

进阶应用

1. 模糊查询

因为人们对于查询一个东西的方式各种各样，所以我们需要一种操作将这些标准化规范化，且易于准确查询到指定的内容

Title to search:

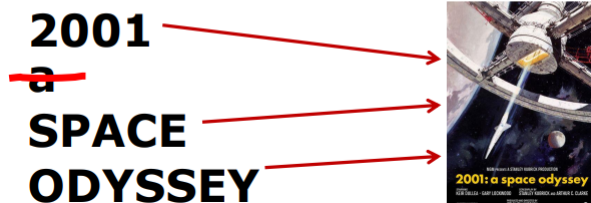
2001: A Space Odyssey

人们使用的是所谓的“全文搜索”，这里是它工作原理的简化版本

文本分割成(纯)单词，删除太常见的单词，然后将每个单词与影片标识符关联起来

FULL-TEXT SEARCH

2001: a space odyssey



这些词存储在一个特殊的表中

需要注意的是，一个真正的全文搜索引擎能够识别出单词的单数和复数，以及不定式、前分词和过去分词

当人们输入一个标题时，我们执行与之前相同的操作，隔离和标准化看起来重要的单词，然后，我们将在前一个表中查找与这些单词相关的MOVIEID值

问题是，这几个词可能会分别出现在几个标题中

- 我们计算通过电影匹配关键词数量，然后再排名



或者我们只搜索一个词，而这一个词产生了多个匹配结果

- 可以采取不同的策略（从找到的标题中删除输入的词，然后比较匹配次数）
- 我们甚至可以用常见的拼写错误和拼写错误扩展我们的搜索关键词

Title to search:



2. 事务

介绍

在一次数据库操作中，如果出现意外（如不满足条件、系统崩溃等），一次数据库的修改存在问题，那么我们需要事务对其进行维护

事务中有几个步骤，要么全部完成，要么什么都不做，这些操作只能必须成功或失败

经典的数据库示例是将资金从活期账户转到储蓄账户。您需要更改两行中的余额，如果系统在中间崩溃会发生什么？

Account type	Account number	Balance
CURRENT ACNT	1234567	300.00
SAVINGS ACNT	8765432	1600.00

Single Unit

事务过程

启动事务

- BEGIN / START
- 其他产品（如 Oracle 或 DB2）在开始修改数据时自动启动事务（如果目前不在事务中）

完成事务

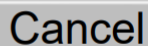
- COMMIT 事务结束，执行事务内修改的全部内容

commit



- ROLLBACK 回滚，自动撤销所有操作

rollback



事务模板

```
1 BEGIN TRANSACTION
2
3 ... 执行操作
4
5 COMMIT
```

注意

并发性

- 数据库通常意味着在许多用户之间共享数据
当您更改数据时，以及在整个事务期间，DBMS 阻止其他用户更改相同的数据
- 每一次提交都定义了一个一致的数据状态，这是当时数据的“官方”状态。回滚将把你带回已知的最后一个一致状态。这一切都是突飞猛进的，每一笔交易都是一次飞跃

自动提交

- 许多产品（MySQL等）在自动提交模式下启动，这意味着每个更改都是自动提交的，并且不能撤销，除非运行反向操作(并不总是容易的)
- 一些接口（JDBC Java数据库连接）总是以自动提交模式启动，即使在访问像 Oracle 这样从未在这种模式下本机工作的产品时也是如此

- 对于像 Oracle 和 MySQL，任何对数据库结构的更改（**DDL**操作）都会自动提交所有未决的更改到数据
- SQL Server 和 PostgreSQL 可以将 DDL 语句写在事务里面

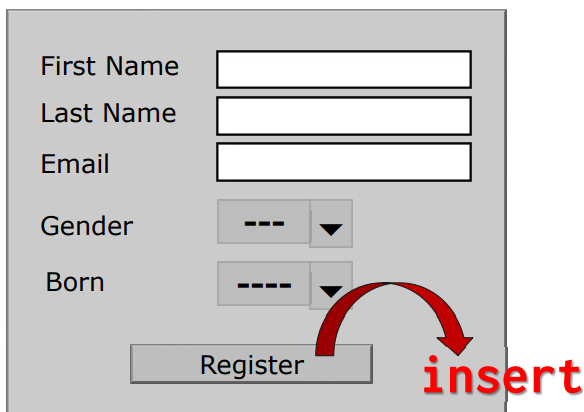
虚拟删除

- 删除通常是虚拟删除
- 事实上，它们是 **flag** 的更新，数据只是被隐藏了起来，一个很好的理由是外键，它可以防止物理删除一个网络商店不能从物理上删除一个不再销售但出现在许多过去订单中的商品，你必须删除大部分的销售历史记录

3. 从文件中载入数据进数据库

交互程序

通常在交互程序中，接口为一行收集数据并发出相应的数据，插入语句，将行添加到正确的表中



The image shows a registration form with the following fields and controls:

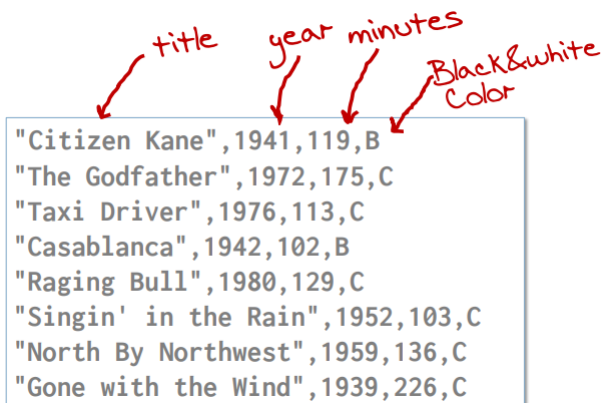
- First Name: Text input field
- Last Name: Text input field
- Email: Text input field
- Gender: Dropdown menu with a downward arrow
- Born: Dropdown menu with a downward arrow
- Register: Button

A red curved arrow points from the Register button to the word **insert** in red text.

然而，通常希望批量上传数据，要么是为了最初填充数据库，要么是因为数据是通过使用文件在不同系统之间交换的

CSV

- 逗号是一种非常流行的分隔值格式，其中一些字段可以用双引号括起来



```
"Citizen Kane",1941,119,B
"The Godfather",1972,175,C
"Taxi Driver",1976,113,C
"Casablanca",1942,102,B
"Raging Bull",1980,129,C
"Singin' in the Rain",1952,103,C
"North By Northwest",1959,136,C
"Gone with the Wind",1939,226,C
```

- 另一种流行的格式是用制表符\t分隔字

```
Citizen Kane	1941	119	B
The Godfather	1972	175	C
Taxi Driver	1976	113	C
Casablanca	1942	102	B
Raging Bull	1980	129	C
Singin' in the Rain	1952	103	C
North By Northwest	1959	136	C
Gone with the Wind	1939	226	C
```

根据文件的位置不同，您不会使用相同的命令

- 有时你想从你的电脑上上传一个文件
- 有时，您想上载其他进程自己存储在DBMS服务器上的文件

上传命令 csv/txt

MySQL

```
1 LOAD DATA INFILE '<文件地址>'
2 INTO TABLE <表名>
3 FIELD TERMINATED BY '<分隔符号>'
4 OPTIONALLY ENCLOSED BY '<可选字段包围符号>'
```

PostgreSQL

```
1 COPY <表名>
2 FROM '<文件地址>'
3 WITH (FORMAT csv)
```

PostgreSQL使用(非标准)SQL

复制命令以加载位于服务器上的文件，默认格式为制表符分隔

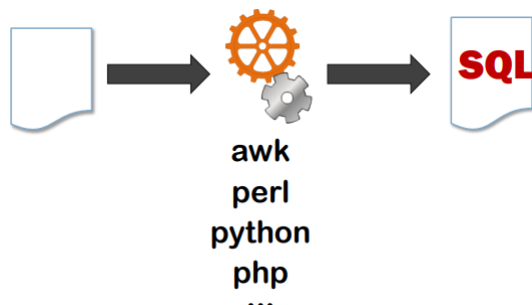
上传命令 xml json

XML也是一种流行的交换格式。大型产品提供用于上传XML文件的实用程序

JSON也是一种越来越流行的格式

有时你会遇到非常奇怪的文本文件格式

使用脚本语言生成INSERT语句通常是最简单的解决方案



4. 函数

大多数DBMS(例外的是SQLite，而不是真正的DBMS)实现了内置的、基于sql的编程语言，当声明性语言不再足够时，可以使用这种语言

不用说，当你费劲地编写了一些非常复杂的代码时，您就不希望每次需要时都复制和粘贴代码

您希望存储表达式并在另一个上下文中重用它，事实上您可以。重新定义方法

定义函数

```
1 CREATE FUNCTION <函数名>(<参数1> <数据类型1>, <参数2> <数据类型2>)  
2 RETURNS <返回数据类型>  
3 AS $$  
4 BEGIN  
5     <函数体>  
6 END  
7 $$ LANGUAGE PLPGSQL;
```

使用函数

一旦创建了函数，就可以像使用任何内置函数一样使用它

请注意，为了安全起见，您通常必须用提供的语言编写函数：一个编写糟糕的C函数可能会使整个服务器瘫痪，破坏数据，等等。所提供的语言提供了一种沙盒环境

Procedural extensions to SQL



T-SQL



(no name)

ORACLE

PL/SQL



PL/PGSQL



SQL PL



nothing ...

注意

不要使用函数来显示查询命令

为什么不好呢，我几乎没有讨论过查询优化器，但是有很多方法可以执行连接，其中一些在大容量的情况下特别有效，查找函数强制执行“一次一行”连接，这在大多数情况下是非常可怕的，函数不应该用来查询数据库

- 你应该向比你更有经验的人寻求帮助，谷歌，论坛等等
- 如果不能在单个SQL语句中做到这一点，那么再去函数中定义

5. 存储过程

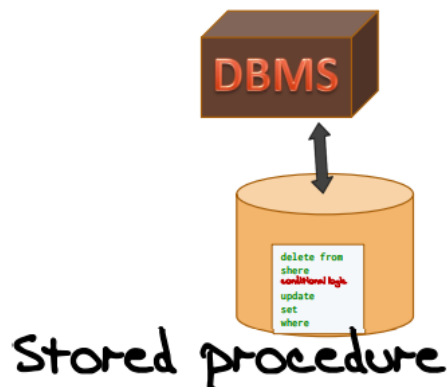
有一些常用的查询，我们可以把它存储下来，将它们转换为单个单元（存储在数据库中）是有意义的

定义存储过程

```
1 CREATE PROCEDURE <存储过程名称>
2
3 SQL 语句...
```

create procedure myproc

```
delete from ...  
where ...  
+ Conditional logic  
update ...  
set ...  
where ...
```



可以通过

```
1 EXECUTE <存储过程名称>
```

进行调用

优点

调用该存储过程是与数据库的单一交互，当数据库位于远程服务器(考虑“云”)上时，将不会浪费时间通过网络与远程服务器交互

安全，对于插入、删除数据而言，我们可以限定必须通过调用存储过程而不能输入命令等

示例

我们想将一些信息添加到数据库，它有

- 名字
- 发行国家
- 年份
- 导演名字
- 演员1名字
- 演员2名字

在 PostgreSQL 中，定义如下

```
1 CREATE FUNCTION movie_registration(  
2     p_title          VARCHAR,  
3     p_country_name  VARCHAR,  
4     p_year           INT,
```



```

5         p_director_fn    VARCHAR, -- first name
6         p_director_sn    VARCHAR, -- second name
7         p_actor1_fn      VARCHAR,
8         p_actor1_sn      VARCHAR,
9         p_actor2_fn      VARCHAR,
10        p_actor2_sn      VARCHAR,
11    ) RETURNS void
12    AS $$
13    DECLARE
14        -- 检查插入数据
15        n_rowcount  INT;
16        n_movieid   INT;
17        n_people    INT;
18    BEGIN
19        -- 插入电影
20        INSERT INTO movies(title, country, year_released)
21            SELECT p_title, country_code, p_year
22            FROM countries
23            WHERE country_name = p_country_name;
24
25        GET DIAGNOSTICS n_rowcount = ROW_COUNT;
26        IF n_rowcount = 0 THEN
27            RAISE EXCEPTION 'country not found in table COUNTRIES'
28        END IF
29        n_movieid := lastval();
30
31        SELECT COUNT(surname)
32        INTO n_people
33        FROM(
34            SELECT p_director_sn AS surname
35            UNION ALL
36            SELECT p_actor1 AS surname
37            UNION ALL
38            SELECT p_actor2 AS surname
39        ) specified_people
40        WHERE surname IS NOT NULL;
41
42        INSERT INTO credits(movieid, peopleid, credited_as)
43        SELECT n_movieid, people.peopleid, provided.credited_as
44        FROM(
45            SELECT COALESCE(p_director_fn, '*') AS first_name,
46            p_director_sn AS surname, 'D' AS credit_as

```

```

47         SELECT COALESCE(p_actor1_fn, '*') AS first_name,
p_director_sn AS surname, 'A' AS credit_as
48         UNION ALL
49         SELECT COALESCE(p_actor2_fn, '*') AS first_name,
p_director_sn AS surname, 'A' AS credit_as
50     ) provided
51     INNER JOIN people
52         ON people.surname = provided.surname
53         AND COALESCE(people.first_name, '*') =
provided.first_name
54     WHERE provided.surname IS NOT NULL
55
56     GET DIAGNOSTICS n_rowcount = row_count;
57     IF n_rowcount != n_people THEN
58         RAISE EXCEPTION 'Some people
59         couldn'' be found '
60     END IF
61 END
62 $$ LANGUAGE pipgsql;
63

```

存储过程的使用

直接使用

```

1 SELECT movie_registration(
2     'The Adventures of Robin Hood',
3     'United States', 1938,
4     'Michael', 'Curtiz',
5     'Errol', 'Flynn',
6     null, null);

```

在函数/存储过程中继续调用

```

1 PERFORM movie_registration(
2     'The Adventures of Robin Hood',
3     'United States', 1938,
4     'Michael', 'Curtiz',
5     'Errol', 'Flynn',
6     null, null);

```

6. 触发器

当表中的数据发生变化时，可以向表附加动作，这些动作将自动执行

一般来说 `SELECT` 不会触发任何操作，只有修改/删除/插入可能会触发触发器

触发器的作用

截断传入的数据

触发器可以用于动态更改输入

例如，您希望确保数据总是小写的，但数据输入程序不强制使用它，并且您无法访问其源代码。触发器可以触发小写 `SQL` 命令

- 需要对每一行操作的时候执行触发器
- 需要在插入之前触发触发器

检查复杂的规则

有些规则过于复杂，无法通过对于表的约束来实现，我们可以增加触发器来检查

- 需要对每一行操作的时候执行触发器
- 需要在插入之前触发触发器

检查数据冗余

当你插入一张表的时候，触发器可以把你的数据提取出来插入到另外一张表里去

- 需要对每一行操作的时候执行触发器
- 需要在插入之后触发触发器

激活触发器

什么时候触发触发器？

事实上，根据触发器的设计目的，它可能会被不同的事件在不同的可能的精确时刻触发

触发时间

我们可以选择激活触发器的时间

1. 插入前
2. 插入后
3. 每插入一行激活一次

不同数据库系统的激活选择不一样，比如



触发事件

另一个重要参数是触发触发器的参数

在删除一行时，不需要触发更改大小写的触发器

主要是在

- 插入
- 更新
- 删除

的时候触发触发器

使用触发器

```
1 CREATE TRIGGER <触发器名字>
2 [BEFORE | AFTER] [INSERT | UPDATE | DELETE]
3 ON <表名>
4 FOR EACH ROW
5 AS BEGIN
6     ... 具体要做的事情
7 END
```

7. 数据审计

管理数据冗余的一个很好的例子是保持审计跟踪

它不会为窃取数据的人做任何事情（请记住SELECT不能触发触发器——尽管对于大型产品，您可以跟踪所有查询）

但它可能对检查那些修改了他们不应该修改的数据的人有用，我们可以追踪修改数据的人和它做出的修改，我们用 PostgreSQL 为例

创建数据修改表

我们可以创建一个表，专门储存我们需要追踪的修改的记录，我们可以使用触发器进行追踪

另一个选项可能是用一个大字符串存储XML或JSON格式的所有更改，以减轻数据库的负担

查看修改

people_audit

auditid	peopleid	type_of_change	column_name	old_value	new_value	changed_by	time_changed
1	95	I	first_name	NULL	Ryan	root@localhost	... 23:05:01
2	95	I	surname	NULL	Gosling	root@localhost	... 23:05:01
3	95	I	born	NULL	1980	root@localhost	... 23:05:01

我们可以在创建好后查看

触发器的作用

触发器是数据库的最后一道防线

即使应用程序没有检查好所有的东西，你也不能逃离触发器，除非通过删除或取消激活它们

但是，尽量避免使用触发器，它们增加了许多复杂性，一个简单的操作可能会因为编写得不好的触发器所做的事情而表现怪异，而且触发器几乎不被注意到。知道触发器是否激活需要特殊的检查

此外，它们经常被用来“修复”从一开始就不应该存在的问题，这些问题通常是由于糟糕的数据库设计造成的

如果可以的话

- 不要使用触发器来修复设计问题

- 最好将存储过程用于触发器

但是，如果用户可以通过其他方式访问数据库，而不是通过您的程序

- 如果有多个接入点，请使用触发器

8.视图

介绍

视图是一个只有在查询中使用**VIEW**才存在的表，它被称为虚拟表，其行为和表一样，也可以执行表可用的操作

视图可以是你想要的复杂查询，并且通常会返回一些不像表那样规范化的东西，但更容易理解

当你改变表格中的某些东西时你会改变它的值，它会反映在视图中，那也会得到不同的值

语法

创建视图

```
1 CREATE VIEW <视图名称> AS
2 <查询语句...>
```

```
1 -- 显示指定视图中的列名
2 CREATE VIEW <视图名称> (<列名1>,<列名2>,...) AS
3 <查询语句...>
```

删除视图

```
1 DROP VIEW <视图名称>
```

调用视图

```
1 SELECT *
2 FROM <视图>
3 WHERE ...
```

视图可以视作一个虚拟表，假设创建了视图 `vmovies`，则下面两种语句没有区别

```
create view vmovies
as select m.movieid,
        m.title,
        m.year_released,
        c.country_name
from movies m
     inner join countries c
       on c.country_code = m.country
```

```
select *
from vmovies
where country_name = 'Italy'
```

```
select *
from (select m.movieid,
            m.title,
            m.year_released,
            c.country_name
      from movies m
           inner join countries c
             on c.country_code = m.country)
     vmovies
where country_name = 'Italy'
```

视图的应用

数据库中的视图是永久对象

其内容与底层表中的数据将会改变，但结构将保持不变

1. 创建一些虚拟表，它们不满足三大范式，但是对用户非常友好
2. 简化查询，让聪明的人设计 View，让新手去使用 View

视图的缺点

视图隐藏了复杂性，因为视图本身可能需要调用相当多的值来显示完整的虚拟表，而可能实际的需求并不需要很多值，有些可能是无用的

视图的问题在于，只要还没有看到它们是如何定义的，就不知道它们可能有多复杂

它们可能是相当无害的，或者它们可能是关于死亡的询问（它们经常是）

注意，我们的计算资源是有限的，需要考虑到峰值的问题

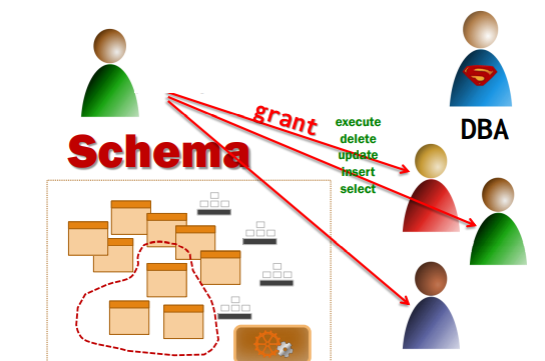
9. 权限

要访问数据库，必须进行身份验证，这通常意味着输入用户名和密码

权限分为两类

- 系统权限赋予用户发出DDL命令
- 更改数据库结构的权限

不是很多人都能得到



语法

权限授予

```
1 GRANT <权限> TO <用户>
```



```
grant select on tablename to accountname
grant insert on tablename to accountname
grant update on tablename to accountname
grant delete on tablename to accountname
grant select, insert on tablename to accountname
```

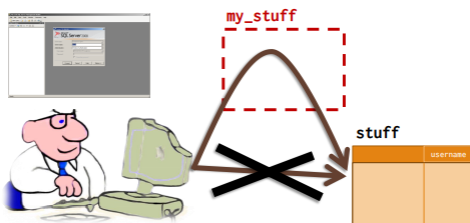
权限撤回

```
1 REVOKE <权限> TO <用户>
```

```
revoke privilege on tablename
from accountname
```

权限的使用

如果有时候我们虽然可以让用户查到某些表，但是我们不想让他们查到某些信息，那么我们可以创建一个视图，隐藏一些列/行/信息



然后

```
1 GRANT <权限> ON <视图>
```