые данные;
2) логические данные;
3) строковые данные;
4) дата-временные данные;
5) идентификационные данные.

Базовые типы данных – это любые типы данных, заданные в системах управления базами данных изначально, т. е. присутствующие там по умолчанию (в отличие от пользовательского типа данных).

Необходимо заметить, что все базовые типы данных могут иметь различные **варианты по диапазону** представления данных, например: вариантами четырехбайтового типа данных (integer) могут быть восьмибайтовые (bigint) и двухбайтовые (smallint) типы данных.

2. Базовые типы данных.

2.1. В **числовом** виде данных выделяют следующие типы:

1) Integer. Этим ключевым словом обычно обозначают целый тип данных;
2) Real, соответствующий вещественному типу данных;
3) Decimal (n, m). Это десятичный тип данных. Причем в обозначении n – это число, фиксирующее общее количество знаков числа, а m показывает, сколько символов из них стоит после десятичной точки;
4) Money или Currency, введен специально для удобного представления данных денежного типа данных.

2.2. В **логическом** виде данных обычно выделяют только один базовый тип, это Logical.

2.3. **Строковый** вид данных насчитывает четыре базовых типа (имеются в виду, разумеется, наиболее распространенные):

1) Bit (n). Это строки бит с фиксированной длиной n;
2) Varbit (n). Это тоже строки бит, но с переменной длиной, не превышающей n бит;
3) Char (n). Это строки символов с постоянной длиной n;
4) Varchar (n). Это строки символов, с переменной длиной, не превышающей n символов.

2.4. Вид данных **дата и время** включает в себя следующие базовые типы данных:

1) Date – тип данных даты;
2) Time – тип данных, выражающих время суток;
3) Date-time – тип данных, выражающий одновременно и дату, и время.

2.5. **Идентификационный** вид данных содержит в себе только один включенный по умолчанию в СУБД тип, и это GUID (глобальный уникальный идентификатор).

2.5.1. Тип GUID.

Этот тип предназначен для хранения шестнадцатибайтовых значений так называемого глобального уникального идентификатора. Все различные значения этого идентификатора генерируются автоматически при вызове специальной встроенной функции **NewId ()**. Это обозначение происходит от полного английского словосочетания New Identification, что в переводе буквально и означает «новое значение идентификатора». Каждое генерируемое на конкретном компьютере значение идентификатора уникально в пределах всех производимых компьютеров.

GUID-идентификатор используется, в частности, для организации репликации баз данных, т. е. при создании копий каких-то уже имеющихся баз данных.

Такие GUID-идентификаторы могут быть использованы и разработчиками баз данных наравне с другими базовыми типами.

2.5.2. Тип Counter.

Промежуточное положение между типом GUID и другими базовыми типами занимает еще один специальный базовый тип – тип **счетчика** .

Для обозначения данных этого типа используется специальное ключевое слово «счетчик». Значения типа Counter обязательно являются целочисленными.

Counter (x_0, Δx), параметр x_0 задает начальное значение, а Δ x – шаг приращения.

Работа с этим базовым типом данных включает в себя ряд очень интересных **особенностей**:

- Значения типа Counter не задаются, как мы привыкли при работе со всеми другими типами данных, они **генерируются по требованию**, почти как для значений типа глобального уникального идентификатора.
- Атрибут данных типа Counter может быть задан только при определении таблицы и только тогда! В программном коде этот тип использовать нельзя.
- СУБД не позволит объявить в одной таблице несколько счетчиков. При определении таблицы тип счетчика может быть задан исключительно для одного столбца в таблице.
- Значения данных типа счетчик генерируются автоматически при вставке строк. Причем эта генерация проводится без повторений, так что счетчик всегда будет уникально идентифицировать каждую строку.
- Изменить вручную значение счетчика СУБД никогда не позволит ради гарантирования его уникальности.

Эти особенности создает некоторые неудобства при работе с таблицами, содержащими данные типа счетчик. Если, например, данные в отношении, заданном таблицей, изменятся и их придется удалить или поменять местами, значения счетчика легко могут «спутать карты».

Приведем пример, иллюстрирующий подобную ситуацию.

Счетчик типа Counter (1, 1)	...
1	...
2	...
3	...
4	...

Пусть дана таблица в которую введены четыре строки. Счетчик каждой новой строке автоматически дал уникальный номер.→

Пусть теперь необходимо удалить вторую и четвертую строчки из таблицы, а потом добавить одну дополнительную строчку.

Эти операции приведут к следующему преобразованию таблицы →

Счетчик типа Counter (1, 1)	...
1	...
3	...
5	...

Таким образом, счетчик удалил вторую и четвертую строчки вместе с их уникальными именами, и не стал «переприсваивать» их новым строчкам, как можно было ожидать.

Обычно счетчик используется как суррогатный, т. е. искусственный ключ в таблице.

Интересно знать, что уникальных значений четырехбайтового счетчика при скорости генерации одно значение в секунду хватит более чем на 100 лет. Покажем, как это подсчитано:

За 1 год расходуется (365д * 24ч * 60м * 60с * 1 знач. в сек. = 2^{25}) значений счётчика.
Возможное количество значений составляет ($2^{4*8} = 2^{32}$) ==> ($2^{32} / 2^{25} = 2^7$ = 128 лет).

3. Пользовательский тип данных.

Пользовательский тип данных отличается от всех базовых типов тем, что он не был изначально вшит в систему управления базами данных, он не был описан как тип данных по умолчанию. Этот тип может создать для себя любой пользователь и программист баз данных в соответствии с собственными запросами и требованиями.

Таким образом, **пользовательский тип данных** – это подтип некоторого базового типа, т. е. это базовый тип с некоторыми ограничениями множества допустимых значений.

3.1. Создание пользовательского типа данных.

В записи на псевдокоде, пользовательский тип данных создается с помощью следующего стандартного оператора:

Create subtype *имя подтипа*
Type *имя базового типа*
As *ограничение подтипа* ;

Итак, в первой строчке мы должны указать имя нашего нового, пользовательского типа данных, во второй – какой из имеющихся базовых типов данных мы взяли за образец, создавая собственный, и, наконец, в третьей – те ограничения, которые нам необходимо добавить в уже имеющиеся ограничения множества значений базового типа данных – образца.

Ограничения подтипа записываются как условие, зависящее от имени определяемого подтипа.

3.2. Пример.

Чтобы лучше понять принцип действия оператора Create, рассмотрим следующий пример. Пусть нам необходимо создать свой специализированный тип данных, допустим, для работы на почте. Это будет тип для работы с данными вида чисел почтового индекса. От обычных десятичных шестизначных чисел наши числа будут отличаться тем, что они могут быть только положительными. Запишем оператор, для создания нужного нам подтипа:

Create subtype *Почтовый индекс*
Type *decimal* (4, 0)
As *Почтовый индекс* > 0.

Почему мы взяли именно decimal (4, 0)? Вспоминая обычный вид индекса, мы видим, что такие числа должны состоять из четырёх целых чисел от нуля до девяти.

В общем случае условие, накладываемое на базовый тип данных, может содержать логические связки **not, and, or** и вообще быть выражением любой произвольной сложности.

Определенные таким образом пользовательские подтипы данных могут беспрепятственно использоваться наряду с другими базовыми типами данных и в программном коде, и при определении типов данных в столбцах таблицы, т. е. базовые типы данных и пользовательские при работе с ними совершенно равноправны.

В визуальной среде разработки они появляются в списках допустимых типов вместе с другими базовыми типами данных.

3.3. Удаление пользовательского типа данных.

При составлении предметных баз данных без проектирования собственных (пользовательский) типов данных обойтись практически невозможно. Ведь по умолчанию в СУБД вшиты только самые общие типы данных, пригодные соответственно для решения самых общих задач.

Но, иногда требуется удалить созданный подтип, чтобы не загромождать и не усложнять код. Для этого в СУБД обычно встроен специальный оператор **drop**, что и означает «удалить».

Общий вид этот оператор удаления ненужных пользовательских типов:

Drop subtype *имя пользовательского типа* ;

Пользовательские типы данных, как правило, рекомендуется вводить для подтипов достаточно общего назначения.

4. Значения по умолчанию.

Системы управления базами данных могут иметь возможность создания любых произвольных значений по умолчанию или, как их еще называют, умолчаний. Эта операция в любой среде программирования имеет достаточно большой вес, ведь практически в любой задаче может возникнуть необходимость введения констант, неизменяемых значений по умолчанию.

Для создания умолчания в системах управления базами данных используется уже знакомая нам по прохождению пользовательского типа данных функция **Create**. Только в случае создания значения по умолчанию используется также дополнительное ключевое слово **default**, которое и означает «умолчание». Оператор создания в БД значения по умолчанию:

Create default *имя умолчания*
As *константное выражение* ;

Понятно, что на месте константного значения при применении этого оператора нужно написать значение или выражение, которое мы хотим сделать значением или выражением по умолчанию. И, разумеется, надо решить, под каким именем нам будет удобно его использовать в нашей базе данных, и записать это имя в первую строчку оператора.

Итак, что мы получили? Мы вывели, что умолчание представляет собой именованную константу, сохраняемую в базах данных, как и ее объект. В визуальной среде разработки умолчания появляются в списке выделенных значений по умолчанию.

Приведем пример создания умолчания. Пусть для правильной работы нашей базы данных необходимо, чтобы в ней функционировало значение со смыслом неограниченного срока действия чего-либо. Тогда нужно ввести в список значений этой базы данных значение по умолчанию, отвечающему данному требованию. Это может быть необходимо хотя бы потому, что каждый раз при встрече в тексте кода с этим довольно громоздким выражением будет крайне неудобно выписывать его заново. Именно поэтому воспользуемся означенным выше оператором Create для создания умолчания, со смыслом неограниченного срока действия чего-либо.

Create default *«срок не ограничен»*
As ‘9999-12-31 23: 59:59’

Здесь также был использован синтаксис языка Transact-SQL, согласно которому значения констант типа «дата – время» (в данном случае, ‘9999-12-31 23: 59:59’) записываются как строки символов определенного направления. Интерпретация строк символов как значений типа «дата – время» определяется контекстом использования этих строк. Например, в нашем конкретном случае, сначала в константной строчке записано предельное значение года, а потом времени.

Однако при всей своей полезности умолчания, как и пользовательский тип данных, иногда тоже могут требовать того, чтобы их удалили. В системы управления базами данных обычно есть специальный встроенный предикат, аналогичный оператору, удаляющему ненужный более пользовательский тип данных. Это предикат **Drop** и сам оператор выглядят следующим образом:

Drop default *имя умолчания* ;

5. Виртуальные атрибуты.

Все атрибуты в системах управления базами данных делятся (по абсолютной аналогии с отношениями) на базовые и виртуальные. Так называемые **базовые атрибуты** – это хранимые атрибуты, которые необходимо использовать не один раз, а следовательно, целесообразно сохранить. А, в свою очередь, **виртуальные атрибуты** – это не хранимые, а вычисляемые атрибуты. Что это значит? Это значит, что значения так называемых виртуальных атрибутов реально не хранятся, а вычисляются через базовые атрибуты на ходу посредством задаваемых формул. При этом домены вычисляемых виртуальных атрибутов определяются автоматически.

Приведем пример таблицы, задающей отношение, в которой два атрибута – обычные, базовые, а третий атрибут – виртуальный. Он будет вычисляться по специально введенной формуле.

...	Вес Кг	Цена Руб за Кг	Стоимость = Вес Кг * Цена Руб за Кг	...
...	2	10	20	...
...	3	20	60	...

Итак, мы видим, что атрибуты «Вес Кг» и «Цена Руб за Кг» – базовые атрибуты, потому что они имеют обыкновенные значения и хранятся в нашей базе данных. А вот атрибут «Стоимость» – виртуальный атрибут, потому что он задан формулой своего вычисления и реально в базе данных храниться не будет.

Интересно заметить, что в силу своей природы, виртуальные атрибуты не могут принимать значения по умолчанию, да и вообще, само понятие значения по умолчанию для виртуального атрибута лишено смысла, а следовательно, не применяется.

И еще необходимо знать, что, несмотря на то что домены виртуальных атрибутов определяются автоматически, тип вычисляемых значений иногда нужно заменить с имеющегося на какой-нибудь другой. Для этого в языке систем управления базами данных имеется специальный предикат Convert, с помощью которого и может быть переопределен тип вычисляемого выражения. Convert – это так называемая функция явного преобразования типов. Записывается она следующим образом:

Convert (*тип данных, выражение*);

Выражение, стоящее вторым аргументом функции Convert, посчитается и выведется в виде таких данных, тип которых указан первым аргументом функции.

Рассмотрим пример. Пусть нам необходимо посчитать значение выражения «2*2», но вывести это нужно не в виде целого числа «4», а строкой символов. Для выполнения этого задания запишем следующую функцию Convert:

Convert (Char (1), 2 * 2).

Таким образом, можно увидеть, что данная запись функции Convert в точности даст необходимый нам результат.

6. Базовые и виртуальные отношения.

Как мы уже знаем, базы данных – это как бы своеобразный контейнер, основное предназначение которого заключается в хранении данных, представленных в виде отношений.

Необходимо знать, что в зависимости от своей природы и структуры, отношения делятся на:

1) **базовые отношения;**

2) **виртуальные отношения.**

Отношения базового вида содержат только независимые данные и не могут быть выражены через какие-либо другие отношения баз данных.

В коммерческих системах управления базами данных базовые отношения обычно называются просто **таблицами** (Table) в отличие от **представлений** (View), соответствующих понятию виртуальных отношений.

В данном курсе мы будем довольно подробно рассматривать только базовые отношения, основные приемы и принципы работы с ними. Виртуальные отношения будут затронуты незначительно.

Отсутствующие данные.

В СУБД для определения отсутствующих данных описаны два вида значений: пустые (**Empty-значения**) и неопределенные (**Null-значения**).

В некоторой (преимущественно коммерческой) литературе на Null-значения иногда ссылаются как на пустые или нулевые значения, однако это неверно. Смысл пустого и неопределенного значения принципиально различается, поэтому необходимо внимательно следить за контекстом употребления того или иного термина.

1. Пустые значения (Empty-значения).

Пустое значение – это просто одно из множества возможных значений какого-то вполне определенного типа данных.

Перечислим наиболее «естественные», непосредственные **пустые значения** (т. е. пустые значения, которые мы могли бы выделить самостоятельно, не имея никакой дополнительной информации):

1) 0 (нуль) – нулевое значение является пустым для числовых типов данных;
2) false (неверно) – является пустым значением для логического типа данных;
3) B'' – пустая строка бит для строк переменной длины;
4) “” – пустая строка для строк символов переменной длины.

В приведенных выше случаях определить, пустое значение или нет, можно путем сравнивания имеющегося значения с константой пустого значения, определенной для каждого типа данных. Но системы управления базами данных в силу реализованных в них схем долговременного хранения данных могут работать только со строками постоянной длины. Из-за этого пустой строкой бит можно назвать строку двоичных нулей. Или строку, состоящую из пробелов или каких-либо других управляющих символов, – пустой строкой символов.

Вот несколько примеров пустых строк постоянной длины:

1) B'0';
2) B'000';
3) ' '.

Как же в этих случаях определить, является ли строка пустой?

В системах управления базами данных для проверки на пустоту применяется логическая функция, т. е. предикат **IsEmpty (<выражение>)** , что буквально означает «есть пустой». Этот предикат обычно встроен в систему управления базами данных и может применяться к выражению абсолютно любого типа. Если такого предиката в системах управления базами данных нет, то можно написать логическую функцию самим и включить ее в список объектов проектируемой базы данных.

Рассмотрим еще один пример, когда не так просто определить, пустое ли мы имеем значение. Данные типа «дата». Какое значение в этом типе считать пустым значением, если дата может варьироваться в диапазоне от 01.01.0100. до 31.12.9999.? Для этого в СУБД вводится

специальное обозначение для **константы пустой даты {…}** , если значения этого типа записывается: {ДД. ММ. ГГ} или {ГГ. ММ. ДД}. С этим значением и происходит сравнение при проверке значения на пустоту. Оно считается вполне определенным, «полноправным» значением выражения этого типа, причем наименьшим из возможных.

При работе с базами данных пустые значения часто используются как значения по умолчанию или применяются, если значения выражений отсутствуют.

2. Неопределенные значения (Null-значения).

Слово **Null** используется для обозначения **неопределенных значений** в базах данных.

Чтобы лучше понять, какие значения понимаются под неопределенными, рассмотрим таблицу, являющуюся фрагментом базы данных:

№	Фамилия	Год рождения	№ паспорта
1	Хайретдинов	1980	Null
2	Карамазов	2000	Null
3	Коваленко	Null	Null

Итак, **неопределенное значение** или **Null-значение** – это:

1) неизвестное, но обычное, т. е. применимое значение. Например, у господина Хайретдинова, который является номером один в нашей базе данных, несомненно,

имеются какие-то паспортные данные (как у человека 1980 г. рождения и гражданина страны), но они не известны, следовательно, не занесены в базу данных. Поэтому в соответствующую графу таблицы будет записано значение Null;
2) неприменимое значение. У господина Карамазова (№ 2 в нашей базе данных) просто не может быть никаких паспортных данных, потому что на момент создания этой базы данных или внесения в нее данных, он являлся ребенком;
3) значение любой ячейки таблицы, если мы не можем сказать применимое оно или нет. Например, у господина Коваленко, который занимает третью позицию в составленной нами базе данных, неизвестен год рождения, поэтому мы не можем с уверенностью говорить о наличие или отсутствии у него паспортных данных. А следовательно, значениями двух ячеек в строке, посвященной господину Коваленко будет Null-значение (первое – как неизвестное вообще, второе – как значение, природа которого неизвестна). Как и любые другие типы данных, Null-значения тоже имеют определенные **свойства**.

Перечислим самые существенные из них:

1) с течением времени понимание Null-значения может меняться. Например, у господина Карамазова (№ 2 в нашей базе данных) в 2014 г., т. е. по достижении совершеннолетия, Null-значение изменится на какое-то конкретное вполне определенное значение;
2) Null-значение может быть присвоено переменной или константе любого типа (числового, строкового, логического, дате, времени и т. д.);
3) результатом любых операций над выражениями с Null-значе-ниями в качестве операндов является Null-значение;
4) исключением из предыдущего правила являются операции конъюнкции и дизъюнкции в условиях законов поглощения (подробнее о законах поглощения смотрите в п. 4 лекции № 2).

3. Null-значения и общее правило вычисления выражений.

Общее правило работы с Null-значениями (то, что результат операций над Null-значениями есть Null-значение) применяется к следующим операциям:

1) к арифметическим;
2) к побитным операциям отрицания, конъюнкции и дизъюнкции (кроме законов поглощения);
3) к операциям со строками (например, конкотинации – сцепления строк);
4) к операциям сравнения (<, ≤, ≠, ≥, >).

В результате применений следующих операций будут получены Null-значения:

3 + Null, 1/ Null, (Иванов' + '' + Null) ≔ Null

Здесь вместо обычного равенства использована **операция подстановки** «≔» из-за особого характера работы с Null-значениями. Далее в подобных ситуациях также будет использоваться этот символ, который означает, что выражение справа от символа подстановки может заменить собой любое выражение из списка слева от символа подстановки.

Характер Null-значений приводит к тому, что часто в некоторых выражениях вместо ожидаемого нуля получается Null-значение, например:

(x – x), y * (x – x), x * 0 ≔ Null при x = Null.

Все дело в том, что при подстановке, например, в выражение (x – x) значения x = Null, мы получаем выражение (Null – Null), и в силу вступает общее правило вычисления значения выражения, содержащего Null-значения, и информация о том, что здесь Null-значение соответствует одной и той же переменной теряется.

Можно сделать вывод, что при вычислении любых операций, кроме логических, Null-значения интерпретируются как **неприменимые**, и поэтому в результате получается тоже Null-значение.

К не менее неожиданным результатам приводит использование Null-значений в операциях сравнения. Например, в следующих выражениях также получаются Null-значения вместо ожидаемых логических значений True или False:

(Null < Null); (Null ≤ Null); (Null = Null); (Null ≠ Null); (Null > Null); (Null ≥ Null) ≔ Null;

Таким образом, делаем вывод, что нельзя говорить о том, что Null-значение равно или не равно самому себе. Каждое новое вхождение Null-значения рассматривается как независимое, и каждый раз Null-значения воспринимаются как различные неизвестные значения. Этим Null-значения кардинально отличаются от всех остальных типов данных, ведь мы знаем, что обо всех пройденных ранее величинах и их типах с уверенностью можно было говорить, что они равны или не равны друг другу.

Итак, мы видим, что Null-значения не являются значениями переменных в обычном смысле этого слова. Поэтому становится невозможным сравнивать значения переменных или выражения, содержащие Null-значения, поскольку в результате мы будем получать не логические значения True или False, а Null-значения, как в следующих примерах:

(x < Null); (x ≤ Null); (x = Null); (x ≠ Null); (x > Null); (x ≥ Null) ≔ Null;

Поэтому по аналогии с пустыми значениями для проверки выражения на Null-значения необходимо использовать специальный предикат:

IsNull (<выражение>) , что буквально означает «есть Null».

Логическая функция возвращает значение True, если в выражении присутствует Null или оно равно Null, и False – в противном случае, но никогда не возвращает значение Null. Предикат IsNull может применяться к переменным и выражению любого типа. Если применять его к выражениям пустого типа, предикат всегда будет возвращать False.

Например:

IsNull(0)	False
IsNull(x + 'abc' + Null)	True
IsNull(2 * Null)	True
IsNull(Null)	True

Итак, действительно, видим, что в первом случае, когда предикат IsNull взяли от нуля, на выходе получилось значение False. Во всех случаях, в том числе во втором и третьем, когда аргументы логической функции оказались равными Null-значению, и в четвертом случае, когда сам аргумент и был изначально равен Null-значению, предикат выдал значение True.

4. Null-значения и логические операции.

Обычно в системах управления базами данных непосредственно поддерживаются только три логические операции: отрицание ¬, конъюнкция & и дизъюнкция ∨. Операции следования ⇒ и равносильности ⇔ выражаются через них с помощью подстановок:

$(x \Rightarrow y) \coloneqq (\neg x \vee y)$;
$(x \Leftrightarrow y) \coloneqq (x \Rightarrow y) \,\&\, (y \Rightarrow x)$;

Заметим, что эти подстановки полностью сохраняются и при использовании Null-значений.

Интересно, что при помощи операции отрицания «¬» любая из операций конъюнкция & или дизъюнкция ∨ может быть выражена одна через другую следующим образом:

$(x \,\&\, y) \coloneqq \neg (\neg x \vee \neg y)$;
$(x \vee y) \coloneqq \neg (\neg x \,\&\, \neg y)$;

На эти подстановки, как и на предыдущие, Null-значения влияния не оказывают.

А теперь приведем таблицы истинности логических операций отрицания, конъюнкции и дизъюнкции, но кроме привычных значений True и False, используем также Null-значение в качестве операндов. Для удобства введем следующие обозначения: вместо True будем писать t, вместо False – f, а вместо Null – n.

1. Отрицание ¬x.

x	$\neg\ x$
f	t
n	n
t	f

Стоит отметить следующие интересные моменты касательно операции отрицания с использованием Null-значений:

1) ¬¬x ≔ x – закон двойного отрицания;
2) ¬Null ≔ Null – Null-значение является неподвижной точкой.

2. Конъюнкция x & y.

y / x	f	n	t
f	f	f	f
n	f	n	n
t	f	n	t

Эта операция также имеет свои свойства:

1) $x \,\&\, y \coloneqq y \,\&\, x$ – коммутативность;
2) x & x ≔ x – идемпотентность;
3) False&y ≔ False, здесь False – поглощающий элемент;
4) True & y ≔ y , здесь True – нейтральный элемент.

3. **Дизъюнкция x ∨ y.**

y / x	f	n	t
f	f	n	t
n	n	n	t
t	t	t	t

Свойства:
1) $x \vee y \coloneqq y \vee x$ – коммутативность;
2) $x \vee x \coloneqq x$ – идемпотентность;
3) False ∨ y ≔ y , здесь False – нейтральный элемент;
4) True ∨ y ≔ True, здесь True – поглощающий элемент.

Исключение из общего правила составляют правила вычисления логических операций конъюнкция & и дизъюнкция ∨ в условиях действия **законов поглощения** :

(False & *y*) ≔ (*x* & False) ≔ False;
(True ∨ *y*) ≔ (*x* ∨ True) ≔ True;

Эти дополнительные правила формулируются для того, чтобы при замене Null-значения значениями False или True результат бы все равно не зависел бы от этого значения.

Как и ранее было показано для других типов операций, применение Null-значений в логических операциях могут также привести к неожиданным значениям. Например, логика на первый взгляд нарушена в **законе исключения третьего** (x ∨ ¬x) и в **законе рефлексивности** (x = x), поскольку при x ≔ Null имеем:

(x ∨ ¬x), (x = x) ≔ Null.

Законы не выполняются! Объясняется это так же, как и раньше: при подстановке Null-значения в выражение информация о том, что это значение сообщается одной и той же переменной теряется, а в силу вступает общее правило работы с Null-значениями.

Таким образом, делаем вывод: при выполнении логических операций с Null-значениями в качестве операнда эти значения определяются системами управления базами данных как **применимое, но неизвестное**.

5. Null-значения и проверка условий

Итак, из всего вышесказанного можно сделать вывод, что в логике систем управления базами данных имеются не два логических значения (True и False), а три, ведь Null-значение также рассматривается как одно из возможных логических значений. Именно поэтому на него часто ссылаются как на неизвестное значение, значение Unknown.

Однако, несмотря на это, в системах управления базами данных реализуется только двузначная логика. Поэтому условие с Null-значением (неопределенное условие) должно интерпретироваться машиной либо как True, либо как False.

В языке СУБД по умолчанию установлено опознавание условия с Null-значением как значения False. Проиллюстрируем это следующими примерами реализации в системах управления базами данных условных операторов If и While:

```
If P then A else B;
```

Эта запись означает: если P принимает значение True, то выполняется действие A, а если P принимает значение False или Null, то выполняется действие B.

Теперь применим к этому оператору операцию отрицания, получим:

```
If ¬P then B else A;
```

В свою очередь, этот оператор означает следующее: если ¬P принимает значение True, то выполняется действие B, а в том случае, если ¬P принимает значение False или Null, то будет выполняться действие A.

И снова, как мы видим, при появлении Null-значения мы сталкиваемся с неожиданными результатами. Дело в том, что два оператора If в этом примере не эквивалентны! Хотя один из них получен из другого отрицанием условия и перестановкой ветвей, т. е. стандартной операцией. Такие операторы в общем случае эквивалентны! Но в нашем примере мы видим, что Null-значению условия P в первом случае соответствует команда B, а во втором – A.

А теперь рассмотрим действие условного оператора While:

```
While P do A; B;
```

Как работает этот оператор? Пока переменная P имеет значение True, будет выполняться действие A, а как только P примет значение False или Null, выполнится действие B.

Но не всегда Null-значения интерпретируются как False. Например, в ограничениях целостности неопределенные условия опознаются как True (ограничения целостности – это условия, накладываемые на входные данные и обеспечивающие их корректность). Это происходит потому, что в таких ограничениях отвергнуть нужно только заведомо ложные данные.

И опять-таки в системах управления базами данных существует специальная **функция подмены IfNull (ограничения целостности, True)**, с помощью которой Null-значения и неопределенные условия можно представить в явном виде.

Перепишем условные операторы If и While с использованием этой функции:

1) If IfNull (P, False) then A else B;
2) While IfNull (P, False) do A; B;

Итак, функция подмены IfNull (выражение 1, выражение 2) возвращает значение первого выражения, если оно не содержит Null-значения, и значение второго выражения – в противном случае.

Надо заметить, что на тип возвращаемого функцией IfNull выражения никаких ограничений не накладывается. Поэтому с помощью этой функции можно явно переопределить любые правила работы с Null-значениями.

Данные в MySQL.

Самостоятельно познакомиться с материалом на сайте phpclub.ru, см.

http://phpclub.ru/mysql/doc/column-types.html

Краткий справочник по типам данных MySQL.

MySQL поддерживает числовые, строковые, календарные данные и данные типа NULL. Рассмотрим их по очереди.

Числовые типы данных

Тип данных	Объем памяти	Диапазон	Описание
TINYINT (M)	1 байт	от -128 до 127 или от 0 до 255	Целое число. Может быть объявлено положительным с помощью ключевого слова UNSIGNED, тогда элементам столбца нельзя будет присвоить отрицательное значение. Необязательный параметр M - количество отводимых под число символов. Необязательный атрибут ZEROFILL позволяет свободные позиции по умолчанию заполнить нулями. *Примеры:* TINYINT - хранит любое число в диапазоне от -128 до 127. TINYINT UNSIGNED - хранит любое число в диапазоне от 0 до 255. TINYINT (2) - предполагается, что значения будут двузначными, но по факту будет хранить и трехзначные. TINYINT (3) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 002.
SMALLINT (M)	2 байта	от -32768 до 32767 или от 0 до 65535	Аналогично предыдущему, но с большим диапазоном.

			Примеры: SMALLINT - хранит любое число в диапазоне от -32768 до 32767. SMALLINT UNSIGNED - хранит любое число в диапазоне от 0 до 65535. SMALLINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить и пятизначные. SMALLINT (4) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 0002.
MEDIUMINT (M)	3 байта	от -8388608 до 8388608 или от 0 до 16777215	Аналогично предыдущему, но с большим диапазоном. *Примеры:* MEDIUMINT - хранит любое число в диапазоне от -8388608 до 8388608. MEDIUMINT UNSIGNED - хранит любое число в диапазоне от 0 до 16777215. MEDIUMINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить и семизначные. MEDIUMINT (5) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 00002.
INT (M) или INTEGER (M)	4 байта	от -2147683648 до 2147683648 или от 0 до 4294967295	Аналогично предыдущему, но с большим диапазоном. *Примеры:* INT - хранит любое число в диапазоне от -2147683648 до 2147683648. INT UNSIGNED - хранит любое число в диапазоне от 0 до 4294967295. INT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить максимально возможные. INT (5) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 00002.
BIGINT (M)	8 байта	от -2^{63} до $2^{63}-1$ или от 0 до 2^{64}	Аналогично предыдущему, но с большим диапазоном. *Примеры:* BIGINT - хранит любое число в диапазоне от -2^{63}до $2^{63}-1$. BIGINT UNSIGNED - хранит любое число в диапазоне от 0 до 2^{64}. BIGINT (4) - предполагается, что значения будут четырехзначные, но по факту будет хранить максимально возможные. BIGINT (7) ZEROFILL - свободные позиции слева заполнит нулями. Например, величина 2 будет отображаться, как 0000002.

BOOL или BOOLEAN	1 байт	либо 0, либо 1	Булево значение. 0 - ложь (false), 1 - истина (true).
DECIMAL (M,D) или DEC (M,D) или NUMERIC (M,D)	M + 2 байта	зависят от параметров M и D	Используются для величин повышенной точности, например, для денежных данных. M - количество отводимых под число символов (максимальное значение - 64). D - количество знаков после запятой (максимальное значение - 30). *Пример:* DECIMAL (5,2) - будет хранить числа от -99,99 до 99,99.
FLOAT (M,D)	4 байта	мин. значение +(-) $1.175494351 * 10^{-39}$ макс. значение +(-) $3.402823466 * 10^{38}$	Вещественное число (с плавающей точкой). Может иметь параметр UNSIGNED, запрещающий отрицательные числа, но диапазон значений от этого не изменится. M - количество отводимых под число символов. D - количество символов дробной части. *Пример:* FLOAT (5,2) - будет хранить числа из 5 символов, 2 из которых будут идти после запятой (например: 46,58).
DOUBLE (M,D)	8 байт	мин. значение +(-) $2.2250738585072015 * 10^{-308}$ макс. значение +(-) $1.797693134862315 * 10^{308}$	Аналогично предыдущему, но с большим диапазоном. *Пример:* DOUBLE - будет хранить большие дробные числа.

Необходимо понимать, чем больше диапазон значений у типа данных, тем больше памяти он занимает. Поэтому, если предполагается, что значения в столбце не будут превышать 100, то используйте тип TINYINT. Если при этом все значения будут положительными, то используйте атрибут UNSIGNED. Правильный выбор типа данных позволяет сэкономить место для хранения этих данных.

Строковые типы данных

Тип данных	Объем памяти	Максимальный размер	Описание
CHAR (M)	M символов	M символов	Позволяет хранить строку фиксирован длины M. Значение M - от 0 до 65535. *Примеры:* CHAR (8) - хранит строки из 8 символов и занимает 8 байтов. Например, любое из следующих значений: '', 'Иван','Ирина', 'Сергей' будет занимать по 8 байтов памяти. А при попытке ввести значение 'Александра', оно будет усечено до 'Александ', т.е. до 8 символов.
VARCHAR (M)	L+1 символов	M символов	Позволяет хранить переменные строки длиной L. Значение M - от 0 до 65535. *Примеры:* VARCHAR (3) - хранит строки максимум из 3 символов, но пустая строка '' занимает 1 байт памяти, строка 'а' - 2 байта, строк 'аа' - 3 байта, строка 'ааа' - 4 байта. Значение более 3 символов будет усечено до 3.
BLOB, TEXT	L+2 символов	2^{16}-1 символов	Позволяют хранить большие объемы текста. Причем тип TEXT используется для хранения именно текста, а BLOB - для хранения изображений, звука, электронных документов и т.д.
MEDIUMBLOB, MEDIUMTEXT	L+3 символов	2^{24}-1 символов	Аналогично предыдущему, но с большим размером.
LONGBLOB, LONGTEXT	L+4 символов	2^{32}-1 символов	Аналогично предыдущему, но с большим размером.
ENUM ('value1', 'value2', ...,'valueN')	1 или 2 байта	65535 элементов	Строки этого типа могут принимать только одно из значений указ. множества. *Пример:* ENUM ('да', 'нет') - в столбце с таким типом может храниться только одно из имеющихся значений. Удобно использовать, если предусмотрено, что в столбце должен храниться ответ на вопрос.
SET ('value1', 'value2', ...,'valueN')	до 8 байт	64 элемента	Строки этого типа могут принимать любой или все элементы из значений указанного множества. *Пример:* SET ('первый', 'второй') - в столбце с таким типом может храниться одно из перечисленных значений, оба сразу или значение может отсутствовать вовсе.

Календарные типы данных

Тип данных	Объем памяти	Диапазон	Описание
DATE	3 байта	от '1000-01-01' до '9999-12-31'	Предназначен для хранения даты. В качестве первого значения указывается год в формате "YYYY", через дефис - месяц в формате "MM", а затем день в формате "DD". В качестве разделителя может выступать не только дефис, а любой символ отличный от цифры.
TIME	3 байта	от '-838:59:59' до '838:59:59'	Предназначен для хранения времени суток. Значение вводится и хранится в привычном формате - hh:mm:ss, где hh - часы, mm - минуты, ss - секунды. В качестве разделителя может выступать любой символ отличный от цифры.
DATATIME	8 байт	от '1000-01-01 00:00:00' до '9999-12-31 23:59:59'	Предназначен для хранения и даты и времени суток. Значение вводится и хранится в формате - YYYY-MM-DD hh:mm:ss. В качестве разделителей могут выступать любые символы отличные от цифры.
TIMESTAMP	4 байта	от '1970-01-01 00:00:00' до '2037-12-31 23:59:59'	Предназначен для хранения даты и времени суток в виде количества секунд, прошедших с полуночи 1 января 1970 года (начало эпохи UNIX).
YEAR (M)	1 байт	от 1970 до 2069 для M=2 и от 1901 до 2155 для M=4	Предназначен для хранения года. M - задает формат года. Например, YEAR (2) - 70, а YEAR (4) - 1970. Если параметр M не указан, то по умолчанию считается, что он равен 4.

NULL данные

По сути это указатель на возможность отсутствия значения, т.е. обязательные и необязательные поля. Для того, чтобы хранить такую информацию в БД используют два значения:
NOT NULL (значение не может отсутствовать) для полей логин и пароль,
NULL (значение может отсутствовать) для полей дата рождения и пол.

По умолчанию всем столбцам присваивается тип NOT NULL, поэтому его можно явно не указывать. *Пример:* create table users (login varchar(20), passw varchar(15), sex enum('man', 'woman') NULL, birth date NULL);

Типология архитектур доступа к записям и индексация таблиц.

Программисту или пользователю необходимо иметь возможность обращаться к отдельным, нужным ему записям или отдельным элементам данных. В зависимости от уровня программного обеспечения прикладной программист может использовать следующие способы:

• Задать машинный адрес данных и в соответствии с физическим форматом записи прочитать значение. Это случай, когда программист должен быть «навигатором».

• Сообщить системе имя записи или элемента данных, которые он хочет получить, и возможно, организацию набора данных. В этом случае система сама произведет выборку (по предыдущей схеме), но для этого она должна будет использовать вспомогательную информацию о структуре данных и организации набора. Такая информация по существу будет избыточной по отношению к объекту, однако общение с базой данных не будет требовать от пользователя знаний программиста и позволит переложить заботы о размещении данных на систему.

В качестве ключа, обеспечивающего доступ к записи, можно использовать идентификатор – отдельный элемент данных. Ключ, который идентифицирует запись единственным образом, называется первичным (главным).

В том случае, когда ключ идентифицирует некоторую группу записей, имеющих определенное общее свойство, ключ называется вторичным (альтернативным). Набор данных может иметь несколько вторичных ключей, необходимость введения которых определяется прак-

тической необходимостью – оптимизацией процессов нахождения записей по соответствующему ключу.

Иногда в качестве идентификатора записи используют составной сцепленный ключ – несколько элементов данных, которые в совокупности, например, обеспечат уникальность идентификации каждой записи набора данных.

Один из способов использования вторичного ключа в качестве входа – организация инвертированного списка, каждый вход которого содержит значение ключа вместе со списком идентификаторов соответствующих записей.

Существует несколько различных способов адресации и поиска записей.

1. Последовательное сканирование файла. Наиболее простым способом локализации записи является сканирование файла с проверкой ключа каждой записи. Этот способ, однако, требует слишком много времени и может применяться, когда каждая запись все равно должна быть прочитана.

2. Блочный поиск. Если записи упорядочены по ключу, то при сканировании файла не требуется чтение каждой записи. ЭВМ могла бы, например, просматривать каждую сотую запись в последовательности возрастания ключей. При нахождении записи с ключом большим, чем искомое значение, просматриваются последние 99 записей, которые были пропущены.

Этот способ называется блочным поиском. Записи группируются в блоки и каждый блок проверяется по одному разу до тех пор, пока не будет найден нужный блок. Иногда данный способ называется поиском с пропусками.

3. Двоичный поиск, B-дерево. При двоичном поиске в файле записей, упорядоченных по ключу, анализируется запись, находящаяся в середине поисковой области файла (изначально всего файла), а ее ключ сравнивается с поисковым ключом. Затем поисковая область делится пополам и процесс повторяется для соответствующей половины области, пока не будет обнаружено искомое значение или длина области не станет равной 1. Число сравнений в этом случае будет меньше, чем для случая блочного поиска.

Двоичный поиск эффективен для поиска в файлах, организованных в виде двоичного дерева с указателями, когда поиск проходит в направлении, задаваемом указателями. Кроме того, добавление в файл новых записей не приводит к сдвигу других записей, что требует много времени и является достаточно сложной процедурой.

Таким образом, двоичный поиск более пригоден для поиска в индексе файла, чем в самом файле.

4. Индексно-последовательные файлы. Если файл упорядочен по ключам, то для адресации может использоваться таблица, называемая индексом, связывающая ключ хранимой записи с ее относительным или абсолютным адресом во внешней памяти.

Индекс можно определить как таблицу, с которой связана процедура, воспринимающая на входе информацию о некоторых значениях атрибутов и выдающая на выходе информацию, способствующую быстрой локализации записи или записей, которые имеют заданные значения атрибутов.

Если записи файла упорядочены по ключу, индекс обычно содержит не ссылки на каждую запись, а ссылки на блоки записей, внутри которых можно выполнять поиск или сканирование. Хранение ссылок на блоки записей, а не на отдельные записи в значительной степени уменьшает размер индекса. Причем даже в этом случае индекс часто оказывается слишком большим для поиска, и поэтому используется индекс индекса.

5. Индексно-произвольные файлы. Произвольный (не упорядоченный по ключу) файл можно индексировать точно так же, как и последовательный файл. Однако при этом индекс должен содержать по одному элементу для каждой записи файла, а не для блока записей. Более того, в нем должны содержаться полные абсолютные (или относительные) адреса, в то время как в индексе последовательного файла адреса могут содержаться в усеченном виде, так как старшие знаки последовательных адресов будут совпадать.

Произвольные файлы в основном используются для обеспечения возможности адресации записей файла с несколькими ключами. Если файл упорядочен по одному ключу, то он не упорядочен по другому ключу. Для каждого типа ключей может существовать свой индекс: для упорядоченных ключей индекс будет иметь по одному элементу на блок записей, для других ключей индексы буду более длинными, так как должны будут содержать по одному

элементу для каждой записи. Ключ, который чаще всего используется при адресации файла, обычно служит для его упорядочения.

В индексно-произвольных файлах часто используются символические, а не абсолютные адреса, так как при добавлении новых или удалении старых записей изменяется местоположение записей. Если в записях имеется несколько ключей, то индекс вторичного ключа может содержать в качестве выхода первичный ключ записи. При определении же местоположения записи по ее первичному ключу можно использовать какой-нибудь другой способ адресации. По этому методу поиск осуществляется медленнее, чем поиск, при котором физический адрес записи определяется по индексу. В файлах, в которых положение записей часто изменяется, символическая адресация может оказаться предпочтительнее.

6. Адресация с помощью ключей, преобразуемых в адрес. Известно много методов преобразования ключа непосредственно в значение адреса в файле. В тех случаях, когда возможно преобразование значения ключа непосредственно в значение адреса в файле, такой способ адресации обеспечивает самый быстрый доступ; при этом нет необходимости организовывать поиск внутри файла или выполнять операции с индексами.
В некоторых приложениях адрес может быть вычислен на основе значений некоторых элементов данных записи.

К недостаткам данного способа относится малое заполнение файла: в файле остаются свободные участки, поскольку ключи преобразуются не в непрерывное множество адресов. Другим недостатком схем прямой адресации является их малая гибкость. Машинные адреса

записей могут измениться при обновлении файла. Для устранения этого недостатка прямую адресацию обычно выполняют в два этапа. Сначала ключ преобразуется в порядковый номер, который затем преобразуется в машинный адрес.

7. Хэширование. Простым и полезным способом вычисления адреса является хэширование (перемешивание). В данном методе ключ преобразуется в квазислучайное число, которое используется для определения местоположения записи.

Более экономичным является указание на область, в которой размещается группа записей. Эта область называется участком записей (slot, bucket).

При первоначальной загрузке файла адрес, по которому должна быть размещена запись, определяется следующим образом:

1) Ключ записи преобразуется в квазислучайное число, находящееся в диапазоне от 1 до числа участков, используемых для размещения записей.

2) Число преобразуется в адрес участка и, если на участке есть свободное место, то логическая запись размещается на нем.

3) Если участок заполнен, запись должна быть размещена на участке переполнения – следующий по порядку участок либо участок в отдельной области переполнения.

При чтении записей из файла их поиск выполняется аналогично, причем может оказаться, что для поиска записи потребуется чтение нескольких участков переполнения.

Из-за вероятностей природы алгоритма в этом способе не удалось достичь 100% плотности заполнения памяти, однако для большинства файлов может быть достигнута плотность 80 или 90%; при этом память для индексов не требуется. Большинство записей можно найти за одно обращение, но для некоторых потребуется второе обращение (при переполнении). Для очень небольшой части записей потребуется третье или четвертое обращение к файлу.

Кроме того, в этом случае менее эффективно используется память, чем в индексных методах; записи не упорядочены для последовательной обработки.

8. Комбинации способов адресации. При адресации записей некоторых файлов используются комбинации перечисленных выше способов. Например, с помощью индекса может определяться ограниченная поисковая область файла, затем эта область просматривается последовательно, либо в ней выполняется двоичный поиск. С помощью алгоритма прямой адресации может определяться нужный раздел индекса и, таким образом, исчезает необходимость поиска во всем индексе.

9. Индексация.

Создание индексов значительно ускоряет работу с таблицами.

Индекс часто представляет из себя структуру типа B-дерева (см. выше), но могут использоваться и другие структуры.

Какие поля необходимо индексировать:

- Первоначальное определение структуры индексов производится разработчиком на стадии создания прикладной системы. Обязательно надо строить индексы для первичных ключей, поскольку по их значениям осуществляется доступ к данным при операциях соединения двух и более таблиц.

- В дальнейшем структура индексов уточняется администратором системы по результатам анализа ее работы, учета наиболее часто выполняющихся запросов и т.д.

9.1. SQL DDL: Операторы создания индексов.

Создание индекса:

CREATE [UNIQUE] INDEX <имя_индекса> ON <имя_таблицы> (<имя_столбца>,...)

Эта команда создает индекс с заданным именем для таблицы <имя_таблицы> по столбцам, входящим в список, указанный в скобках. В случае указания необязательного параметра **UNIQUE** СУБД будет проверять каждое значение индекса на уникальность.

Удаление индекса:

DROP INDEX <имя_индекса>

Пример, создание индексов для первичных ключей таблиц в БД *publications*:

```
CREATE INDEX au_index ON authors (au_id);
CREATE INDEX title_index ON titles (title_id);
CREATE INDEX pub_index ON publishers (pub_id);
CREATE INDEX site_index ON wwwsites (site_id);
```

Пример, создание индексов для часто используемых запросов в БД *publications*:

```
CREATE INDEX au_names ON authors (author);
```

Т.к. можно ожидать, что одним из наиболее частых запросов будет выборка всех публикаций данного автора, для минимизации времени этого запроса был построен индекс для таблицы *authors* по именам авторов.

Понятие о транзакции.

Транзакция – неделимая с точки зрения воздействия на БД последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), такая, что:

1) либо результаты всех операторов, входящих в транзакцию, отображаются в БД;
2) либо воздействие всех операторов полностью отсутствует.

При этом для поддержания ограничений целостности на уровне БД допускается их нарушение внутри транзакции так, чтобы к моменту завершения транзакции условия целостности были соблюдены.

Для обеспечения контроля целостности каждая транзакция должна начинаться при целостном состоянии БД и должна сохранить это состояние целостным после своего завершения. Если операторы, объединенные в транзакцию, выполняются, то происходит нормальное завершение транзакции, и БД переходит в обновленное (целостное) состояние. Если же происходит сбой при выполнении транзакции, то происходит так называемый откат к исходному состоянию БД.

Модели транзакций. Рассмотрим две модели транзакций, используемые в большинстве коммерческих СУБД: модель автоматического выполнения транзакций и модель управляемого выполнения транзакций, обе основаны на инструкциях языка SQL – COMMIT и ROLLBACK.

1. Автоматическое выполнение транзакций.

В стандарте ANSI/ISO зафиксировано, что транзакция автоматически начинается с выполнения пользователем или программой первой инструкции SQL. Далее происходит последовательное выполнение инструкций до тех пор, пока транзакция не завершается одним из двух способов:

- инструкцией COMMIT, которая выполняет завершение транзакции: изменения, внесенные в БД, становятся постоянными, а новая транзакция начинается сразу после инструкции COMMIT;

- инструкцией ROLLBACK, которая отменяет выполнение текущей транзакции и возвращает БД к состоянию начала транзакции, новая транзакция начинается сразу после инструкции ROLLBACK.

Такая модель создана на основе модели, принятой в СУБД DB2.

2. Управляемое выполнение транзакций.

Отличная от модели ANSI/ISO модель транзакций используется в СУБД Sybase, где применяется диалект MySQL, в котором для обработки транзакций служат инструкции:

- инструкция BEGIN сообщает о начале транзакции, т.е. начало транзакции задается явно;

- инструкция COMMIT сообщает об успешном выполнении транзакции, но при этом новая транзакция не начинается автоматически;

- инструкция ROLLBACK отменяет выполнение текущей транзакции и возвращает БД к состоянию начала транзакции.

3. Виды конфликтов при параллельном выполнении транзакций.

При параллельной обработке данных (т.е. при совместной работе с БД нескольких пользователей) СУБД должна гарантировать, что пользователи не будут мешать друг другу. Средства обработки транзакций позволяют изолировать пользователей друг от друга таким образом, чтобы у каждого из них было ощущение монопольной работы с БД.

Транзакции являются подходящими единицами изолированности пользователей благодаря свойству сохранения целостности БД. Действительно, если с каждым сеансом работы с базой данных ассоциируется транзакция, то каждый пользователь начинает работу с согласованным состоянием базы данных, т.е. с таким состоянием, в котором база данных могла бы находиться, даже если бы пользователь работал с ней в одиночку.

Чтобы понять, как должны выполняться параллельные транзакции, рассмотрим проблемы, возникающие при параллельной работе с данными.

3.1. Пропавшие обновления

Рассмотрим пример работы двух диспетчеров модифицированной БД «Сессия». Пусть Диспетчер 1 вносит в текущий учебный план для каждой дисциплины, читаемой на третьем курсе, сведения о преподавателях, параллельно изменяя при этом значение столбца «Нагрузка» в таблице «Кадровый состав», а Диспетчер 2 выполняет такую же операцию для дисциплин второго курса.

Диспетчер 1 начинает работу по изменению таблицы «Учебный план» для Дисциплины 1 с количеством часов, равным 50. В столбец ID_Преподаватель для этой дисциплины предполагается внести значение 5. Запрос текущей нагрузки преподавателя возвращает значение 350, и Диспетчер 1 подтверждает изменение таблицы «Учебный план». При этом выполняются дополнительные действия по изменению столбца Нагрузка в таблице «Кадровый состав» для строки с ID_Преподаватель = 5 (в столбец заносится значение 350+50=400).

До завершения операции Диспетчера 1, Диспетчер 2 начинает те же действия для Дисциплины 2 с количеством часов 32, которую должен читать тот же преподаватель. Запрос текущей нагрузки преподавателя также возвращает значение 350, с которым и работает дальше Диспетчер 2. Выполнив те же операции, что и Диспетчер 1 (но после него), Диспет-

чер 2 помещает в столбец Нагрузка значение 382, отменив тем самым предыдущие изменения Диспетчера 1.

Для избежания подобных ситуаций к СУБД по части параллельно выполняемых транзакций предъявляется минимальное требование – отсутствие потерянных изменений.

3.2. Чтение «грязных» данных.

Диспетчер 1 и Диспетчер 2 опять выполняют действия, описанные в предыдущем примере, но Диспетчер 2 начинает запрашивать данные о нагрузке преподавателя в тот момент, когда изменения, сделанные Диспетчером 1, уже зафиксированы в столбце Нагрузка, а транзакция еще не закончилась. Запрос Диспетчера 2 возвращает значение 400, и Диспетчер 2 вынужден отменить свои действия потому, что 400 часов – это максимальное разрешенное значение нагрузки. Между тем транзакция Диспетчера 1 закончилась возвратом к исходному состоянию, т. .е на самом деле Диспетчер 2 мог бы успешно завершить операцию.

Это тоже не соответствует требованию изолированности пользователей (каждый пользователь начинает свою транзакцию при согласованном состоянии базы данных и вправе ожидать увидеть согласованные данные). Чтобы избежать ситуации чтения «грязных» данных, необходимо требовать, чтобы до завершения одной транзакции, изменившей некоторый объект, никакая другая транзакция не могла читать изменяемый объект.

3.3. Чтение несогласованных данных.

По-прежнему Диспетчер 1 выполняет операцию по заполнению строки учебного плана, а Диспетчер 2 при выполнении своей операции должен сделать выбор между двумя преподавателями в соответствии с их текущей нагрузкой.

Начиная работу практически одновременно с Диспетчером 1, Диспетчер 2 получает следующие сведения: нагрузка первого преподавателя (ID_Преподаватель = 5) составляет 350 часов, а нагрузка второго преподавателя (ID_Преподаватель = 7) составляет 370 часов. Далее Диспетчер 2 принимает решение в пользу первого преподавателя, но повторный запрос нагрузки возвращает значение 400, так как Диспетчер 1 уже сохранил новые данные в таблице «Кадровый состав».

В большинстве систем обеспечение изолированности пользователей в подобных ситуациях является максимальным требованием к синхронизации транзакций.

3.4. Строки-призраки.

К более тонким проблемам изолированности транзакций относится так называемая проблема строк-призраков, вызывающая ситуации, которые также противоречат изолированности пользователей. Рассмотрим следующий сценарий.

Диспетчер 1 инициирует Транзакцию 1, которая выполняет оператор выборки строк таблицы в соответствии с некоторыми условиями (например, формирование списка студентов, сдавших дисциплину с ID_Дисциплина = N, по таблице «Сводная ведомость»). До завершения Транзакции 1 Транзакция 2, вызванная Диспетчером 2, вставляет в таблицу «Сводная ведомость» новую строку, удовлетворяющую условию отбора Транзакции 1 (данные о результате сдачи дисциплины N еще одним студентом), и успешно завершается. Транзакция 1 повторно выполняет оператор выборки, и в результате появляется строка, которая отсутствовала при первом выполнении оператора. Конечно, такая ситуация противоречит идее изолированности транзакций.