Базы данных

Свинцов Дмитрий

19-06-2019

Оглавление

Глава 1

Содержание

1.1 Системы управления базами данных

См.также:

https://ru.wikipedia.org/wiki/\T2A\CYRS\T2A\cyri\T2A\cyrs\T2A\cyrt\T2A\cyre\T2A\cyrm\T2A\cyra_\T2A\cyru\T2A\cyrp\T2A\cyrr\T2A\cyra\T2A\cyrv\T2A\cyrl\T2A\cyre\T2A\cyrn\T2A\cyri\T2A\cyrya_\T2A\cyrb\T2A\cyra\T2A\cyrz\T2A\cyra\T2A\cyrm\T2A\cyri_\T2A\cyrd\T2A\cyra\T2A\cyrn\T2A\cyrn\T2A\cyrery\T2A\cyrh

Система управления базами данных (СУБД) — совокупность программных и лингвистических средств общего или специального назначения, обеспечивающих управление созданием и использованием баз данных.

1.1.1 Основные функции СУБД

- управление данными во внешней памяти (на дисках);
- управление данными в оперативной памяти с использованием дискового кэша;
- журнализация изменений, резервное копирование и восстановление базы данных после сбоев;
- поддержка языков БД (язык определения данных, язык манипулирования данными).

Обычно современная СУБД содержит следующие компоненты:

- **ядро**, которое отвечает за управление данными во внешней и оперативной памяти и журнализацию,
- **процессор языка базы данных**, обеспечивающий оптимизацию запросов на извлечение и изменение данных и создание, как правило, машинно-независимого исполняемого внутреннего кода,
- **подсистему поддержки времени исполнения**, которая интерпретирует программы манипуляции данными, создающие пользовательский интерфейс с СУБД
- а также **сервисные программы** (внешние утилиты), обеспечивающие ряд дополнительных возможностей по обслуживанию информационной системы.

1.1.2 Классификации СУБД

По модели данных

Иерархические

Используется представление базы данных в виде древовидной (иерархической) структуры, состоящей из объектов (данных) различных уровней.

Между объектами существуют связи, каждый объект может включать в себя несколько объектов более низкого уровня. Такие объекты находятся в отношении предка (объект более близкий к корню) к потомку (объект более низкого уровня), при этом возможна ситуация, когда объект-предок не имеет потомков или имеет их несколько, тогда как у объекта-потомка обязательно только один предок. Объекты, имеющие общего предка, называются близнецами (в программировании применительно к структуре данных дерево устоялось название братья).

Иерархической базой данных является файловая система, состоящая из корневого каталога, в котором имеется иерархия подкаталогов и файлов.

Примеры: Caché, Google App Engine Datastore API.

Сетевые

Сетевые базы данных подобны иерархическим, за исключением того, что в них имеются указатели в обоих направлениях, которые соединяют родственную информацию.

Примеры: Caché.

Реляционные

Практически все разработчики современных приложений, предусматривающих связь с системами баз данных, ориентируются на реляционные СУБД. По оценке Gartner в 2013 году рынок реляционных СУБД составлял 26 млрд долларов с годовым приростом около 9%, а к 2018 году рынок реляционных СУБД достигнет 40 млрд долларов. В настоящее время абсолютными лидерами рынка СУБД являются компании Oracle, IBM и Microsoft, с общей совокупной долей рынка около 90%, поставляя такие системы как Oracle Database, IBM DB2 и Microsoft SQL Server.

Объектно-ориентированные

Управляют базами данных, в которых данные моделируются в виде объектов, их атрибутов, методов и классов.

Этот вид СУБД позволяет работать с объектами баз данных так же, как с объектами в программировании в объектно-ориентированных языках программирования. ООСУБД расширяет языки программирования, прозрачно вводя долговременные данные, управление параллелизмом, восстановление данных, ассоциированные запросы и другие возможности.

Примеры: GemStone.

Объектно-реляционные

Этот тип СУБД позволяет через расширенные структуры баз данных и язык запросов использовать возможности объектно-ориентированного подхода: бъекты, классы и наследование.

Зачастую все те СУБД, которые называются реляционными, являются, по факту, объектно-реляционными.

В данном курсе мы будем, в первую очередь, гооврить об этом виде СУБД.

Примеры: PostgreSQL, DB2, Oracle, Microsoft SQL Server.

По степени распределённости

- Локальные СУБД (все части локальной СУБД размещаются на одном компьютере)
- Распределённые СУБД (части СУБД могут размещаться на двух и более компьютерах).

По способу доступа к БД

Файл-серверные

В файл-серверных СУБД файлы данных располагаются централизованно на файл-сервере. СУБД располагается на каждом клиентском компьютере (рабочей станции). Доступ СУБД к данным осуществляется через локальную сеть. Синхронизация чтений и обновлений осуществляется посредством файловых блокировок. Преимуществом этой архитектуры является низкая нагрузка на процессор файлового сервера. Недостатки: потенциально высокая загрузка локальной сети; затруднённость или невозможность централизованного управления; затруднённость или невозможность обеспечения таких важных характеристик как высокая надёжность, высокая доступность и высокая безопасность. Применяются чаще всего в локальных приложениях, которые используют функции управления БД; в системах с низкой интенсивностью обработки данных и низкими пиковыми нагрузками на БД.

На данный момент файл-серверная технология считается устаревшей, а её использование в крупных информационных системах — недостатком.

Примеры: Microsoft Access, Paradox, dBase, FoxPro, Visual FoxPro.

Клиент-серверные

Клиент-серверная СУБД располагается на сервере вместе с БД и осуществляет доступ к БД непосредственно, в монопольном режиме. Все клиентские запросы на обработку данных обрабатываются клиент-серверной СУБД централизованно. Недостаток клиент-серверных СУБД состоит в повышенных требованиях к серверу. Достоинства: потенциально более низкая загрузка локальной сети; удобство централизованного управления; удобство обеспечения таких важных характеристик как высокая надёжность, высокая доступность и высокая безопасность.

Примеры: Oracle, Firebird, Interbase, IBM DB2, Informix, MS SQL Server, Sybase Adaptive Server Enterprise, PostgreSQL, MySQL, Caché, ЛИНТЕР.

Встраиваемые

Встраиваемая СУБД — СУБД, которая может поставляться как составная часть некоторого программного продукта, не требуя процедуры самостоятельной установки. Встраиваемая СУБД предназначена для локального хранения данных своего приложения и не рассчитана на коллективное использование в сети. Физически встраиваемая СУБД чаще всего реализована в виде подключаемой библиотеки. Доступ к данным со стороны приложения может происходить через SQL либо через специальные программные интерфейсы (API).

Примеры: OpenEdge, SQLite, BerkeleyDB, Firebird Embedded, Microsoft SQL Server Compact, ЛИНТЕР.

Стратегии работы с внешней памятью

- СУБД с непосредственной записью — это СУБД, в которых все измененные блоки данных незамедлительно записываются во внешнюю память при поступлении сигнала подтверждения любой транзакции. Такая стратегия используется только при высокой эффективности внешней памяти.
- СУБД с отложенной записью — это СУБД, в которых изменения аккумулируются в буферах внешней памяти до наступления любого из следующих событий:
  - контрольной точки;
  - конец пространства во внешней памяти, отведенное под журнал. СУБД выполняет контрольную точку и начинает писать журнал сначала, затирая предыдущую информацию;
  - останов. СУБД ждёт, когда всё содержимое всех буферов внешней памяти будет перенесено во внешнюю память, после чего делает отметки, что останов базы данных выполнен корректно;
  - при нехватке оперативной памяти для буферов внешней памяти.

Такая стратегия позволяет избежать частого обмена с внешней памятью и значительно увеличить эффективность работы СУБД.

1.2 Реляционная модель данных

См.также:

Michael Bayer: Introduction to SQLAlchemy

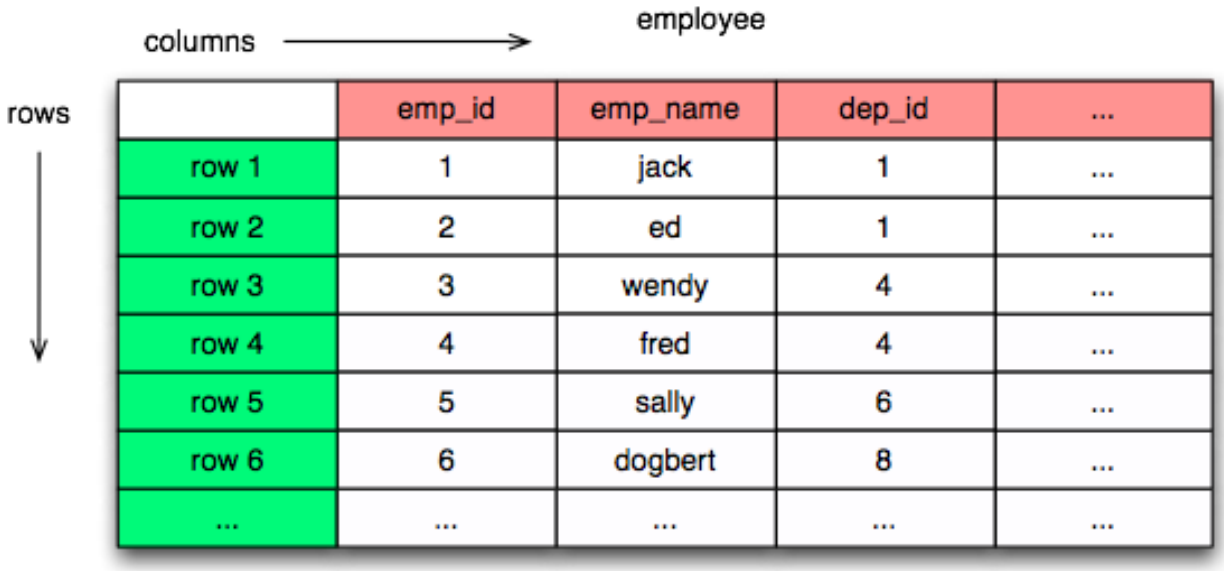
Wikipedia: ACID

Этот документ является переводом части лекции Майка Байера (Michael Bayer) о SQLAlchemy, которая была представлена на Pycon 2013.

1.2.1 Реляционная модель

Модель представляет собой фиксированную структуру математических понятий, которая описывает то, как будут представлены данные. Базовой единицей данных в пределах реляционной модели является таблица.

Таблица



	emp_id	emp_name	dep_id	...
row 1	1	jack	1	...
row 2	2	ed	1	...
row 3	3	wendy	4	...
row 4	4	fred	4	...
row 5	5	sally	6	...
row 6	6	dogbert	8	...
...	...	...	...	...

Таблица - это базовая единица данных. В реляционной алгебре она называется «отношение» (relation). Состоит из столбцов (columns), которые определяют конкретные типы данных. Данные в таблице организованы в строки (rows), которые содержат множества значений столбцов.

Язык описания данных (DDL)

В SQL для создания таблицы используется оператор CREATE TABLE. Этот оператор является примером языка описания данных (DDL). DDL служит для описания структуры базы данных. Пример использования:

```
CREATE TABLE employee (
  emp_name VARCHAR(30),
  dep_id INTEGER
)
```

Первичные ключи

При создании таблицы могут быть использованы различные «ограничения» (constraints), которые содержат правила, указывающие, какие данные представлены в ней. Одним из самых используемых ограничений является первичный ключ (primary key constraint), который гарантирует, что каждая строка таблицы должна содержать уникальное значение. Первичный ключ может состоять из одного или нескольких столбцов. Первичные ключ, состоящие из нескольких столбцов, также называются «составными» (composite).

Правильным считается наличие первичного ключа во всех таблицах базы данных. При этом существует два варианта первичных ключей: искусственный (surrogate primary key) и естественный (natural primary key).

Первый вариант обычно представляет собой целочисленный идентификатор. Применяется там где нет возможности использовать натуральный первичный ключ. Позволяют решать те же практические задачи, что и естественные: улучшение производительности памяти и индексов при операциях обновления.

Второй же вариант представляет собой данные, которые уже присутствуют в описываемой предметной области. Например, почтовые индексы могут быть использованы как естественные первичные ключи без дополнительной обработки. Их использование, если, конечно, оно возможно, считается более правильным, чем искусственных.

Пример создания первичного ключа:

```
CREATE TABLE employee (
  emp_id INTEGER,
  emp_name VARCHAR(30),
  dep_id INTEGER,
  PRIMARY KEY (emp_id)
)
```

Внешние ключи

В то время как одна таблица имеет первичный ключ, другая таблица может иметь ограничение, описывающее, что её строки ссылаются на гарантированно существующие строки в первой таблице. Это реализуется через создание в «удалённой» таблице («потомке») столбца (может быть и несколько), значениями которого являются значения первичного ключа из «локальной» таблицы («родителя»). Вместе наборы этих столбцов составляют внешний ключ (foreign key constraint), который является механизмом базы данных, гарантирующим что значения в «удалённых» столбцах присутствуют как первичные ключи в «локальных». Это ограничение контролирует все операции на этих таблицах: добавление / изменение данных в «удалённой» таблице; удаление / изменение данных в «родительской» таблице. Внешний ключ проверяет, чтобы данные корректно присутствовали в обоих таблицах. Иначе операции будут отменены.

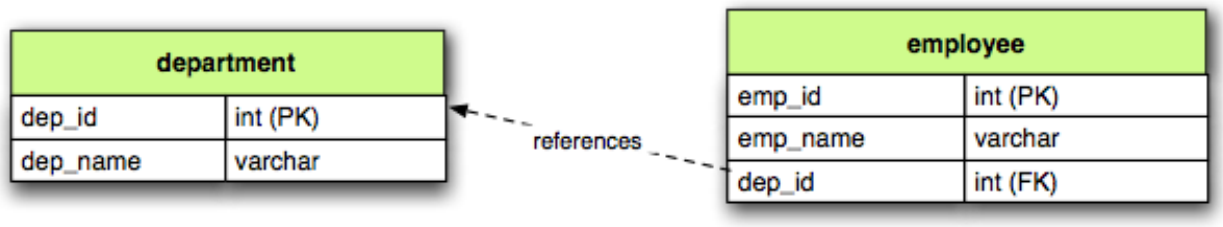
Внешние ключи могут быть составными, если входящие в них первичные ключи являются таковыми.

В примере представлена таблица «department», которая связана с таблицей «employee» через отношение столбцов «employee.dep_id» и «department.dep_id»:

Представленная на рисунке связь может быть описана через DDL следующим образом:

```
CREATE TABLE department (
  dep_id INTEGER,
```

(continues on next page)



(продолжение с предыдущей страницы)

```
  dep_name VARCHAR(30),
  PRIMARY KEY (dep_id)
)

CREATE TABLE employee (
  emp_id INTEGER,
  emp_name VARCHAR(30),
  dep_id INTEGER,
  PRIMARY KEY (emp_id),
  FOREIGN KEY (dep_id)
    REFERENCES department(dep_id)
)
```

Нормализация

Реляционная модель базируется на реляционной алгебре, одним из ключевых понятий которой является нормализация.

Основной идея нормализации в исключении повторяющихся данных так, чтобы конкретная часть данных была представлена только в одном месте. Этот подход позволяет упростить данные до максимально атомарного вида, с которым намного проще работать: искать, производить какие-либо операции.

Классический пример денормализованных данных:

Таблица 1: Employee Language

name	language	department
Dilbert	C++	Systems
Dilbert	Java	Systems
Wally	Python	Engineering
Wendy	Scala	Engineering
Wendy	Java	Engineering

Строки в этой таблице могут быть уникально идентифицированы по столбцам «name» и «language», которые являются потенциальным ключом. По теории нормализации таблица из примера нарушает вторую нормальную форму. Потому как неосновной атрибут

«department» логически связан только со столбцом «name». Правильная нормализация в данном случае выглядит следующим образом:

Таблица 2: Employee Department

name	department
Dilbert	Systems
Wally	Engineering
Wendy	Engineering

Таблица 3: Employee Language

name	language
Dilbert	C++
Dilbert	Java
Wally	Python
Wendy	Scala
Wendy	Java

Теперь наглядно видно, как вторая форма улучшила структуру данных. Изначально пример содержал повторы связок полей «name» и «department» так часто, как часто встречался уникальный для данного имени «язык». Улучшенный же вариант сделал связки «name/department» и «name/language» независимыми друг от друга.

Ограничения данных, такие как первичные и внешние ключи, предназначены как раз для достижения состояния нормализации. Для примера выше это будет выглядеть так:

- «Employee Department -> name» - первичный ключ;
- «Employee Language -> name, language» - составной первичный ключ;
- «Employee Language -> name», в свою очередь, - внешний ключ, на поле «Employee Department -> name».

Если таблицу удаётся сходу свернуть в отношения ключей, то это, зачастую, значит, что она не нормализована.

1.2.2 Язык управления данными (DML)

После того как определена схема базы данных и таблиц, в них можно помещать данные и изменять их с помощью DML, который реализован частью конструкций SQL. Далее будут подробно разобраны основные из этих конструкций.

Вставка (insert)

Новые строки добавляются с помощью команды `INSERT`. Эта команда содержит часть *VALUES*, в которой прописаны данные для каждой добавляемой строки:

```
INSERT INTO employee (emp_id, emp_name, dep_id)
    VALUES (1, 'dilbert', 1);

INSERT INTO employee (emp_id, emp_name, dep_id)
    VALUES (2, 'wally', 1);
```

Автоинкрементные целочисленные ключи

Большинство современных баз данных содержит в себе функционал для генерации инкрементных целочисленных значений, которые обычно используются в качестве искусственных первичных ключей. Как в примере с таблицами «employee» и «department». Например, при использовании *SQLite*, столбец `emp_id` в коде выше будет автоматически создан целочисленным; при использовании MySQL для создания автоинкрементных ключей используется опция `AUTO INCREMENT`; в PostgreSQL для этих целей служит тип данных `SERIAL`. Когда используются генераторы автоинкрементных первичных ключей, можно опустить эти столбцы в команде `INSERT`:

```
INSERT INTO employee (emp_name, dep_id)
    VALUES ('dilbert', 1);

INSERT INTO employee (emp_name, dep_id)
    VALUES ('wally', 1);
```

Базы данных с этой функциональностью также позволяют получить сгенерированное при вставке значение. При этом используются нестандартные для SQL конструкции и / или функции. Например, в PostgreSQL это параметр `RETURNING`:

```
INSERT INTO employee (emp_name, dep_id)
    VALUES ('dilbert', 1) RETURNING emp_id;
```

emp_id
1

Обновление (Update)

Команда `UPDATE` служит для изменения данных в существующих строках, использую параметр *WHERE* для фильтрации строк по какому-либо условию и параметр *SET* для установки нового значения в нужный столбец:

```
UPDATE employee SET dep_id=7 WHERE emp_name='dilbert'
```

Когда команда `UPDATE` выполняется по условию, как в примере выше, в результате может быть изменено любое количество строк. В том числе и ни одна. Обычно присутствует некоторый счётчик строк, который позволяет получить информацию о том, сколько строк было отфильтровано и, как следствие, изменено.

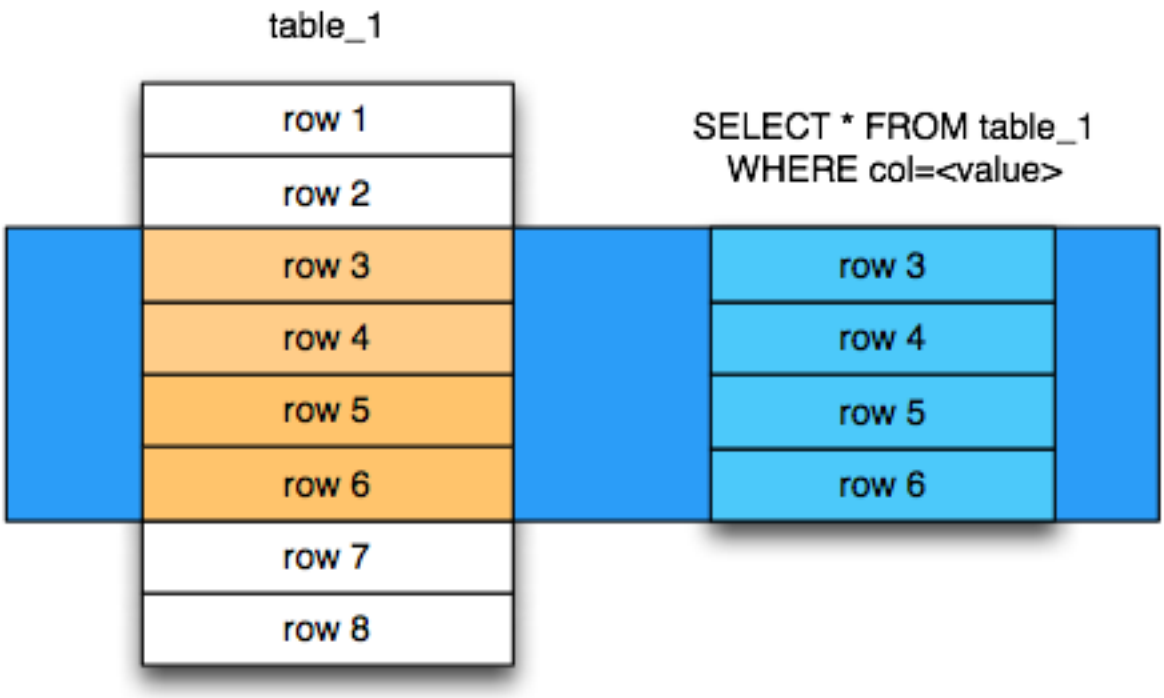
Удаление (Delete)

Команда `DELETE` служит для удаления строк. Также как и `UPDATE` использует параметр *WHERE* для выборки нужных строк:

```
DELETE FROM employee WHERE dep_id=1
```

1.2.3 Запросы (Queries)

Ключевой особенностью SQL является возможность построения запросов к данным. Для этого используется команда `SELECT`. Также как и в командах `UPDATE` и `DELETE` в ней присутствует параметр *WHERE*.



Например, можно выбрать строки у которых `dep_id` равен 12:

```
SELECT emp_id, emp_name FROM employee WHERE dep_id=12
```

Команда SELECT из примера выше имеет следующие части:

1. Параметр *FROM* указывает таблицы, из которых выбираются строки.
2. Параметр *WHERE* используется для фильтрации выбираемых строк по какому-либо условию.
3. Между словами SELECT и FROM расположен список столбцов, которые необходимо показать из каждой отфильтрованной строки.

Результат примера может выглядеть как-то так:

emp_id	emp_name
1	wally
2	dilbert
5	wendy

Сортировка

К команде SELECT можно добавить параметр *ORDER BY* задающий по какому полю сортировать результаты:

```
SELECT emp_id, emp_name FROM employee WHERE dep_id=12 ORDER BY emp_name
```

emp_id	emp_name
2	dilbert
1	wally
5	wendy

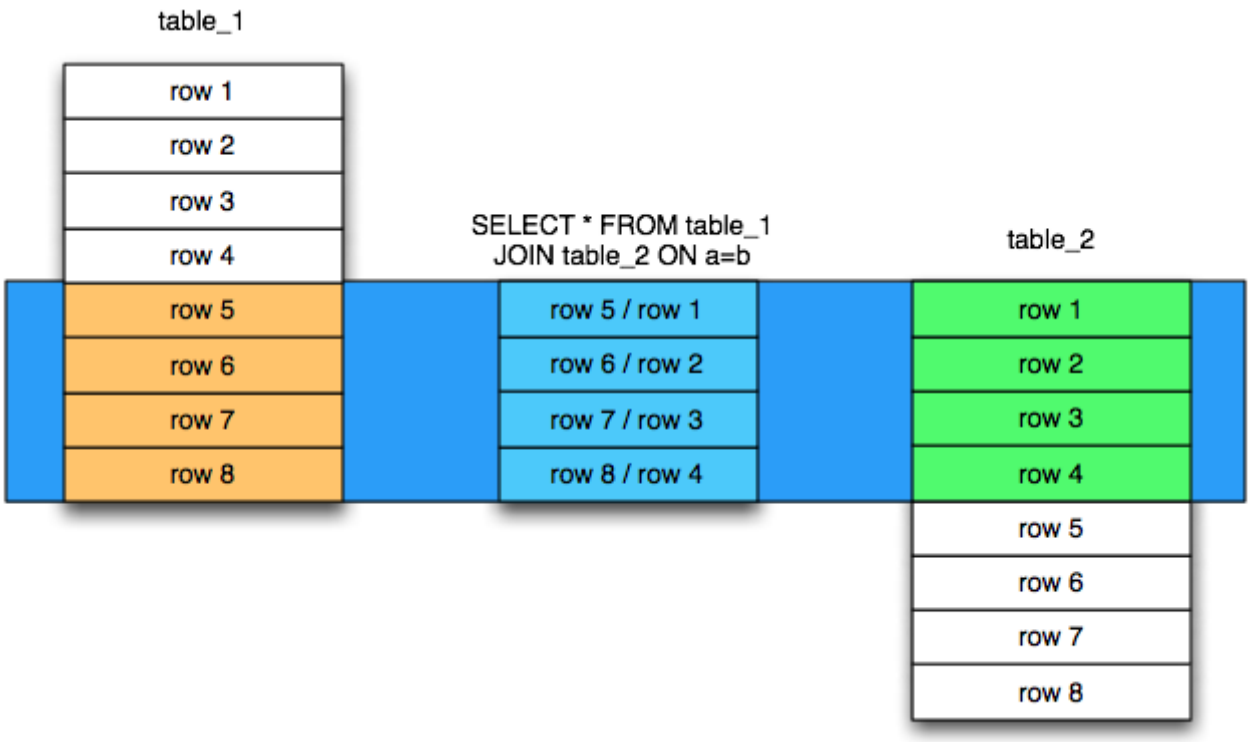
Объединения (Joins)

Запросы могут использовать механизм объединений для строк из двух таблиц и представления их как одна строка. Обычно объединение производится по *внешним ключам*.

Параметр *JOIN* помещается внутри блока *FROM*, между именами объединяемых таблиц. Он, в свою очередь, в себе содержит параметр *ON*, который отвечает за критерий объединения строк из разных таблиц.

JOIN создаёт промежуточную структуру табличного вида. Она содержит в себе объединенные данные из обоих таблиц.

Используя примеры с таблицами Department и Employee, выберем сотрудников вместе с названиями их отделов:




```
SELECT e.emp_id, e.emp_name, d.dep_name
  FROM employee AS e
    JOIN department AS d
    ON e.dep_id=d.dep_id
  WHERE d.dep_name = 'Software Artistry'
```

emp_id	emp_name	dep_name
2	dilbert	Software Artistry
1	wally	Software Artistry
5	wendy	Software Artistry

Левое внешнее объединение (Left Outer Join)

Этот вид объединения позволяет вернуть строки из «левой» части даже в том случае, если у них нет соответствия в «правой». Например, если мы хотим выбрать отделы и их сотрудников, дополнительно, получить названия отделов без сотрудников, то необходимо использовать конструкцию *LEFT OUTER JOIN*:

```
SELECT d.dep_name, e.emp_name
  FROM department AS d
  LEFT OUTER JOIN employee AS e
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  ON d.dep_id=e.dep_id
```

Допустим наша компания имеет три отдела, из которых отдел «Sales» на данный момент не имеет сотрудников. В этом случае результаты могут выглядеть следующим образом:

dep_name	emp_name
Management	dogbert
Management	boss
Software Artistry	dilbert
Software Artistry	wally
Software Artistry	wendy
Sales	<NULL>

Также существует «right outer join», который использует «правую» часть, как основную. Но использование этой конструкции считается не очень элегантным шагом.

Агрегация

Функция агрегации принимает на вход множество значений, выдавая на выходе одно. Наиболее часто используемая функция агрегации - это‘‘count()‘‘. Она получает набор строк и возвращает их количество.

В качестве параметра может использоваться любое SQL выражение. Наиболее часто используется шаблон `*`, означающий «все столбцы». В отличии от большинства функций агрегации `count()` не вычисляет значение своего аргумента, а просто считает сколько раз он был вызван:

```
SELECT count(*) FROM employee
```

count
18

Другая функция агрегации может вернуть нам среднее количество сотрудников в офисах. Для этого нам также потребуется использовать конструкцию *GROUP BY* в подзапросе:

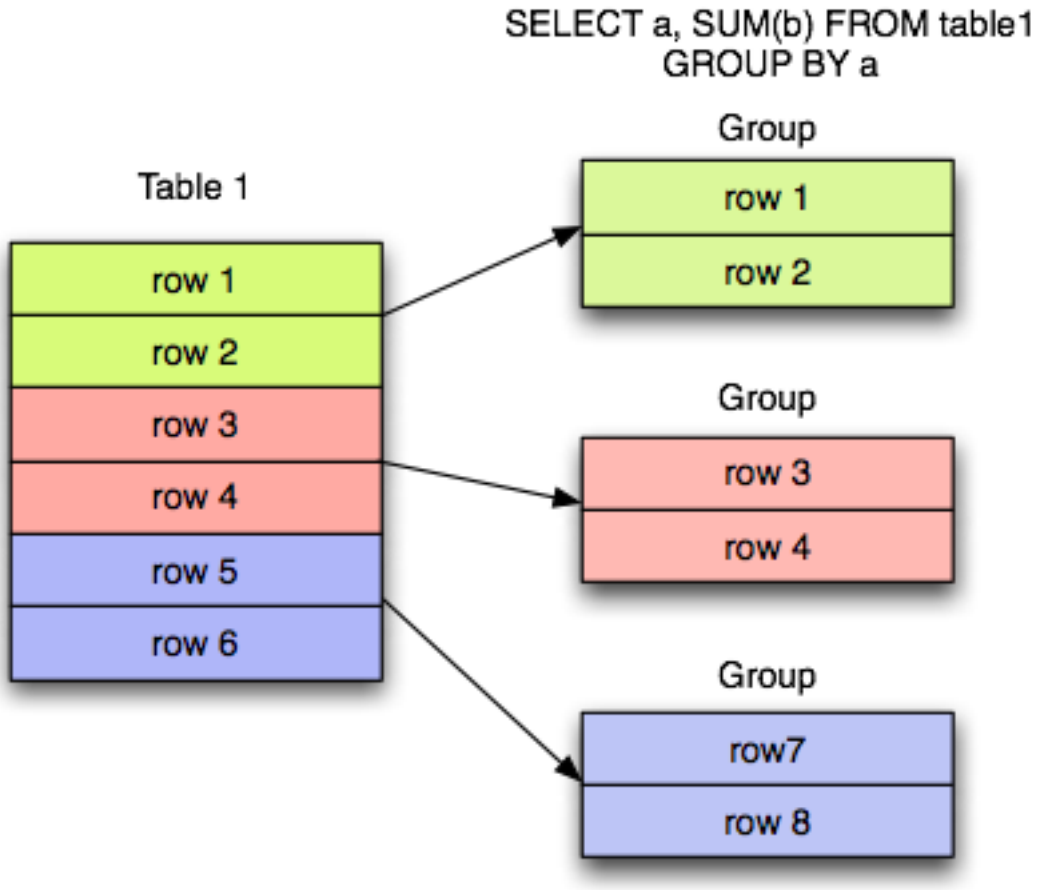
```
SELECT avg(emp_count) FROM
  (SELECT count(*) AS emp_count
    FROM employee GROUP BY dep_id) AS emp_counts
```

count
2

Примечание: Запрос в этом примере производит подсчёт только по отделам, в которых есть сотрудники. Для включения в расчёты отделы без сотрудников нужно использовать более сложный подзапрос.

Группировка

Конструкция *GROUP BY*, применяемая в выражении `SELECT`, служит для группировки результатов по какому-либо полю. Она зачастую используется совместно с агрегацией для применения агрегирующей функции к каждой из групп.



На изображении выше строки разделены на 3 подгруппы по некоему полю «a». Затем применена функция `SUM()` к полю «b» в каждой из этих групп.

В качестве примера совместного использования агрегации и конструкции *GROUP BY* рассчитаем количество сотрудников в каждом отделе:

```
SELECT count(*) FROM employee GROUP BY dep_id
```

count	dep_id
2	1
10	2
6	3
9	4

Having

Для фильтрации сгруппированных агрегированных значений применяется конструкция *HAVING*. Например, можно изменить вывод примера выше: отфильтровать отделы, в которых количество сотрудников больше семи:

```
SELECT count(*) as emp_count FROM employee GROUP BY dep_id HAVING emp_count > 7
```

count	dep_id
10	2
9	4

Выборка. Резюме

Рассмотрим, как команда SELECT ведёт себя при комбинации всех вышеописанных конструкций.

Для примера возьмём следующий набор строк:

Таблица 4: Employee

emp_id	emp_name	dep_id
1	wally	1
2	dilbert	1
3	jack	2
4	ed	3
5	wendy	1
6	dogbert	4
7	boss	3

К которому будет применён вот этот код:

```
SELECT count(emp_id) as emp_count, dep_id
  FROM employee
  WHERE dep_id=1 OR dep_id=3 OR dep_id=4
  GROUP BY dep_id
```

(continues on next page)

(продолжение с предыдущей страницы)

```
HAVING emp_count > 1
ORDER BY emp_count, dep_id
```

1. Конструкция *FROM* определяет таблицы, из которых будут выбраны строки. В нашем примере это таблица `employee`:

```
... FROM employee ...
```

emp_id	emp_name	dep_id
1	wally	1
2	dilbert	1
3	jack	2
4	ed	3
5	wendy	1
6	dogbert	4
7	boss	3

2. Каждая строка проверяется на соответствие условию, описанному в блоке *WHERE*. Только строки, прошедшие эту проверку, подвергаются дальнейшей обработке:

```
... WHERE dep_id=1 OR dep_id=3 OR dep_id=4 ...
```

emp_id	emp_name	dep_id
1	wally	1
2	dilbert	1
4	ed	3
5	wendy	1
6	dogbert	4
7	boss	3

3. Затем производится фоновая группировка по какому-либо критерия с помощью конструкции *GROUP BY*. Дальнейшая обработка производится уже над этими группами. Для текущего примера это выглядит следующим образом:

```
... GROUP BY dep_id ...
```

«group»	emp_id	emp_name	dep_id
dep_id=1	1	wally	1
	2	dilbert	1
	5	wendy	1
dep_id=3	4	ed	3
	7	boss	3
dep_id=4	6	dogbert	4

4. Затем функции агрегации применяются к каждой группе строк. В примере функция `count()` применяется к полю *emp_id*. Это значит, что для группы «1» она получит «1», «2» и «5», в качестве значений. Функции `count()` нет дела до того, какие значения в неё были переданы. И можно для упрощения передать `*`, что означает «все столбцы». Тем не менее, большинству функций агрегации важно, что именно передаётся в них в качестве параметров. Потому хорошей практикой считается определение тех столбцов, которые передаются в качестве параметров. Результат нашего примера выглядит примерно так:

```
... count(emp_id) AS emp_count ...
```

emp_count	dep_id
3	1
2	3
1	4

5. Конструкция *HAVING* работает как *WHERE*, но только для агрегированных значений. В текущем примере она осуществляет фильтрацию групп, которые имеют более одного сотрудника:

```
... HAVING emp_count > 1 ...
```

emp_count	dep_id
3	1
2	3

6. В конце применяется конструкция *ORDER BY*. Важно помнить, что реляционная математика, заложенная в основу SQL, базируется на понятии *множеств*. Которые являются по определению неупорядоченными. В обычном случае выборка, агрегация и фильтрация применяются к неупорядоченным строкам. И только в конце, перед отдачей результата пользователю, производится упорядочивание по какому-либо признаку:

```
... ORDER BY emp_count, dep_id
```

emp_count	dep_id
2	3
3	1

1.2.4 Модель ACID

Большинство реляционных баз данных реализует транзакционную модель. Акроним ACID содержит основные принципы такого подхода.

Атомарность (Atomicity)

Атомарность гарантирует, что никакая транзакция не будет зафиксирована в системе частично. Будут либо выполнены все её подоперации, либо не выполнено ни одной. Поскольку на практике невозможно одновременно и атомарно выполнить всю последовательность операций внутри транзакции, вводится понятие «отката» (rollback): если транзакцию не удаётся полностью завершить, результаты всех её до сих пор произведённых действий будут отменены и система вернётся во «внешне исходное» состояние - со стороны будет казаться, что транзакции и не было. Сигнал завершения транзакции называется «фиксация» (commit).

Согласованность (Consistency)

Транзакция достигающая своего нормального завершения (EOT - end of transaction, завершение транзакции) и, тем самым, фиксирующая свои результаты, сохраняет согласованность базы данных. Другими словами, каждая успешная транзакция по определению фиксирует только допустимые результаты. Это условие является необходимым для поддержки четвёртого свойства.

Согласованность является более широким понятием. Например, в банковской системе может существовать требование равенства суммы, списываемой с одного счёта, сумме, зачисляемой на другой. Это бизнес-правило и оно не может быть гарантировано только проверками целостности, его должны соблюсти программисты при написании кода транзакций. Если какая-либо транзакция произведёт списание, но не произведёт зачисление, то система останется в некорректном состоянии и свойство согласованности будет нарушено.

Наконец, ещё одно замечание касается того, что в ходе выполнения транзакции согласованность не требуется. В нашем примере, списание и зачисление будут, скорее всего, двумя разными подоперациями и между их выполнением внутри транзакции будет видно несогласованное состояние системы. Однако не нужно забывать, что при

выполнении требования изоляции, никаким другим транзакциям эта несогласованность не будет видна. А атомарность гарантирует, что транзакция либо будет полностью завершена, либо ни одна из операций транзакции не будет выполнена. Тем самым эта промежуточная несогласованность является скрытой.

Изолированность (Isolation)

Во время выполнения транзакции параллельные транзакции не должны оказывать влияние на её результат. Изолированность - требование дорогое, поэтому в реальных БД существуют различные уровни изоляции.

- **Read uncommitted**. Низший (нулевой) уровень изоляции. Он гарантирует только отсутствие потерянных обновлений Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определенное последней успешно выполненной транзакцией. При этом возможно считывание не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.
- **Read committed**. На этом уровне обеспечивается защита от чернового, «грязного» чтения, тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных.
- **Repeatable Read**. Уровень, при котором читающая транзакция «не видит» данные, которые были изменены но еще не зафиксированы другой транзакцией. При этом никакая другая транзакция не может изменять данные читаемые текущей транзакцией, пока та не окончена.
- **Serializable**. Самый высокий уровень изолированности; транзакции полностью изолируются друг от друга, каждая выполняется так, как будто параллельных транзакций не существует. Только на этом уровне параллельные транзакции не подвержены эффекту «фантомного чтения» (ситуация, когда при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк).

Надежность (Durability)

Независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершённой транзакцией, должны остаться сохранёнными после возвращения системы в работу. Другими словами, если пользователь получил подтверждение от системы, что транзакция выполнена, он может быть уверен, что сделанные им изменения не будут отменены из-за какого-либо сбоя.

1.3 SQLite

См.также:

- https://ru.wikipedia.org/wiki/SQLite
- https://www.sqlite.org

SQLite — компактная встраиваемая реляционная база данных. Исходный код библиотеки передан в общественное достояние. Является чисто реляционной базой данных.

Слово «встраиваемый» означает, что SQLite не использует парадигму клиент-сервер. Т.е. движок SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а предоставляет библиотеку, с которой программа компонуется и движок становится составной частью программы. Таким образом, в качестве протокола обмена используются вызовы функций (API) библиотеки SQLite. Такой подход уменьшает накладные расходы, время отклика и упрощает программу. SQLite хранит всю базу данных (включая определения, таблицы, индексы и данные) в единственном стандартном файле на том компьютере, на котором исполняется программа. Простота реализации достигается за счёт того, что перед началом исполнения транзакции записи весь файл, хранящий базу данных, блокируется; ACID[1]-функции достигаются в том числе за счёт создания файла журнала.

Несколько процессов или потоков могут одновременно без каких-либо проблем читать данные из одной базы. Запись в базу можно осуществить только в том случае, если никаких других запросов в данный момент не обслуживается; в противном случае попытка записи оканчивается неудачей, и в программу возвращается код ошибки. Другим вариантом развития событий является автоматическое повторение попыток записи в течение заданного интервала времени. Можно, также, ввести таймаут операций. Тогда подключение, столкнувшись с занятостью БД, будет ждать N секунду прежде, чем отвалиться с ошибкой SQLITE_BUSY.

Также с версии 3.7.0 присутствует режим WAL[2], с помощью которого можно использовать одну и ту же базу несколькими приложениями, как на чтение, так и на запись.

В комплекте поставки идёт также функциональная клиентская часть в виде исполняемого файла **sqlite3**, с помощью которого демонстрируется реализация функций основной библиотеки. Клиентская часть работает из командной строки, позволяет обращаться к файлу БД на основе типовых функций ОС.

Благодаря архитектуре движка возможно использовать SQLite как на встраиваемых системах, так и на выделенных машинах с гигабайтными массивами данных.

Формат файла базы данных является кросс-платформеным, что позволяет без проблем использовать одну и ту же базу на нескольких операционных системах. Также присутствует возможность хранения базы в памяти, без её записи на диск. Этот вариант

[1] Atomicity Consistency Isolation Durability
[2] Write-Ahead Logging

используется по умолчанию для консольной утилиты **sqlite3**, если не указано имя файла.

1.3.1 Поддержка стандарта SQL

Как известно, в своем развитии SQL устремился в разные стороны. Крупные производители начали впихивать всякие расширения. И хотя принимаются всякие стандарты, в реальной жизни все крупные БД не поддерживают стандартов полностью. Но зато имеют что-то свое.

Так вот, SQLite старается жить по принципу «минимальный, но полный набор». Она не поддерживает сложные штуки, но во многом соответствует SQL 92. И вводит некие свои особенности, которые очень удобны, но — не стандартны.

Неподдерживаются следующие возможности:

- **RIGHT** и **FULL OUTER JOIN**. Реализован только **LEFT OUTER JOIN**.
- Частично реализован **ALTER TABLE**. Доступны только **RENAME TABLE** и **ADD COLUMN**.
- Частичная поддержка тригеров. Доступены только **FOR EACH ROW** тригеры.
- Запись во **VIEWS**. В SQLite **VIEWS** доступны только на чтение. Частично обходится через тригеры.
- В силу реализации базы данных, как единственного файла и отхода от концепции «клиент-сервер», не используются возможности **GRANT** и **REVOKE**.
- По умолчанию отключены внешние ключи. Это сделано для обратной совместимости.

1.3.2 Типизация

SQLite использует динамическое типизирование данных. Это значит, что тип столбца не определяет тип хранимого значения в этом поле записи. Т.е. в любой столбец можно занести любое значение.

Тип столбца определяет как сравнивать значения (нужно же их привести к единому типу при сравнении, скажем, внутри индекса). Но не обязывает заносить значения именно такого типа в столбец.

Допустим, мы объявили столбец как «A INTEGER». SQLite позволяет занести в этот столбец значения любого типа (999, «abc», «123», 678.525). Если вставляемое значение — не целое, то SQLite пытается привести его к целому. Т.е. строка «123» превратится в целое 123, а остальные значения запишутся «как есть».

Возможные типы полей: NULL, INTEGER, REAL, TEXT, BLOB.

1.3.3 Надёжность

Ситуация с покрытиями тестами исходного кода SQLite в некотором роде является легендой.

Это связно с тем, что тестами описано практически все возможные (и невозможные) ситуации. Кода, описывающего тесты, намного больше, чем кода реализующего SQLite. Естественно, что покрытие кода тестами 100%.

Сам же подход к тестированию использующийся разработчиками SQLite достоин подражания. И может быть взят как идеал, к которому следуют стремиться.

1.3.4 Практика

В качестве практического занятия для закрепления понимания реляционной модели и иллюстрации её ограниченности, мы рассмотрим пример создания структуры каталога товаров.

Для того чтобы запустить SQL-код необходимо выполнить следующую команду:

```
sqlite3 ~/example.sqlite
```

Далее откроется интерактивный SQLite-shell, куда можно вводить команды.

Создание структуры каталога

В этом примере мы рассмотрим как создать простую струкртуру каталога товаров.

```
1  CREATE TABLE category (name TEXT NOT NULL);
2
3  INSERT INTO category (name) VALUES
4      ('Тапки'),
5      ('Самолёты'),
6      ('Ноутбуки');
7
8
9  CREATE TABLE product (
10     name TEXT NOT NULL,
11     price NUMERIC NOT NULL,
12     category REFERENCES category(rowid)
13 );
14
15 INSERT INTO product (name, price, category) VALUES
16     ('Босоножки', 1.17, 1),
17     ('Вьетнамки', 2.36, 1),
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    ('Макасины', 4.99, 1),
    ('ИЛ-2', 556000, 2),
    ('Суперджет 100', 1500000, 2),
    ('Ту-160', 25000000, 2),
    ('Dell', 590, 3),
    ('Lenovo', 200, 3),
    ('Sony', 437, 3);
```

Расширение структуры каталога

В этом примере мы рассмотрим, как расширить существующую структуру под новые требования.

Требования будут следующими:

- Для категории «Тапки» необходимо добавить поле «Размер».
- Для категории «Самолёты» необходимо добавить поле «Вместимость».
- Для категории «Ноутбуки» необходимо добавить поле «Прцессор».

```
ALTER TABLE product ADD size INTEGER;

ALTER TABLE product ADD capacity INTEGER;

ALTER TABLE product ADD processor TEXT;
```

Сущетсвенным недостатком данного подхода является тот факт, что поля одних категорий товаров являются обязательными для других категорий. Т.е., например, поле «размер», которое добавлено для категории «Тапки», становится обязательным и для товаров из категории «Самолёты» и «Ноутбуки».

Классическая реляционная модель не позволяет удобно применить отдельные поля для разных категорий. Можно, конечно, использовать поле типа BLOB и складывать туда всё, что душе вздумается, но тогда потеряются такие прелести реляционных СУБД, как индексы, тригеры (либо они очень усложнятся) и фильтрация в запросах.

Можно ещё, как вариант, оставить базовые поля в одной таблице, а остальные вынести в дополнительные, связав их внешними ключами. Но это, опять же, ненужное усложнение.

Поэтому предлагаю оставить замечательную реляционную базу SQLite в покое. Она замечательно справляется с возложенными на неё обязанностями. Благодаря чему и является, пожалуй, самой популярной встраиваемой базой данных в мире.

1.4 ZODB

См.также:

- http://www.zodb.org/

ZODB - объектно-ориентированная база данных для Python-объектов.

```
import persistent

class Account(persistent.Persistent):

  def __init__(self):
      self.balance = 0.0

  def deposit(self, amount):
      self.balance += amount

  def cash(self, amount):
      assert amount < self.balance
      self.balance -= amount
```

ZODB используется в Zope, Plone, Zenoss, ERP5 и некоторых других системах. Например, ZODB и ZEO (без Zope) используются в системе Indico — программном обеспечении для организации симпозиумов, конференций, лекций и т. п., разработанном и используемом ЦЕРНом. Примером отечественного проекта с использованием этой базы данных является ранняя версия программы электронного документооборота «NauDoc» от компании Naumen.

На данный момент реализована полностью на Python[1]. Но в зависимостях используется BTrees, который реализован с использованием языка C.

1.4.1 Конкурентный доступ

Изначально является базой данных доступной только для одного процесса. Но данное ограничение обходится с помощью ZEO (Zope Enterprise Objects). Данная технология подменяет стандартное файловое хранилище ZODB на клиент-серверное. «Клиент» не записывает данные на диск, а передаёт их для этого на «сервер», который уже и реализует логику записи. Клиентов может быть несколько, благодаря чему и достигается возможность одновременной работы с одним хранилищем.

Существует ещё один способ конкурентного доступа к данным в ZODB: реализация хранилища в реляционных базах данных[2]. При таком способе множественный доступ

[1] https://github.com/zopefoundation/ZODB
[2] https://github.com/zodb/relstorage

реализуется самой РСУБД стандартными для неё методами через управление подключениями.

1.4.2 Особенности

ZODB является иерархической базой данных. Корневой элемент создаётся при старте базы данных. Структура представляет собой словарь сериализованных (pickle) объектов языка Python. Для записи объекта в хранилище достаточно, чтобы он мог быть сериализован методом стандартной библиотеки Python'а.

Для оптимизации хранения древовидных структур применяется встроенный модуль OOBTree реализованный на языке C. Но для его эффективной работы необходимо ручное поддержание структуры дерева.

Полностью поддерживается механизм транзакций и ACID[3]. Также присутствует поддержка точек сохранения. Это когда в кэш сохраняется часть большой транзакции для освобождения ресурсов. Присутствует сохранение истории изменений и их отмена.

Для хранения Blob-данных (таких как изображения, например) используется специальный тип хранилища. В нём отсутствует реализация механизмов версионирования данных. За счёт чего эти данные не оказывают критического влияния на быстродействие системы хранения.

Для востребованных данных существует кэширование в памяти. Он динамически изменяется в зависимости от нужды в объектах: неиспользуемые объекты автоматически удаляются.

Для объектов в базе применяется версионирование, что может значительным образом увеличить размер хранилища. Для сокращения объёма хранилища и удаления неиспользуемых версий объектов используется метод packing. Его рекомендуется применять по расписанию, т.к. удалённые объекты также остаются в хранилище. Это аналог команды VACUUM из PostgreSQL.

1.4.3 Заключение

В заключение хочется сказать, что аналоги ZODB присутствуют во многих языках программирования. Например, в Ruby это PStore, в Java - ObjectDB.

Также необходимо упомянуть главные недостатки ZODB:

- Отсутствие встроенных механизмов сложных индексов. В ZODB нет таких мощных механизмов создания индексов как в реляционных СУБД. Она использует перебор словаря для нахождения нужных значений. Что накладывает серьёзные ограничения на реализацию поиска по сложным критериям.

[3] Atomicity Consistency Isolation Durability

- Отсутствие удобного доступа к данным из приложений реализованных не на Python. В случае реализации какого-либо сложного проекта на базе ZODB, придётся реализовывать API для доступа к данным, хранящимся в ней, для тех программ, которые выполнены не на Python. Что не всегда возможно.

Но для тех областей, где не так важны индексы, а, наоборот, мешают ограничения реляционного подхода ZODB может оказаться удачным вариантом.

Примером такого проекта может являться задача учёта структуры сети интернет-провайдера.

1.4.4 Практика

Для сегодняшней практической части мы возьмём уже знакомую нам задачу о каталоге товаров.

Настройка окружения

Установим необходимые зависимости для проекта.

```
pip install zodb
```

Реализация задачи

Для начала создадим классы для реализации задачи без сохранения.

1.5 Redis

См.также:

- https://ru.wikipedia.org/wiki/Redis
- http://eax.me/redis/
- http://redis.io/

Redis (REmote DIctionary Server) - это нереляционная высокопроизводительная СУБД.

Redis работает на большинстве POSIX систем. Официальной поддержки для сборок Windows нет.

1.5.1 Особенности

Redis умеет сохранять данные на диск. Можно настроить Redis так, чтобы данные вообще не сохранялись, сохранялись периодически по принципу copy-on-write, или сохранялись периодически и писались в журнал (binlog). Таким образом, всегда можно добиться требуемого баланса между производительностью и надежностью.

В основе своей использует записи вида **ключ-значение**. Ключи бинарнобезопасны. Это значит, что в качесте ключа может быть использована любая бинарная последовательность, полученная хоть из сроки, хоть из JPG-картинки. Максимальный размер ключа - 512 MB.

В качестве значений поддерживаются следующие структуры данных:

- Строки (strings). Это те же самые ключи, но сохранённые как значения.
- Списки (lists). Классические списки строк, упорядоченные в порядке вставки, которая возможна как со стороны головы, так и со стороны хвоста списка.
- Множества (sets). Множества строк в математическом понимании: не упорядочены, поддерживают операции вставки, проверки вхождения элемента, пересечения и разницы множеств.
- Упорядоченные множества (sorted sets). Упорядоченное множество отличается от обычного тем, что его элементы упорядочены по особому параметру «score». Позволяет выбирать группы значений из множества. Например, 10 сверху. Присутствует возможность лексических упорядоченных множеств строк, у которых поле score одинаковое.
- Хеш-таблицы (hashes). Классические хеш-таблицы или ассоциативные массивы.
- Битовые массивы (bitmaps).
- HyperLogLog. Структура данных для реализации алгоритма случайного вероятностного подсчета количества уникальных значений.

Для всех этих типов поддерживаются атомарные операции (например вставка в список или пересечение множеств).

Позволяет хранить не только строки, но и массивы (которые могут использоваться в качестве очередей или стеков), словари, множества без повторов, , а также множества, отсортированные по некой величине. Разумеется, можно работать с отдельными элементами списков, словарей и множеств. Присутствует возможность указать время жизни данных (двумя способами - «удалить тогда-то» и «удалить через . . . »).

Интересная особенность Redis заключается в том, что это - однопоточный сервер. Такое решение сильно упрощает поддержку кода, обеспечивает атомарность операций и позволяет запустить по одному процессу Redis на каждое ядро процессора. Разумеется, каждый процесс будет прослушивать свой порт.

В Redis есть репликация. Репликация с несколькими главными серверами не поддерживается. Каждый подчиненный сервер может выступать в роли главного для других. Репликация в Redis не приводит к блокировкам ни на главном сервере, ни на подчиненных. На репликах разрешена операция записи. Когда главный и подчиненный сервер восстанавливают соединение после разрыва, происходит полная синхронизация (resync).

Также Redis поддерживает транзакции (будут последовательно выполнены либо все операции, либо ни одной) и пакетную обработку команд (выполняем пачку команд, затем получаем пачку результатов). Притом ничто не мешает использовать их совместно.

С версии 2.6.0 добавлена поддержка Lua, позволяющего выполнять запросы на сервере. Lua позволяет атомарно совершить произвольную обработку данных на сервере и предназначена для использования в случае, когда нельзя достичь того же результата с использованием стандартных команд.

Еще одна особенность Redis - поддержка механизма publish/subscribe. С его помощью приложения могут создавать каналы, подписываться на них и помещать в каналы сообщения, которые будут получены всеми подписчиками. Что-то вроде IRC-чата.

С версии 3.0 появилась встроенная поддержка кластеризации.

1.5.2 Заключение

Redis благодаря его архитектуре очень часто используется как хранилище сессий пользователей, построения очередей, а также для кеша.

1.5.3 Практика

В качестве практики рассмотрим примеры работы с разными структурами данных в Redis.

Настройка окружения

Установим необходимые пакеты:

```
wget http://download.redis.io/redis-stable.tar.gz
tar xzf redis-stable.tar.gz
cd redis-stable
make
```

Запустим сервер:

```
src/redis-server  --loglevel=DEBUG
```

Подключимся с помощью клиента к запущенному серверу:

```
src/redis-cli
```

Строки

Установка / получение значения:

```
> set mykey somevalue
OK
> get mykey
"somevalue"
```

Через команду `SET` можно выполнять условную установку значения для ключа (например, присвоить значение только в том случае, если ключ не / существует), а также задать время «жизни» ключа.

Для целочисленных значений присутствует возможность их де / инкремента через встроенные команды:

```
> set counter 100
OK
> incr counter
(integer) 101
> incrby counter 50
(integer) 151
> decr counter
(integer) 150
> decrby counter 50
(integer) 100
```

Операции де / инкремента атомарны. Т.е. при чтении двумя клиентов некоторого ключа со значением 10 и последующего его инкремента, финальное значение этого ключа будет 12.

Для присвоения / получения нескольких значений сразу служат команды `MSET` и `MGET` соответственно:

```
> mset a 10 b 20 c 30
OK
> mget a b c
1) "10"
2) "20"
3) "30"
```

Списки

Для построения списка необходимо добавить в него элементы. Добавить можно как с головы списка (слева), так и с конца (справа):

```
> lpush mylist A first
(integer) 2
> rpush mylist B
(integer) 3
> lrange mylist 0 -1
1) "first"
2) "A"
3) "B"
```

Для получения крайних элементов слева / справа служат соответствующие команды:

```
> lpop mylist
"first"
> rpop mylist
"B"
```

Существуют блокирующие реализации этих команды `BLPOP` и `BRPOP`. Они вернут результат выполнения только после того как будет добавлен элемент или по истечении времени ожидания.

Хеш-таблицы

Могут служить для хранения каких-либо объектов:

```
> hmset user:1000 username antirez birthyear 1977 verified 1
OK
> hget user:1000 username
"antirez"
> hget user:1000 birthyear
"1977"
> hgetall user:1000
1) "username"
2) "antirez"
3) "birthyear"
4) "1977"
5) "verified"
6) "1"
```

Аналогично команде HMSET, которая позволяет записать сразу несколько значений в хеш-таблицу, существует команда HMGET, позволяющая получить массив значений из желаемых полей:

```
> hmget user:1000 username birthyear no-such-field
1) "antirez"
2) "1977"
3) (nil)
```

Присутствует возможность инкремента целочисленных значений в хеш-таблице:

```
> hincrby user:1000 birthyear 10
(integer) 1987
```

Множества

Это классические математические множества:

```
> sadd myset 1 2 3 2
(integer) 3
> smembers myset
1. 3
2. 1
3. 2
```

Есть возможность проверки множества на наличие в нём элемента:

```
> sismember myset 3
(integer) 1
> sismember myset 30
(integer) 0
```

Для данных этого типа доступны операции над множествами: пересечение, объединение, разность и т.п. Для получения элемента служит команда SPOP.

Упорядоченные множества

Упорядоченные множества отличаются от классических наличием специального поля (score) для сортировки элементов. Для добавления элементов в упорядоченное множество служит команда ZADD, первым аргументом которой передаётся значение для поля score:

```
> zadd hackers 1940 "Alan Kay"
(integer) 1
> zadd hackers 1957 "Sophie Wilson"
(integer 1)
> zadd hackers 1953 "Richard Stallman"
(integer) 1
```

(continues on next page)

(продолжение с предыдущей страницы)

```
> zadd hackers 1949 "Anita Borg"
(integer) 1
> zadd hackers 1965 "Yukihiro Matsumoto"
(integer) 1
> zadd hackers 1914 "Hedy Lamarr"
(integer) 1
> zadd hackers 1916 "Claude Shannon"
(integer) 1
> zadd hackers 1969 "Linus Torvalds"
(integer) 1
> zadd hackers 1912 "Alan Turing"
(integer) 1
> zrange hackers 0 -1 withscores
1) "Alan Turing"
2) "1912"
3) "Hedy Lamarr"
4) "1914"
5) "Claude Shannon"
6) "1916"
7) "Alan Kay"
8) "1940"
9) "Anita Borg"
10) "1949"
11) "Richard Stallman"
12) "1953"
13) "Sophie Wilson"
14) "1957"
15) "Yukihiro Matsumoto"
16) "1965"
17) "Linus Torvalds"
18) "1969"
```

Присутствуют возможности выборки определённых наборов элементов из упорядоченного множества по полю score (команда ZRANGEBYSCORE) и определение позиции элемента в множестве (команда ZRANK):

```
> zrangebyscore hackers -inf 1950
1) "Alan Turing"
2) "Hedy Lamarr"
3) "Claude Shannon"
4) "Alan Kay"
5) "Anita Borg"
> zrank hackers "Anita Borg"
(integer) 4
```

В Redis имеется возможность лексической (по алфавиту) сортировки упорядоченных

множеств строковых значений:

```
> zadd hackers 0 "Alan Kay" 0 "Sophie Wilson" 0 "Richard Stallman" 0 "Anita Borg"␣
↪0 "Yukihiro Matsumoto" 0 "Hedy Lamarr" 0 "Claude Shannon" 0 "Linus Torvalds" 0
↪"Alan Turing"
(integer) 9
> zrange hackers 0 -1
1) "Alan Kay"
2) "Alan Turing"
3) "Anita Borg"
4) "Claude Shannon"
5) "Hedy Lamarr"
6) "Linus Torvalds"
7) "Richard Stallman"
8) "Sophie Wilson"
9) "Yukihiro Matsumoto"
```

Для таких множеств также присутствует возможность выборки наборов элементов:

```
> zrangebylex hackers [B [P
1) "Claude Shannon"
2) "Hedy Lamarr"
3) "Linus Torvalds"
```

Битовые массивы и HyperLogLog

Эти две структуры данных являются очень специфичными. Поэтому отдельного их описания здесь не будет. При необходимости особенности их применения легко можно узнать из официальной документации.

1.6 PostgreSQL

См.также:

- RU документация
- https://ru.wikipedia.org/wiki/PostgreSQL
- Восхитительная книга «Работа с PostgreSQL»
- Книга рецептов для СУБД PostgreSQL

PostgreSQL – свободная объектно-реляционная система управления базами данных (СУБД).

1.6.1 Поддержка стандартов, возможности, особенности

PostgreSQL базируется на языке SQL и поддерживает многие из возможностей стандарта SQL:2011.

На данный момент PostgreSQL имеются следующие ограничения:

- Максимальный размер базы данных: нет ограничений
- Максимальный размер таблицы: 32 Тбайт
- Максимальный размер записи: 1,6 Тбайт
- Максимальный размер поля: 1 Гбайт
- Максимум записей в таблице: нет ограничений
- Максимум полей в записи: 250 - 1600, в зависимости от типов полей
- Максимум индексов в таблице: нет ограничений

Важнейшие ТТХ PostgreSQL

- Надежность и устойчивость. Надежность PostgreSQL является известным фактом, доказанным на примере многих проектов, в которых PostgreSQL работает без сбоев под высокими нагрузками на протяжении нескольких лет. * Превосходная поддержка. Сообщество PostgreSQL предоставляет квалифицированную и быструю помощь. Коммерческие компании предлагают свои услуги по всему миру.
- Конкурентная работа при большой нагрузке. PostgreSQL использует многоверсионность (MVCC) для обеспечения надежной и быстрой работы в конкурентных условиях под высокой нагрузкой. * Масштабируемость. PostgreSQL отлично использует современную архитектуру многоядерных процессоров – его производительность растет линейно вплоть до 64-х ядер. Кластерные решения на основе Postgres-XC, Postgres-XL помогают с горизонтальной масштабируемостью.
- Кроссплатформенность. PostgreSQL работает под всеми видами UNIX-подобных систем, включая Linux, FreeBSD, Solaris, HP/UX, Mac OS X, а также под MS Windows. * Расширяемость. Доступны исходные коды PostgreSQL, что делает возможным добавление новой функциональности для вашего проекта без дополнительных проблем. Расширяемость PostgreSQL позволяет создавать новые типы данных и методы доступа.
- Доступность. PostgreSQL распространяется под лицензией, близкой к BSD, которая не накладывает никаких ограничений на коммерческое использование и не требует лицензионных выплат.

1.6.2 Основные возможности

- Высокий уровень соответствия ANSI SQL 92, ANSI SQL 99 и ANSI SQL 2003, 2011.

- Интерфейсы для Tcl, Perl, C, C++, PHP, JSON, ODBC, JDBC, Embedded SQL in C, Python, Ruby, Java и других. * Интеграция защиты данных с операционной системой (SE-Linux).
- Представления, последовательности, наследование, outer joins, подзапросы, ссылочная целостность, оконные функции, CTE (рекурсивные запросы).
- Пользовательские функции, хранимые процедуры, триггеры.
- Процедурные языки PL/PgSQL, PL/Perl, PL/Python, PL/Java и другие.
- Расширяемый набор типов данных с поддержкой индексов (GiST, GIN, SP-GiST).
- Встроенная система полнотекстового поиска с поддержкой всех европейских языков.
- Встроенная поддержка слабоструктурированных данных (xml, json, jsonb) с поддержкой индексов.
- Горячее резервирование и репликация (синхронная, асинхронная, каскадная), PITR, двунаправленная (BDR).
- Полная поддержка ACID, уровни изоляции, эффективная сериализация транзакций.
- Функциональные и частичные индексы.
- Интернационализация, поддержка Unicode и locale.
- Загружаемые расширения, например, PostGIS, hstore.
- Поддержка SSL и Kerberos-аутентификации.
- Foreign Data Wrappers (writable), поддержка всех основных баз данных.

Функции

Функции являются блоками кода, исполняемыми на сервере, а не на клиенте БД. Хотя они могут писаться на чистом SQL, реализация дополнительной логики, например, условных переходов и циклов, выходит за рамки собственно SQL и требует использования некоторых языковых расширений. Функции могут писаться с использованием одного из следующих языков:

- Встроенный процедурный язык PL/pgSQL, во многом аналогичный языку PL/SQL, используемому в СУБД Oracle;
- Скриптовые языки – PL/Lua, PL/LOLCODE, PL/Perl, PL/PHP, PL/Python, PL/Ruby, PL/sh, PL/Tcl и PL/Scheme;
- Классические языки – C, C++, Java (через модуль PL/Java);
- Статистический язык R (через модуль PL/R).

PostgreSQL допускает использование функций, возвращающих набор записей, который далее можно использовать так же, как и результат выполнения обычного запроса.

Функции могут выполняться как с правами их создателя, так и с правами текущего пользователя.

Иногда функции отождествляются с хранимыми процедурами, однако между этими понятиями есть различие. С девятой версии возможно написание автономных блоков, которые позволяют выполнять код на процедурных языках без написания функций, непосредственно в клиенте.

Триггеры

Триггеры определяются как функции, инициируемые DML-операциями. Например, операция INSERT может запускать триггер, проверяющий добавленную запись на соответствия определённым условиям. При написании функций для триггеров могут использоваться различные языки программирования.

Триггеры ассоциируются с таблицами. Множественные триггеры выполняются в алфавитном порядке.

Правила и представления

Механизм правил (rules) представляет собой механизм создания пользовательских обработчиков не только DML-операций, но и операции выборки. Основное отличие от механизма триггеров заключается в том, что правила срабатывают на этапе разбора запроса, до выбора оптимального плана выполнения и самого процесса выполнения. Правила позволяют переопределять поведение системы при выполнении SQL-операции к таблице.

Хорошим примером является реализация механизма представлений (views). При создании представления создается правило, которое определяет, что вместо выполнения операции выборки к представлению система должна выполнять операцию выборки к базовой таблице/таблицам с учетом условий выборки, лежащих в основе определения представления. Для создания представлений, поддерживающих операции обновления, правила для операций вставки, изменения и удаления строк должны быть определены пользователем.

Индексы

В PostgreSQL имеется поддержка индексов следующих типов: B-дерево, хэш, R-дерево, GiST, GIN. При необходимости можно создавать новые типы индексов, хотя это далеко не тривиальный процесс.

Индексы в PostgreSQL обладают следующими свойствами:

- возможен просмотр индекса не только в прямом, но и в обратном порядке – создание отдельного индекса для работы конструкции `ORDER BY ... DESC` не нужно;
- возможно создание индекса над несколькими столбцами таблицы, в том числе над столбцами различных типов данных;
- индексы могут быть функциональными, т.е. строиться не на базе набора значений некоего столбца(ов), а на базе набора значений функции от набора значений;
- индексы могут быть частичными, то есть строиться только по части таблицы (по некоторой её проекции); в некоторых случаях это помогает создавать намного более компактные индексы или достигать улучшения производительности за счёт использования разных типов индексов для разных (например, с точки зрения частоты обновления) частей таблицы;
- планировщик запросов может использовать несколько индексов одновременно для выполнения сложных запросов.

Многоверсионность (MVCC)

PostgreSQL поддерживает одновременную модификацию БД несколькими пользователями с помощью механизма Multiversion Concurrency Control (MVCC). Благодаря этому соблюдаются требования ACID, и практически отпадает нужда в блокировках чтения.

Типы данных

PostgreSQL поддерживает большой набор встроенных типов данных:

- Численные типы
  - Целые
  - С фиксированной точкой
  - С плавающей точкой
  - Денежный тип (отличается специальным форматом вывода, а в остальном аналогичен числам с фиксированной точкой с двумя знаками после запятой)
- Символьные типы произвольной длины
- Двоичные типы (включая BLOB)
- Типы «дата/время» (полностью поддерживающие различные форматы, точность, форматы вывода, включая последние изменения в часовых поясах)
- Булев тип
- Перечисление
- Геометрические примитивы

- Сетевые типы
  - IP и IPv6-адреса
  - CIDR-формат
  - MAC-адрес
- UUID-идентификатор
- XML-данные
- Массивы
- JSON
- Идентификаторы объектов БД
- Псевдотипы
- Типы для текстового поиска
- Диапазонные типы

Более того, пользователь может самостоятельно создавать новые требуемые ему типы и программировать для них механизмы индексирования с помощью GiST.

Пользовательские объекты

PostgreSQL может быть расширен пользователем для собственных нужд практически в любом аспекте. Есть возможность добавлять собственные:

- Преобразования типов
- Типы данных
- Домены (пользовательские типы с изначально наложенными ограничениями)
- Функции (включая агрегатные)
- Индексы
- Операторы (включая переопределение уже существующих)
- Процедурные языки

Наследование

Таблицы могут наследовать характеристики и наборы полей от других таблиц (родительских). При этом данные, добавленные в порождённую таблицу, автоматически будут участвовать (если это не указано отдельно) в запросах к родительской таблице.

1.6.3 Коммерческие расширения

На базе PostgreSQL компанией EnterpriseDB созданы более мощные варианты этой СУБД, являющиеся платными для коммерческого использования – Postgres Plus (состоит целиком только из продуктов с открытыми исходными кодами; плата требуется только при необходимости приобретения коммерческой поддержки продукта) и Postgres Plus Advanced Server (расширение PostgreSQL специальными возможностями для обеспечения совместимости с Oracle Database). В комплекте поставки данных продуктов содержится большой набор ПО для разработчиков и DBA:

- Postgres Studio – более мощный аналог pgAdmin;
- Postgres Plus Debugger – отладчик для кода на PL/pgSQL, интегрированный с предыдущим пакетом;
- Migration Studio – инструмент для автоматического преобразования баз данных из MySQL / Oracle в PostgreSQL

1.6.4 Заключение

PostgreSQL на данный момент является одной из самых (если не самой-самой) перспективных СУБД в мире. Благодаря великолепной архитектуре, бесплатности, отличному сообществу и огромнейшим возможностям.

Обычно PostgreSQL сравнивают с MySQL. Но на данный момент PostgreSQL далеко обходит по возможностям MySQL. И, благодаря своим возможностям по аналитике данных и манипулированию ими, зачастую может конкурировать с признанными лидерами рынка enterprise-СУБД: Oracle и MS SQL.

А благодаря своим возможностям в области хранения нереляционных данных (JSON, text search types, HStore) PostgreSQL напрямую конкурирует с NoSQL-решениями (например, MongoDB).

1.6.5 Практика

Настройка окружения