目录

前言

VoltDB 旨在帮助您在吞吐量和可伸缩性方面实现世界级的性能。同时，从本质上讲，VoltDB 是一个关系数据库，可以做传统关系数据库所能做的所有事情。因此，在讨论如何调优数据库应用程序以获得最佳性能之前，我们先回顾一下基础知识。

下面的教程将逐步让您熟悉 VoltDB 的特性和功能，首先从它的 SQL 根源开始，然后介绍其特性，这些特性帮助 VoltDB 在灵活性和性能方面表现优越，主要包括:

- Schema 定义
- SQL 查询
- 分区
- Schema 升级
- 存储过程

出于本教程的需要，让我们假设一个场景，假设我们想要了解更多关于我们居住的地方的信息。比如有多少个城镇?他们在哪儿?有多少人住在那里?我们还能发现什么有趣的事实呢?

我们的研究将利用政府网站上免费提供的数据。现在让我们从基本结构开始讲起。

如何使用本手册

您可以通过仅仅阅读本教程来了解 VoltDB 的工作原理。但如果您愿意，我们鼓励您动手来尝试一下。

本教程使用的数据文件可从公共网站免费获得;文本中提供了链接。然而，初始数据相当大。因此，我们已经创建了一个子集的源文件和预处理数据文件，这些文件可以从 VoltDB 数据库的网站上获得，地址如下:

http://downloads.voltdb.com/technologies/other/tutorial_files_50.zip

对于本教程的每个部分，都有一个包含必要源文件的子文件夹，以及一个包含数据文件的子文件夹 data。要跟随本教程，请执行以下操作:

1. 创建一个工作目录
2. 解压教程文件到工作目录
3. 在教程的每个章节开始时
   a. 将默认值设置为教程文件夹
   b. 拷贝当前章节的教程目录到前面创建的工作目录中，比如
   c. $cp –r tutoriall/* ./

本教程还使用了 VoltDB 命令行命令。请确保设置好您的环境，使命令对您可用。VoltDB 的安装方法请参考文档《UsingVoltDB》。

第一章 创建数据库

与其他 SQL 数据库一样，在 VoltDB 中使用 SQL 数据定义语言(DDL)语句定义数据库模式。因此，如果我们想为我们居住的地方创建一个数据库表，DDL 模式可能如下所示

```
CREATE TABLE towns (
town VARCHAR(64),
county VARCHAR(64),
state VARCHAR(2)
);
```

前面的模式定义了一个表，其中包含了三列:town、county 和 state。我们还可以设置选项，比如默认值和主键。但现在，我们将尽可能保持简单。

启动数据库加载 Schema

一旦定义了模式，就可以初始化并启动数据库，然后加载模式。在初始化 VoltDB 数据库时，有几个可用的选项，我们将在稍后讨论。但现在，我们可以使用最简单的 init 和 start 命令，使用当前机器上的默认选项初始化和启动数据库:

$voltdb init

$voltdb start

voltdb init 命令初始化一个根目录，用于存储 VoltDB 的配置、日志和其他基于磁盘的信息。对于生产数据库，只需要初始化一次根目录。

在进行开发时，您通常希望使用新的设置或使用完全不同的 Schema 重新开始，这时您可以使用 voltdb init -force 命令重新初始化根目录。在本教程中，我们将使用两种方法——启动一个新的数据库和重新启动一个现有的数据库。首先，我们将每次重新初始化数据库根目录。

voltdb start 命令告诉 voltdb 启动数据库进程。一旦启动完成，服务器将报告以下消息:

Server completed initialization.

现在可以加载 Schema 了。为此，您可以使用 VoltDB 交互式命令行实用工具 sqlcmd。创建一个新的终端窗口，并从 shell 提示符发出 sqlcmd 命令:

$ sqlcmd

SQL Command :: localhost:21212

1>

VoltDB 交互式 SQL 命令行首先报告它连接到哪个数据库，然后给出一个行号提示。在提示符处，您可以输入 DDL 语句和 SQL 查询，执行存储过程，或者键入“exit”以结束程

序并返回到 shell 提示符。

要定义数据库模式，您可以手工输入 DDL 模式语句，如果一个文件中包含多个 DDL 语句，您可以使用 file 指令用一个命令处理所有 DDL 语句。由于我们只有一个表定义，我们可以直接输入或剪切粘贴到 sqlcmd 的 DDL:

```
1> CREATE TABLE towns (
2> town VARCHAR(64),
3> county VARCHAR(64),
4> state VARCHAR(2)
5> );
```

使用查询 sql

恭喜你!您已经创建了第一个 VoltDB 数据库。当然，空数据库不是特别有用。因此，您要做的第一件事是创建和检索一些记录，以向您自己证明数据库如您所期望的那样运行。

VoltDB 支持所有标准的 SQL 查询语句，比如 INSERT、UPDATE、DELETE 和 SELECT。您可以通过标准接口(如 JDBC 和 JSON)以编程方式调用查询，也可以将查询包含在编译并加载到数据库的存储过程中。

但是现在，我们只使用 sqlcmd 尝试一些特殊查询。让我们从使用 INSERT 语句创建记录开始。下面的示例为 Billerica、Buffalo 和 Bay View 三个城镇创建了三条记录。一定要在每个语句后面加上分号。

```
1> insert into towns values ('Billerica','Middlesex','MA');
2> insert into towns values ('Buffalo','Erie','NY');
3> insert into towns values ('Bay View','Erie','OH');
```

我们可以查询来验证插入是否按预期工作。以下查询使用 SELECT 语句检索关于数据库记录的信息

4> select count(*) as total from towns;

```
TOTAL
------
3
(1 row(s) affected)
```

5> select town, state from towns ORDER BY town;

```
TOWN STATE
------------ ------
Bay View OH
Billerica MA
Buffalo NY
(3 row(s) affected)
```

当您处理完数据库之后，您可以键入“exit”来结束 sqlcmd 会话并返回到 shell 命令提示符。然后切换回启动数据库的终端会话，并按 CTRL-C 结束数据库进程

第二章 加载和管理数据

尽管 ad hoc 查询非常有用，但是手工输入数据的效率并不高。幸运的是，VoltDB 提供了几个特性来帮助自动化这个过程。

当您使用 CREATE TABLE 语句定义表时，VoltDB 会自动创建存储过程，如果您调用这个存储过程，它会为刚刚创建的表插入记录。您可以预先将想要记录到表中的数据存入数据文件中，例如逗号分隔值(CSV)文件。然后通过命令行工具调用 csvloader 命令读取数据文件，csvloader 使用默认的 inser 存储过程将每个条目作为记录写入指定的数据库表中。

碰巧美国的城镇和其他地标都有现成的数据。地理名称信息服务(GNIS)是美国地质调查局(U.S. Geological Survey)的一部分，提供美国各地官方命名地点的数据文件。我们要用到填充位置的数据文件。这些数据可以从他们的网站

http://geonames.usgs.gov/domestic /download_data.htm 以文本文件的形式获得。

GNIS 提供的信息不仅包括名称、县和州，还包括每个位置的位置(经纬度)和海拔。由于我们不需要提供的所有信息，所以可以将列的数量减少到只需要我们想要的信息。

在我们的例子中，只使用州名称、县名称及其海拔信息。让我们返回修改一下 schema 定义，添加新列:

```
CREATE TABLE towns (
town VARCHAR(64),
state VARCHAR(2),
state_num TINYINT NOT NULL,
county VARCHAR(64),
county_num SMALLINT NOT NULL,
elevation INTEGER
);
```

注意，GNIS 数据包括用于标识县和州的名称和编号。我们重新排列了列，以匹配它们在数据文件中出现的顺序。这使得加载数据更加容易，因为默认情况下，csvloader 命令假定数据列的顺序与模式中指定的顺序相同。

最后，我们可以使用一些 shell 技巧来删除数据文件中不需要的列和行。下面的脚本选择所需的列并删除所有字段为空的记录:

```
$ cut --delimiter="|" --fields=2,4-7,16 POP_PLACES_20120801.txt \
| grep -v "|$" \
| grep -v "||" > data/towns.txt
```

重新启动数据库

因为我们更改了模式并重新排序了 Towns 表的列，所以我们希望从一个空数据库开始重新加载表结构。另外，如果数据库还在运行，可以执行 DROP 表和创建表来删除任何现有数据并替换表定义。

稍后，我们将学习如何重启现有数据库并恢复其数据对象和内容。但是现在，我们将使用与前面学到的命令来重新初始化根目录，并在一个终端会话中启动一个空数据库，在另一个终端会话中使用 sqlcmd 加载数据对象。您需要将—force 参数添加到 voltdb init 命令中，以表明您不需要来自前一个会话的任何旧命令日志或快照。这次我们将使用 file 指令从 DDL 文件加载模式:

[terminal 1]

$ voltdb init --force

$ voltdb start

[terminal 2]

$ sqlcmd

1> FILE towns.sql;

Command succeeded.

2> exit

加载数据

数据库运行并加载数据对象之后，就可以插入新数据了。为此，将默认值设置为教程目录中的/data 子文件夹，并使用 csvloader 命令加载数据文件。

$ cd data

$ csvloader --separator "|" --skip 1 --file towns.txt towns

在上述命令中:

1. --separator 标志允许指定分隔单个数据项的字符。由于 GNIS 数据不是标准的 CSV，所以我们使用--separator 来标识正确的分隔符。
2. 数据文件包含一行列标题。skip 1 标志告诉 csvloader 跳过第一行。
3. file 标志告诉 csvloader 使用什么文件作为输入。如果没有指定文件，csvloader 将使用标准输入作为数据的源。
4. 参数 towns 告诉 csvloader 要将数据加载到哪个数据库表中

csvloader 加载所有的记录到数据库中 ,最终它生成三个日志文件:一个文件中记录了可能出现的任何错误,另一个文件中记录了无法加载数据行,最后还包括一个总结报告,其中包括加载过程花了多长时间,有多少记录加载。

查询数据库

现在我们有了真实的数据，我们可以执行更有趣的查询。例如，哪些城镇位于海拔最高的地方，或者美国有多少地方有相同的名字?

$ sqlcmd

1> SELECT town,state,elevation from towns order by elevation desc limit 5;

```
TOWN STATE ELEVATION
------------------------ ------ ----------
Corona (historical) CO 3573
Quartzville (historical) CO 3529
Logtown (historical) CO 3524
Tomboy (historical) CO 3508
Rexford (historical) CO 3484
(5 row(s) affected)
```

2> select town, count(town) as duplicates from towns

3> group by town order by duplicates desc limit 5;

```
TOWN DUPLICATES
--------------- -----------
Midway 215
```

```
Fairview 213
Oak Grove 167
Five Points 150
Riverside 130
(5 row(s) affected)
```

正如我们所看到的，五个最高的城镇都是落基山脉中被遗弃的矿业城镇。斯普林菲尔德虽然很常见，却没有进入美国前五名。

结合更多数据时，我们可以得到更有趣的发现。现在已经有了位置和海拔的信息。美国人口普查局也可以向我们提供有关人口密度的信息。美国各镇和县的人口数据可从其网站 http://www.census.gov/popest/data/index.html 下载。

要添加新数据，我们必须向数据库添加一个新表。因此，让我们添加一个 people 表。在此过程中，我们可以使用 state_num 和 county_num 这两列为两个表创建索引，这两列是搜索和排序最常用的列。

```
CREATE TABLE towns (
town VARCHAR(64),
state VARCHAR(2),
state_num TINYINT NOT NULL,
county VARCHAR(64),
county_num SMALLINT NOT NULL,
elevation INTEGER
);
CREATE TABLE people (
state_num TINYINT NOT NULL,
county_num SMALLINT NOT NULL,
state VARCHAR(20),
```

```
county VARCHAR(64),
population INTEGER
);
CREATE INDEX town_idx ON towns (state_num, county_num);
CREATE INDEX people_idx ON people (state_num, county_num);
```

同样，对于美国人口普查局下载的文件，我们将列按照它们在数据库表定中出现的顺序排列。并且需要修剪数据文件以删除无关的列。人口普查局的数据包括测量值和估计值。在本教程中，我们将关注一个人口数量。

用于修剪数据文件的 shell 命令如下。(同样，生成的数据文件作为可下载教程包的一部分提供。)

```
$ grep -v "^040," CO-EST2011-Alldata.csv \
| cut --delimiter="," --fields=4-8 > people.txt
```

一旦有了数据和新的 DDL 语句，就可以更新数据库模式。我们可以停止并重新启动数据库，从文本文件加载新模式并重新加载数据。但我们不必这么做。由于我们没有改变 Towns 表或添加一个唯一的索引，我们可以通过简单地剪切和粘贴新的 DDL 语句到 sqlcmd 来改变正在运行的数据库:

```
$ sqlcmd
1> CREATE TABLE people (
2> state_num TINYINT NOT NULL,
3> county_num SMALLINT NOT NULL,
4> state VARCHAR(20),
5> county VARCHAR(64),
```

6> population INTEGER

7>);

8> CREATE INDEX town_idx ON towns (state_num, county_num);

9> CREATE INDEX people_idx ON people (state_num, county_num);

一旦我们创建了新的表和索引，我们可以加载相应的数据文件:

$ cd data

$ csvloader --file people.txt --skip 1 people

现在，我们有两个装载了数据的表。现在我们可以加入表格来寻找海拔和人口之间的关系，就像这样:

$ sqlcmd

1> select top 5 min(t.elevation) as height,

2> t.state,t.county, max(p.population)

3> from towns as t, people as p

4> where t.state_num=p.state_num and t.county_num=p.county_num

5> group by t.state, t.county order by height desc;

HEIGHT STATE COUNTY C4

------- ------ --------- ------

2754 CO Lake 7310

2640 CO Hinsdale 843

2609 CO Mineral 712

2523 CO San Juan 699

2454 CO Summit 27994

(5 row(s) affected)

事实证明，即使不考虑没有人口的鬼城，海拔最高的 5 个有人居住的县都在科罗拉多州。事实上，如果我们反转 select 表达式来查找居住人口最少的县(通过将排序顺序从降序改为升序)，那么最低的县是加利福尼亚州的 Inyo 县--死亡谷之家!

第三章 分区

现在您已经掌握了作为关系数据库的 VoltDB 的基本特性，是时候开始研究是什么使 VoltDB 独一无二了。voldb 最重要的特性之一是分区。

分区是将数据库表的内容组织为多个独立的自治单元。电压数据库分区是独特的 ,因为：

- VoltDB 根据指定的分区列自动分区数据库表。您不必手动管理分区。
- 在一台服务器上可以有多个分区或站点。换句话说，分区不仅仅用于扩展数据量，它还有助于提高性能。
- VoltDB 对数据和访问数据的处理进行分区，这就是 VoltDB 利用并行性提供的吞吐量改进的方式。

分区表

通过将分区列指定为模式的一部分对表进行分区。如果一个表是分区的，每次向该表插入一行时，VoltDB 都会根据分区列的值来决定该行要进入哪个分区。例如，如果按照 name 列对 Towns 表进行分区 那么所有具有相同名称的 Towns 的记录都将保存在相同的分区中。

然而，尽管按名称分区在均匀分布记录方面可能是合理的，但是分区的目标是同时分布数据和处理能力。我们不经常比较同名城镇的信息。然而，比较特定地理区域内的城镇是非

常常见的。让我们把这些记录按州划分这样我们就可以快速地找到指定州内最大或最高的城镇。

town 和 People 表都有表示州名的列。然而，它们略有不同，一个使用州的缩写，另一个使用全称。为了保持一致，我们可以使用 State_num 列来分区，这对两个表都是通用的。

要对表进行分区，只需向数据库模式添加一个 partition TABLE 语句。下面是我们可以添加到模式中的语句，通过 State_num 列对两个表进行分区:

PARTITION TABLE towns ON COLUMN state_num;

PARTITION TABLE people ON COLUMN state_num;

添加了分区信息后，我们需要停止数据库、重新初始化、重新启动和重新加载模式和数据。这一次，我们可以使用 voltadmin shutdown 命令，而不是使用 CTRL-C 来终止数据库进程。voltadmin 命令执行数据库集群的管理功能，而 shutdown 执行数据库的有序关闭，无论是单个节点还是 15 个节点集群。

所以转到第二个终端会话，使用 voltadmin shutdown 来停止数据库:

$ voltadmin shutdown

然后你可以重新初始化和启动数据库，加载新的模式和数据文件:

[terminal 1]

$ voltdb init --force

$ voltdb start

[terminal 2]

$ sqlcmd

1> FILE towns.sql;

Command succeeded.

2> exit

$ cd data

**$ csvloader --separator "|" --skip 1 **

--file towns.txt towns

$ csvloader --file people.txt --skip 1 people

您可能注意到的第一件事是，这次加载数据文件更快。实际上，当 csvloader 运行时，它会创建三个日志文件，总结加载过程的结果。其中一个文件是 csvloader_TABLE-NAME_insert_report。它描述导入进程花费的时间和平均每秒事务数(TPS)。比较添加分区前后的加载时间 ,可以发现添加分区将 Towns 表的吞吐率从大约 5000 个 TPS 提高到 16000 个 TPS——速度是原来的三倍多!

复制表

如前所述，这两个表 Towns 和 People 都有一个 VARCHAR 列作为州名，但是它的值并不一致。所以，我们使用 State_num 列来对两个表进行分区和连接。

State_num 列包含 FIPS 号。也就是说，分配给每个州的联邦标准化标识符。FIPS 号确保状态的惟一和一致标识。然而，尽管 FIPS 数字对于计算很有用，但是大多数人是通过名称而不是数字来考虑位置的。因此，有一个一致的名称来配合数字是很有用的。

我们可以对模式进行规范化，并创建一个单独的表，为每个状态号提供一个权威的状态名，而不是试图修改单个表中的字段。同样，从美国环境保护署的网站 http://www.epa.gov/enviro/html/codes/state.html 免费提供这些信息。虽然不能直接下载为数据文件 ,但是 data 子文件夹中的 data/state.txt 中包含了所有状态的 FIPS 编号和

名称的副本。

所以让我们把这个数据的表定义添加到我们的模式中:

CREATE TABLE states (

abbreviation VARCHAR(20),

state_num TINYINT,

name VARCHAR(20),

PRIMARY KEY (state_num)

);

这种查找表在关系数据库中非常常见。它们减少冗余并确保数据一致性。查找表最常见的两个特点是数据量小和静态值。也就是说，它们主要是只读的。

可以在 State_num 列上对状态表进行分区，就像我们对 Towns 和 Pepole 表。然而，当表相对较小且不经常更新时，最好将其复制到所有分区。这样，即使需要查询另一个分区表(例如按姓氏分区的 customer 表)，存储过程也可以连接这两个表，不管存储过程在哪个分区中执行。

所有记录对所有分区可用的表称为复制表。注意，默认情况下 Voltdb 中创建的表都是复制表。因此，要使状态表成为复制表，只需包含 CREATE，不附带分区表语句即可。

关于复制表的最后一个警告:复制数据的好处是可以从任何单独的分区读取数据。然而，不足之处在于，对复制表的任何更新或插入都必须对所有分区同时执行。这种多分区过程降低了并行处理的好处，并影响吞吐量。这就是为什么不应该把经常更改的表定义为复制表的原因。

第四章 模式更新和持久化

到目前为止，在本教程中，我们已经从头开始初始化和启动数据库，并在每次更改模式时手动重新加载模式和数据。当您第一次开发应用程序并进行频繁更改时，这有时是进行更改的最简单方法。然而，随着应用程序及其使用的数据变得更加复杂，您希望跨会话维护数据库状态。

您可能已经注意到，在本教程的前一节中，我们定义了状态表，但是还没有将其添加到正在运行的数据库中。这是因为我们希望演示修改数据库的方法，而不必每次都从头开始。

保存数据库

首先让我们谈谈耐久性。VoltDB 是一个内存数据库。每次使用 init 和 start 命令重新初始化和启动数据库时，它都会启动一个新的空数据库。显然，在实际的业务环境中，您希望数据能够持久。VoltDB 有几个特性可以跨会话保存数据库内容。

保存数据库的最简单方法是使用 Command log，Command log 是 VoltDB 企业版的默认功能。Command log 将所有数据库活动(包括模式和数据更改)都记录到磁盘。如果数据库停止运行，您可以通过使用 voltdb start 命令重新启动数据库来恢复命令日志。

如果您正在使用企业版，那么现在就尝试一下。使用 voltadmin 关机停止数据库进程，然后使用 voltdb start(不使用 voltdb init)将数据库恢复到之前的状态:

$ voltadmin shutdown

$ voltdb start

命令日志记录使保存和恢复数据库变得简单和自动。另外，如果您正在使用 Community Edition 或不使用命令日志记录，还可以使用快照保存和恢复数据库。

快照是一个完整的 VoltDB 数据库，保存在磁盘上，其中包括在关闭后重新生成数据库

所需的所有内容。您可以使用 voltadmin save 命令在任何时候创建正在运行的 VoltDB 数据库的快照。例如:

$ voltadmin save

默认情况下，快照保存到数据库根目录的子文件夹中。或者，您可以指定快照文件的位置和名称作为 voltadmin save 命令的参数。但是将快照保存到默认位置是有好处的。因为如果根目录中有快照，当数据库重新启动时，voltdb start 命令会自动恢复最近的快照。

为了更简单，VoltDB 让我们在关闭数据库时创建一个最终快照，只需在 shutdown 命令中添加—save 参数。这是在使用社区版本 Voltdb(即不使用命令日志记录)时关闭数据库的推荐方法。让我们试一试:

$ voltadmin shutdown --save

$ voltdb start

我们可以通过在另一个终端会话中执行一些简单的 SQL 查询来验证数据库是否已经恢复:

$ sqlcmd

SQL Command :: localhost:21212

1> select count(*) from towns;

C1

193297

(1 row(s) affected)

2> select count(*) from people;

C1

81691

(1 row(s) affected)

添加和删除表

现在我们知道了如何保存和恢复数据库，可以添加在第三部分中定义的 States 表。使用 sqlcmd 实用程序，可以在数据库运行时“动态”添加和删除表。要添加表，只需使用 CREATE TABLE 语句，就像我们之前所做的那样。在修改现有表时，可以使用 ALTER TABLE 语句。另一种方法是，如果您不关心在表中保存现有数据，您可以执行删除表，然后创建表来替换表定义。

我们将添加一个新表，我们可以简单地输入在 sqlcmd 提示符中输入建表语句(或复制和粘贴)。我们还可以使用 show tables 指令来验证新表是否已经添加。

$ sqlcmd

SQL Command :: localhost:21212

1> CREATE TABLE states (

2> abbreviation VARCHAR(20),

3> state_num TINYINT,

4> name VARCHAR(20),

5> PRIMARY KEY (state_num)

6>);

Command successful

7> show tables;

--- User Tables ----------

PEOPLE

STATES

TOWNS

--- User Views ----------

--- User Export Streams ----------

8> exit

接下来，我们可以从数据文件加载状态信息。最后，我们可以使用 voltadmin save 命令保存数据库的完整副本。

$ csvloader --skip 1 -f data/states.csv states

$ voltadmin save

更新表结构

现在我们已经定义了一个关于州信息的查找表，我们不再需要 town 和 People 表中的冗余列。我们希望保留 FIPS 列 State_num，但是可以从每个表中删除状态列。我们对这两个表的更新模式如下:

CREATE TABLE towns (

town VARCHAR(64),

-- state VARCHAR(2),

state_num TINYINT NOT NULL,

county VARCHAR(64),

county_num SMALLINT NOT NULL,

elevation INTEGER

);

CREATE TABLE people (

state_num TINYINT NOT NULL,

county_num SMALLINT NOT NULL,

-- state VARCHAR(20),

town VARCHAR(64),

population INTEGER

);

如果我们想从头开始重新启动，最好将完整的模式保存在文件中。(或者如果我们想在另一台服务器上重新创建数据库。)但是，要修改正在开发的现有模式，只使用 ALTER TABLE 语句通常更容易。因此，要修改正在运行的数据库以匹配我们的新模式，我们可以使用 ALTER TABLE 和来自 sqlcmd 提示符的 DROP COLUMN 子句:

$ sqlcmd

SQL Command :: localhost:21212

1> ALTER TABLE towns DROP COLUMN state;

Command successful

2> ALTER TABLE people DROP COLUMN state;

Command successful

可以在运行时进行许多模式的更改，包括添加、删除和修改表、列和索引。然而，也有一些限制。例如，您不能向已经包含数据的表添加新的惟一约束。在这种情况下，如果不需要保存数据，可以删除并创建带有新约束的表。或者,如果您需要保存数据,并且确定现有数

据不会违反新约束，您可以先将数据保存到一个快照目录。然后重新初始化数据库根目录，启动数据库，并重新加载新模式,然后使用 voltadmin restore 命令从快照中恢复数据到更新的模式中。

第五章存储过程

现在我们有了一个完整的数据库，可以使用 SQL 查询与之交互。例如，我们可以通过以下 SQL 查询找到任意给定州(例如加利福尼亚州)中人口最少的县:

```
$ sqlcmd
1> SELECT TOP 1 county, abbreviation, population
2> FROM people, states WHERE people.state_num=6
3> AND people.state_num=states.state_num
4> ORDER BY population ASC;
```

然而，在同一个查询中反复输入不同的状态号会很快让人感到厌倦。随着查询变得越来越复杂，情况会变得更糟.

一个简单的存储过程

对于频繁运行的查询，可以创建一个简单的存储过程。存储过程允许您在执行时传入适当的参数，查询语句的主体是需要定义一次即可。存储过程还有一个额外的好处;因为查询是预编译的，所以不需要在运行时计划查询，从而减少了执行每个查询所需的时间。

要创建简单的存储过程(即由单个 SQL 查询组成的过程)，可以使用 create procedure AS 语句在数据库模式中定义整个过程。因此，要将之前的查询转换为存储过程，我们可以在模式中添加以下语句:

CREATE PROCEDURE leastpopulated AS

SELECT TOP 1 county, abbreviation, population

FROM people, states WHERE people.state_num=?

AND people.state_num=states.state_num

ORDER BY population ASC;

其中

- Leastpopulated 是自定义的存储过程名称
- Where 子句中的问号是一个占位符，用于接受存储过程调用时传入的参数值

除了创建存储过程之外，我们还可以指定存储过程是否是分区的，分区存储过程也叫做单分区存储过程。当您对存储过程进行分区时，根据它访问的表将其与特定的分区关联起来。

例如，前面的查询访问了 People 表，并且通过查询条件 people.state_num=?将焦点缩小到分区列 State_num 的特定值。这种情况下存储过程就具备了分区的条件。

注意，您可以在一个单分区存储过程中访问多个表，就像我们在前面的示例中所做的那样。但是，您访问的所有数据都必须在该分区中。换句话说，您访问的所有表都必须基于相同的键值进行分区，对于只读的 SELECT 语句，还可以包括复制表。

因此，我们可以通过添加一个 partition on 子句并指定表和分区列，在 People 表上对存储过程进行分区。如下:

CREATE PROCEDURE leastpopulated

PARTITION ON TABLE people COLUMN state_num

AS

SELECT TOP 1 county, abbreviation, population

FROM people, states WHERE people.state_num=?

AND people.state_num=states.state_num

ORDER BY population ASC;

现在，当我们调用存储过程时，它只在 people.State_num 列与过程的第一个参数匹配的分区中执行。这样就不会涉及到其他分区的数据了，其他分区可以空闲出来并行的处理其他请求。

当然，在使用这个过程之前，我们需要将它添加到数据库中。可以动态地修改存储过程，比如添加和删除表。所以我们不需要重新启动数据库，只需在 sqlcmd 提示符下输入 CREATE PROCEDURE 语句：

sqlcmd>

1> CREATE PROCEDURE leastpopulated

2> PARTITION ON TABLE people COLUMN state_num

3> AS

4> SELECT TOP 1 county, abbreviation, population

5> FROM people, states WHERE people.state_num=?

6> AND people.state_num=states.state_num

7> ORDER BY population ASC;

一旦我们更新了模式，新的过程就可用了。因此，我们现在可以对不同的状态执行多次查询，只需修改存储过程的传参即可:

1> exec leastpopulated 6;

```
COUNTY ABBREVIATION POPULATION
-------------- ------------- -----------
Alpine County CA 1175
(1 row(s) affected)
```

2> exec leastpopulated 48;

```
COUNTY ABBREVIATION POPULATION
-------------- ------------ -----------
Loving County TX 82
(1 row(s) affected)
```

定义更多存储过程

纯 SQL 编写的简单存储过程作为捷径非常方便。然而，有些过程更复杂，需要多个查询和基于查询结果的额外计算。对于更复杂的过程，VoltDB 支持用 Java 编写存储过程。

编写 VoltDB 存储过程并不需要 Java 编程向导。所有 VoltDB 存储过程都具有相同的基本结构。例如，下面用 java 代码再现了我们在前一节中编写的简单存储过程。

```
import org.voltdb.*;①
public class LeastPopulated extends VoltProcedure {②
public final SQLStmt getLeast = new SQLStmt(
" SELECT TOP 1 county, abbreviation, population "
+ " FROM people, states WHERE people.state_num=?"
+ " AND people.state_num=states.state_num"
+ " ORDER BY population ASC;" );③
public VoltTable[] run(int state_num)④
throws VoltAbortException {
voltQueueSQL( getLeast, state_num );⑤
return voltExecuteSQL();⑥
}
}:
```

对本例中标记点的解释：

① 我们首先导入必要的 VoltDB 类和方法。

② 过程本身被定义为一个 Java 类。Java 类名是我们在运行时用来调用过程的名称。在本例中，过程名是 LeastPopulated

③ 在类的开头，声明存储过程将使用的 SQL 查询。这里我们使用来自简单存储过程的相同 SQL 查询，包括使用问号作为占位符。

④ 该过程的主体是一个单一的 run 方法。run 方法的参数是运行时调用过程时必须提供的参数。

⑤ 在 run 方法中，存储过程对一个或多个查询(指定步骤 3 中声明的 SQL 查询名)和占位符使用的参数进行排队。

⑥ 这里我们只有一个带一个参数的查询，即州编号。

⑦ 最后，一个调用执行所有排队的查询，这些查询的结果返回给调用应用程序。

对于这个例子来说，编写一个 Java 存储过程来执行一个 SQL 查询是多余的。我们只是用它来说明程序的基本结构。

在设计与数据库的更复杂交互时，Java 存储过程变得非常重要。VoltDB 存储过程最重要的一个方面是，每个存储过程都作为一个完整的单元执行，即一个事务，作为一个整体成功或失败。如果在事务期间发生任何错误，则在将响应返回到调用应用程序之前回滚事务，或者分区执行任何进一步的工作。

其中一个事务可能正在更新数据库。人口数据恰好来自美国人口普查局包含了实际的人口普查结果和未来几年的人口估计数字。如果我们想要更新数据库，用 2011 年的估计数据(或一些未来的估计)取代 2010 年的结果，我们需要一个过程来完成如下需求:

1. 检查指定的州和县是否已经存在记录。

2. 如果是，使用 SQL UPDATE 语句更新记录。

3. 如果没有，使用 INSERT 语句创建一条新记录。

我们可以通过扩展原来的示例 Java 存储过程来实现这几点。Java 类命名为 UpdatePeople，接下来将用到三个 sql 语句（select update insert）.我们还需要向过程中添加更多的参数，使 People 表中所有列都可以查询。最后，我们添加了所需的查询调用和条件逻辑。首先对 SELECT 语句排队并执行它，然后在对 UPDATE 或 INSERT 语句排队之前计算它的结果(即，是否至少有一条记录)。

如下是完整的存储过程代码

```
import org.voltdb.*;
public class UpdatePeople extends VoltProcedure {
public final SQLStmt findCurrent = new SQLStmt(
" SELECT population FROM people WHERE state_num=? AND county_num=?"
+ " ORDER BY population;");
public final SQLStmt updateExisting = new SQLStmt(
" UPDATE people SET population=?"
+ " WHERE state_num=? AND county_num=?;");
public final SQLStmt addNew = new SQLStmt(
" INSERT INTO people VALUES (?,?,?,?);");
public VoltTable[] run(byte state_num,
short county_num,
String county,
long population)
throws VoltAbortException {
```

```
voltQueueSQL( findCurrent, state_num, county_num );
VoltTable[] results = voltExecuteSQL();
if (results[0].getRowCount() > 0) { // found a record
voltQueueSQL( updateExisting, population,
state_num,
county_num );
} else { // no existing record
voltQueueSQL( addNew, state_num,
county_num,
county,
population);
}
return voltExecuteSQL();
}
}
```

编译 java 存储过程

一旦编写了 Java 存储过程，就需要将它加载到数据库中，然后用与处理简单存储过程相同的方法在 DDL 中声明它。但是首先，Java 类本身需要编译。我们使用 Java 编译器 javac 以与其他 Java 程序相同的方式编译该过程。

在编译存储过程时，Java 编译器必须能够找到在存储过程中使用到的 VoltDB 类和方法。为此，我们必须在 Java 类路径中包含 VoltDB 的 jar 包。这些库位于您安装 voltdb 的子文

件夹/voltdb 中。例如，你安装 VoltDB 到目录/opt/voltdb 中，那么编译存储过程的命令如下：

$ javac -cp "$CLASSPATH:/opt/voltdb/voltdb/*" UpdatePeople.java

一旦将源代码编译成 Java 类，就需要将其(以及数据库使用的任何其他 Java 存储过程和类)打包到 Jar 文件中，并将其加载到数据库中。Jar 文件是压缩和打包 Java 文件的标准格式。使用 jar 命令创建一个 jar 文件，需要自定义 jar 文件的名称和要包含的文件。例如，如下命令可以把 UpdatePeople.java 打包到 storedprocs.jar 中。

$ jar cvf storedprocs.jar *.class

一旦将存储过程打包到 Jar 文件中，就可以使用 sqlcmd load classes 指令将它们加载到数据库中。例如:

$ sqlcmd

1> load classes storedprocs.jar;

最后，我们可以在模式中声明存储过程，这与声明简单存储过程的方法非常相似。但是这一次我们使用来自 CLASS 语句的 CREATE 过程，指定类名而不是 SQL 查询。我们还可以在 People 表上对过程进行分区，因为所有查询都被限制为 State_num 的特定值，即分区列。下面是定义和分区存储过程的语句。

CREATE PROCEDURE

PARTITION ON TABLE people COLUMN state_num

FROM CLASS UpdatePeople;

注意，您不需要在“CREATE PROCEDURE”后面指定过程的名称，因为与简单的存储过程不同，CREATE PROCEDURE FROM CLASS 语句从类的名称获取过程名称;在本例中，存储过程的名称即为 UpdatePeople。

最后，在 sqlcmd 提示符下执行 create 语句:

$ sqlcmd

1> CREATE PROCEDURE

2> PARTITION ON TABLE people COLUMN state_num

3> FROM CLASS UpdatePeople;

存储过程的应用

现在我们有了一个 Java 存储过程和一个更新的模式，让我们把它们结合起来。显然，我们不希望为 People 表中的每个记录手动调用新过程。我们可以编写一个程序来为我们做这件事。幸运的是，我们已经有了一个可用的程序。

csvloader 命令通常使用默认的 INSERT 过程将数据加载到表中。但是，如果您愿意，您可以指定一个不同的过程。因此，我们可以使用 csvloader 调用我们的新过程，用数据文件中的每个记录更新数据库。

首先，我们必须过滤数据到我们需要的列。我们使用与创建初始输入文件相同的 shell 命令，只是切换到选择包含 2011 年估计数据的列，而不是实际的普查结果。我们可以将这个文件保存为 data/people2011.txt(包含在源文件中)：

**$ grep -v "^040," data/CO-EST2011-Alldata.csv **

| cut --delimiter="," --fields=4,5,7,11 > data/people2011.txt

在更新数据库之前，我们先看看人口最少的两个县是哪两个:

$ sqlcmd

SQL Command :: localhost:21212

1> SELECT TOP 2 county, abbreviation, population

2> FROM people,states WHERE people.state_num=states.state_num

3> ORDER BY population ASC;

COUNTY ABBREVIATION POPULATION

--------------- ------------ -----------

Loving County TX 82

Kalawao County HI 90

(2 row(s) affected)

现在，我们可以运行 csvloader 来更新数据库，使用-p 标志指示我们指定的是存储过程名，而不是表名:

**$ csvloader --skip 1 --file data/people2011.txt **

-p UpdatePeople

最后，我们可以通过重复前面的查询来查看更新的结果:

$ sqlcmd

SQL Command :: localhost:21212

1> **SELECT TOP 2 county, abbreviation, population**

2> **FROM people,states WHERE people.state_num=states.state_num**

3> **ORDER BY population ASC;**

COUNTY ABBREVIATION POPULATION

--------------- ------------ -----------

Kalawao County HI 90

Loving County TX 94

(2 row(s) affected)

从更新后的结果表明：按照估计，Loving 县和得克萨斯州的人口正在增长，不再是最小的。

第六章 客户端应用

现在我们有了一个带有数据的工作示例数据库。我们还编写了一个存储过程来演示如何更新数据。为了运行存储过程，我们使用了预先存在的 csvloader 实用程序。然而，大多数应用程序需要比单个存储过程更多的逻辑。理解如何将对数据库的调用集成到客户机应用程序中是生成完整业务解决方案的关键，在本章中，我们将说明客户端应用程序是怎么与 VoltDB 进行交互的。

VoltDB 支持多种编程语言实现客户端应用，每种语言都有自己独特的语法、支持的数据类型和功能。但是，无论使用哪种编程语言，从客户机应用程序调用 VoltDB 的一般过程是相同的:

1. 创建到数据库的客户机连接。
2. 再调用一个存储过程并解释其结果。
3. 完成后关闭连接。

本节课将向您展示如何用几种不同的语言执行这些步骤。

使示例程序具有交互性

尽管人口和位置信息很有趣，但它并不是非常动态的。人口变化不会那么快，而地理位置的变化就更慢了。仅围绕这些数据创建交互式应用程序是困难的。然而，如果我们再添加一层数据，事情就变得有趣了。

美国国家气象局(商务部的一部分)发布通告，描述危险的天气状况。这些警报以 XML

格式在线提供，包括受天气影响地区的州和县的 FIPS 号。这意味着有可能加载与我们拥有人口和海拔数据的相同位置的天气预报。我们不仅可以列出给定州和县的天气警报，还可以根据受影响的人口确定哪些事件的影响最大。

设计解决方案

为了利用这些新数据，我们可以构建一个由两个独立应用程序组成的解决方案:

- 加载天气预报数据
- 另一个用于获取特定位置的警报

一个程序可以周期性地(比如每 5 或 10 分钟)重复加载数据，以确保数据库拥有最新的信息。另一个获取警报通常由用户请求触发。

在任何给定时间，只有几百个天气警报，并且 NWS 网站上的警报每 5-10 分钟仅更新一次。因为它是一个不经常更新的小数据集，所以警报通常是复制表的良好候选。然而，在这种情况下，可以有多个州/县与警报相关联。此外，根据业务解决方案中该功能的数量和使用情况，查找特定州县警报的用户请求性能可能非常重要。

因此，我们可以将数据规范化为两个单独的表:nws_alert 用于存储警报的一般信息，local_event 将每个警报(通过惟一 ID 标识)与它所应用的州和县关联。第二个表可以与 towns 和 people 表在同一列 state_num 上分区。新表和相关索引如下:

```
CREATE TABLE nws_event (
id VARCHAR(256) NOT NULL,
type VARCHAR(128),
severity VARCHAR(128),
SUMMARY VARCHAR(1024),
```

```
starttime TIMESTAMP,
endtime TIMESTAMP,
updated TIMESTAMP,
PRIMARY KEY (id)
);
CREATE TABLE local_event (
state_num TINYINT NOT NULL,
county_num SMALLINT NOT NULL,
id VARCHAR(256) NOT NULL
);
CREATE INDEX local_event_idx ON local_event (state_num, county_num);
CREATE INDEX nws_event_idx ON nws_event (id);
PARTITION TABLE local_event ON COLUMN state_num;
```

可以将新的表声明添加到现有模式文件中。但是，只要 DDL 语句没有重叠或冲突，就可以使用 sqlcmd 将多个单独的模式文件加载到数据库中。因此，为了帮助组织源文件，我们可以在单独的模式文件 weather.sql 中创建新的表声明。我们还需要一些新的存储过程，所以现在先不执行新的 DDL 语句。

设计存储过程

定义了模式之后，我们现在可以定义客户机应用程序所需的存储过程。第一个应用程序加载天气警报，需要两个存储过程:

- FindAlert—确定给定的警告是否已经存在于数据库中

- LoadAlert—将信息插入 nws_alert 和 local_alert 表

第一个存储过程是一个基于 id 列的简单 SQL 查询，可以在模式中定义。

第二个存储过程需要在复制表 nws_alert 中创建一条记录 ,然后根据需要在 local_alert 中创建尽可能多的记录。此外，输入文件将州县的 FIPS 数字列分成由空格分隔的 6 位数值组成的字符串，而不是作为单独的字段。因此，第二个过程必须用 Java 编写，这样它就可以对多个查询进行排队，并在将输入值用作查询参数之前对其进行解码。您可以在教程源码目录的 LoadAlert.java 中找到这个存储过程的代码。

这些过程不是分区的，因为它们访问复制的表 nws_alert，在第二个过程中，必须使用多个不同的分区列值将记录插入分区表 local_alert。

最后，我们还需要一个存储过程来检索与特定州和县关联的警报。在本例中，我们可以根据 state_num 字段对过程进行分区。最后一个过程是 GetAlertsByLocation。

```
CREATE PROCEDURE FindAlert AS
SELECT id, updated FROM nws_event
WHERE id = ?;
CREATE PROCEDURE FROM CLASS LoadAlert;
CREATE PROCEDURE GetAlertsByLocation
PARTITION ON TABLE local_event COLUMN state_num
AS SELECT w.id, w.summary, w.type, w.severity,
w.starttime, w.endtime
FROM nws_event as w, local_event as l
WHERE l.id=w.id and
l.state_num=? and l.county_num = ? and
```

w.endtime > TO_TIMESTAMP(MILLISECOND,?)

ORDER BY w.endtime;

现在已经编写了存储过程并创建了附加的模式文件，我们可以编译 Java 存储过程，将它和第 5 部分中的 UpdatePeople 过程打包到一个 Jar 文件中，并将过程和模式加载到数据库中。注意，必须先加载存储过程，这样接下来的 CREATE PROCEDURE FROM CLASS 语句才能找到要使用的类。

```
$ javac -cp "$CLASSPATH:/opt/voltdb/voltdb/*" LoadAlert.java
$ jar cvf storedprocs.jar *.class
$ sqlcmd
1> load classes storedprocs.jar;
2> file weather.sql;
```

创建 LoadWeather 客户端应用

第一个客户机应用程序 LoadWeather 的目标是读取来自国家气象局的天气警报并将其加载到数据库中。程序的基本逻辑是:

1. 读取和解析 NWS 警报提要。
2. 对于每个警报，首先使用 FindAlert 过程检查它是否已经存在于数据库中。

- 如果是，那就继续。
- 如果没有，使用 LoadAlert 过程插入警告

由于这个应用程序将定期运行，所以我们应该用一种编程语言编写它，这种语言允许轻松解析 XML，并且可以从命令行运行。Python 满足这些要求，因此我们将在示例应用程序中使用它。

客户机应用程序的第一个任务是包含我们需要的所有库。在本例中，我们需要用于输入/输出和解析 XML 的 VoltDB 客户机库和标准 Python 库。我们的 Python 程序的开始是这样的:

```
import sys
from xml.dom.minidom import parseString
from voltdbclient import *
```

程序的开头还包含从标准输入读取和解析 XML 的代码，并定义一些有用的函数。您可以在教程目录中的 LoadWeather.py 程序中找到它。

更重要的是，正如前面提到的，我们必须创建一个客户机连接。在 Python 中，这是通过创建一个 FastSerializer 实例来实现的:

```
client = FastSerializer("localhost", 21212)
```

在 Python 中，我们还必须声明要使用的任何存储过程。在这种情况下，我们必须声明 FindAlert 和 LoadAlert:

```
finder = VoltProcedure( client, "FindAlert", [
FastSerializer.VOLTTYPE_STRING,
] )
loader = VoltProcedure( client, "LoadAlert", [
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
FastSerializer.VOLTTYPE_STRING,
```

FastSerializer.VOLTTYPE_STRING,

FastSerializer.VOLTTYPE_STRING,

FastSerializer.VOLTTYPE_STRING

])

应用程序的大部分工作是一组循环，这些循环遍历 XML 结构中的每个警报，检查它是否已经存在于数据库中，如果不存在，则添加它。同样，如果您感兴趣，可以在教程目录中找到解析 XML 的代码。但是调用 VoltDB 存储过程的代码如下:

```
# Check to see if the alert is already in the database.
response = finder.call([ id ])
if (response.tables):
if (response.tables[0].tuples):
# Existing alert
cOld += 1
else:
# New alert
response = loader.call([ id, wtype, severity, summary,
starttime, endtime, updated, fips])
if response.status == 1:
cLoaded += 1
```

请注意应用程序如何以两种不同的方式使用来自过程的响应:

- FindAlert (finder)的响应用于检查是否返回了任何记录。如果是，则警告已经存在于数据库中。

- LoadAlert (loader)的响应用于验证调用的状态。如果返回状态为 1 或成功，则我们知道警报已成功添加到数据库。

除了状态代码和查询返回的数据之外，过程响应中还有其他信息。但是 LoadWeather 显示了两个最常用的组件。

最后一步，当所有警报处理完毕后，关闭与数据库的连接:

client.close()

运行 LoadWeather 应用

因为 Python 是一种脚本语言，所以在运行代码之前不需要编译代码。但是，您确实需要告诉 Python 在哪里可以找到任何自定义库，比如 VoltDB 客户机。只需将 VoltDB 客户机库的位置添加到环境变量 PYTHONPATH 中。例如，如果将 VoltDB 作为~/ VoltDB 文件夹安装在您的主目录中，要使用的命令是:

$ export PYTHONPATH="$HOME/voltdb/lib/python/"

一旦定义了 PYTHONPATH，就可以运行 LoadWeather 了。当然，您还需要加载天气警报数据。教程文件数据目录中包含一个天气数据示例文件:

$ python LoadWeather.py < data/alerts.xml

或者你也可以直接从 NWS 网站传送最新的警报:

$ curl https://alerts.weather.gov/cap/us.php?x=0 | python LoadWeather.py

创建 GetWeather 应用

现在数据库包含天气数据，我们可以编写解决方案的后半部分——一个应用程序来检索与特定位置关联的所有警报。在实际的示例中，GetWeather 应用程序相对简单，它由一个

用户当前位置为参数的 GetAlertsByLocation 存储过程调用组成。手动运行，一次一个查询。但是在实际中，可能有成百上千的用户同时运行应用程序,Voltdb 数据库的性能非常好.

为了演示我们假设的解决方案的两个方面，我们可以编写 GetWeather 应用程序的两个版本:

- 一个用户界面，用于向用户展示前端，显示将 VoltDB 集成到此类应用程序中是多么容易。
- 一个高性能的应用程序，模拟现实世界负载的数据库.

用户应用

第一个示例将 VoltDB 访问添加到用户应用程序。在本例中，是一个用 HTML 和 Javascript 实现的 web 接口。您可以在教程目录中的/MyWeather 文件夹中找到完整的应用程序。通过在 web 浏览器中打开 GetWeather.html 文件运行应用程序。如果您使用教程目录中包含的示例警报数据，您可以为缅因州的 Androscoggin 县查找警报。

大多数应用程序都围绕用户界面，包括 HTML，CSS 和 Javascript 代码来显示初始表单和格式化结果。只有一小部分代码与 VoltDB 访问相关。

实际上，对于这样的应用程序，VoltDB 简化了编程接口，不需要显式地设置和销毁连接。在本例中，我们可以使用 JSON 接口，它不需要打开和关闭显式连接。相反，您只需使用查询调用数据库，它将以标准 JavaScript 对象表示法(JSON)返回结果。VoltDB 负责管理连接、池化查询等等。

因此，实际的数据库调用只占用文件 getalers .js 中的两个语句;

一个用于构造将要调用的 URL，标识数据库服务器、存储过程(GetAlertsByLocation)以及参数。

另一个用于执行实际调用并指定回调例程。

```
var url = "http://localhost:8080/api/1.0/" +
"?Procedure=GetAlertsByLocation&Parameters=" +
"[" + statenum + "," + countynum + "," + currenttime + "]";
callJSON(url,"loadAlertsCallback");
```

一旦存储过程完成，就会调用回调例程。回调例程使用过程响应(这次是 JSON 格式)，这与 LoadWeather 应用程序的做法非常相似。它首先检查状态以确保存储过程运行成功，然后解析响应内容并将其格式化以便显示给用户。

```
function loadAlertsCallback(data) {
if (data.status == 1) {
var output = "";
var results = data.results[0].data;
if (results.length > 0) {
var datarow = null;
for (var i = 0; i < results.length; i++) {
datarow = results[i];
var link = datarow[0];
var descr = datarow[1];
var type = datarow[2];
var severity = datarow[3];
var starttime = datarow[4]/1000;
var endtime = datarow[5]/1000;
```

```
output += '<p><a href="' + link + '">' + type + '</a> '
+ severity + '<br/>' + descr + '</p>';
}
} else {
output = "<p>No current weather alerts for this location.</p>";
}
var panel = document.getElementById('alerts');
panel.innerHTML = "<h3>Current Alerts</h3>" + output;
} else {
alert("Failure: " + data.statusstring);
}
}
```

高性能应用

GetWeather 的第二个示例模拟许多用户同时访问数据库。它与 VoltDB 软件附带的投票者示例应用程序非常相似。

在这种情况下，我们可以用 Java 编写应用程序，与之前使用 LoadWeather 时一样，我们需要导入 VoltDB 库并打开到数据库的连接。Java 中的代码是这样的:

```
import org.voltdb.*;
import org.voltdb.client.*;
[ . . . ]
/*
```

```
* Instantiate a client and connect to the database.
*/
org.voltdb.client.Client client;
client = ClientFactory.createClient();
client.createConnection("localhost");
```

然后程序执行一个特别的查询。可以通过调用@AdHoc 系统过程向应用程序添加特殊查询。通常，为了提高性能，最好总是使用存储过程，因为它们是预编译的，并且可以分区。但是，在这种情况下，如果查询在程序开始时只运行一次，以获得每个州的有效县号列表，则不会产生什么负面影响。

您可以像使用自己的存储过程一样使用@AdHoc 这样的系统过程，识别过程和 callProcedure 方法中的任何参数。同样，我们使用过程响应中的状态来验证过程是否成功完成。

```
ClientResponse response = client.callProcedure("@AdHoc",
"Select state_num, max(county_num) from people " +
"group by state_num order by state_num;");
if (response.getStatus() != ClientResponse.SUCCESS){
System.err.println(response.getStatusString());
System.exit(-1);
}
```

应用程序的大部分是一个程序循环，它随机分配州号和县号，并使用 GetAlertsByLocation 存储过程查找该位置的天气警报。这里的主要区别是，我们不是同步地调用过程并等待结果，而是异步地调用存储过程。

```
while ( currenttime - starttime < timelimit) {
// Pick a state and county
int s = 1 + (int)(Math.random() * maxstate);
int c = 1 + (int)(Math.random() * states[s]);
// Get the alerts
client.callProcedure(new AlertCallback(),
"GetAlertsByLocation",
s, c, new Date().getTime());
currenttime = System.currentTimeMillis();
if (currenttime > lastreport + reportdelta) {
DisplayInfo(currenttime-lastreport);
lastreport = currenttime;
}
}
```

异步过程调用对于高速应用程序非常有用，因为它们确保数据库总是有查询要处理。如果同步调用存储过程，每次调用一个，数据库只能在应用程序发送查询时以最快的速度处理查询。在每个存储过程调用之间，当应用程序处理结果并设置下一个过程调用时，数据库处于空闲状态。实际上，当应用程序执行其他工作时，数据库的所有并行性和分区都被浪费了。

通过异步调用存储过程，当应用程序设置下一个过程调用时，数据库可以对查询进行排队，并行处理多个单分区。换句话说，您的应用程序和数据库都可以以最快的速度运行。这也是模拟多个同步客户机同时访问数据库的好方法。

异步过程调用完成后，通过调用 callProcedure 方法的第一个参数中标识的回调方法通

知应用程序，即 AlertCallback。然后，回调过程处理响应，将其作为参数接收，就像您的应用程序在同步调用之后所做的那样。

```
static class AlertCallback implements ProcedureCallback {
@Override
public void clientCallback(ClientResponse response) throws Exception {
if (response.getStatus() == ClientResponse.SUCCESS) {
VoltTable tuples = response.getResults()[0];
// Could do something with the results.
// For now we throw them away since we are
// demonstrating load on the database
tuples.resetRowPosition();
while (tuples.advanceRow()) {
String id = tuples.getString(0);
String summary = tuples.getString(1);
String type = tuples.getString(2);
String severity = tuples.getString(3);
long starttime = tuples.getTimestampAsLong(4);
long endtime = tuples.getTimestampAsLong(5);
}
}
txns++;
if ( (txns % 50000) == 0) System.out.print(".");
```

}

}

最后，一旦应用程序在运行了指定的时间长度（默认情况下为 5 分钟)之后，它将打印出一个最终报告并关闭连接。

// one final report

if (txns > 0 && currenttime > lastreport)

DisplayInfo(currenttime - lastreport);

client.close();

运行 GetWeather 应用

如何编译和运行客户机应用程序取决于它们使用的编程语言。对于 Java 程序，比如样例 GetWeather 应用程序，您需要在类路径中包含 VoltDB JAR 文件。如果将 VoltDB 安装为主目录的子目录~/ VoltDB，可以将 VoltDB 和相关的 JAR 文件以及当前工作目录像这样添加到 Java 类路径中:

$ export CLASSPATH="$CLASSPATH:$HOME/voltdb/voltdb/*:$HOME/voltdb/lib/*:./"

然后，你可以使用标准 Java 命令编译和运行应用程序:

$ javac GetWeather.java

$ java GetWeather

```
Emulating read queries of weather alerts by location...
.........................
1403551 Transactions in 30 seconds (46783 TPS)
.............................
1550652 Transactions in 30 seconds (51674 TPS)
```

在应用程序运行时，它定期显示处理的事务数量的指标。这些值将根据服务器类型、VoltDB 集群的配置(每个主机的站点、集群中的节点数量等)和其他环境因素而变化。但是这些数字让您大致了解了数据库在负载下的性能。

结论

数据库模式的结构、表和过程的分区以及客户机应用程序的设计都将影响性能。在编写客户端应用程序时，尽管每种编程语言的具体情况有所不同，但基本内容如下:

- 创建到数据库的连接。
- 调用存储过程并解析结果。尽可能使用异步调用来最大化吞吐量。
- 完成连接后关闭连接

另外 ,有关如何设计客户端应用程序以及如何调优它和数据库以获得最大性能的更多信息，可以在网上的 Using VoltDB 手册和 VoltDB 性能指南中找到。