Contents

[Contents 1](#_Toc1401808)

[<NoSQL精粹> 3](#_Toc1401809)

[NoSQL与关系数据库区别？ 3](#_Toc1401810)

[聚合 4](#_Toc1401811)

[物化视图 5](#_Toc1401812)

[分布式模型 5](#_Toc1401813)

[一致性 6](#_Toc1401814)

[版本戳 8](#_Toc1401815)

[数据库业务逻辑 8](#_Toc1401816)

[键值数据模型与文档数据模型的区别？ 8](#_Toc1401817)

[模式迁移 9](#_Toc1401818)

[混合持久化 9](#_Toc1401819)

[Sqlite3 10](#_Toc1401820)

[创建sql文件 10](#_Toc1401821)

[打开或创建数据库 10](#_Toc1401822)

[导入sql文件 11](#_Toc1401823)

[CRUD操作 11](#_Toc1401824)

[事务操作 11](#_Toc1401825)

[MySQL 11](#_Toc1401826)

[install MySQL 11](#_Toc1401827)

[Issue 12](#_Toc1401828)

[若记不住命令格式，请查询help 12](#_Toc1401829)

[若存在字符集问题 12](#_Toc1401830)

[mysql命令 13](#_Toc1401831)

[-设置用户与权限 (参数<PHP and MySQL Web开发>p171 14](#_Toc1401832)

[数据库 14](#_Toc1401833)

[数据库优化 18](#_Toc1401834)

[Redis (remote dictionary service) 18](#_Toc1401835)

[MongoDB 22](#_Toc1401836)

[安装 22](#_Toc1401837)

[启动mongod --ubuntu 22](#_Toc1401838)

[MongoDB shell 23](#_Toc1401839)

[命令行 23](#_Toc1401840)

[基本数据类型 26](#_Toc1401841)

[使用shell执行脚本 26](#_Toc1401842)

[索引 26](#_Toc1401843)

[创建索引，基于字段或多个字段 27](#_Toc1401844)

[固定集合 27](#_Toc1401845)

[TTL索引 27](#_Toc1401846)

[全文本索引 27](#_Toc1401847)

[地理空间索引 28](#_Toc1401848)

[GridFS存储文件 28](#_Toc1401849)

[聚合 29](#_Toc1401850)

[聚合框架 29](#_Toc1401851)

[MapReduce 30](#_Toc1401852)

[聚合命令 31](#_Toc1401853)

[复制 31](#_Toc1401854)

[单台服务器副本集 31](#_Toc1401855)

[多台服务器副本集 32](#_Toc1401856)

[分片 33](#_Toc1401857)

[单台服务器集群 33](#_Toc1401858)

[PostgreSQL 37](#_Toc1401859)

[install PostgreSQL 37](#_Toc1401860)

[Window 37](#_Toc1401861)

[Linux 37](#_Toc1401862)

[连接数据库 37](#_Toc1401863)

[SQL语法 38](#_Toc1401864)

[字符串 39](#_Toc1401865)

[时间类型 40](#_Toc1401866)

[数组: type[] 40](#_Toc1401867)

[区间类型： 41](#_Toc1401868)

[PostgreSQL专用语法 42](#_Toc1401869)

[XNAT 43](#_Toc1401870)

# <NoSQL精粹>

## NoSQL与关系数据库区别？

使用无模式的数据，可以运行在集群环境中，并且能够牺牲传统数据库所具备的一致性，以换取另外一些有用的特性

分类：键值，文档，列族，图

关系数据库价值：获取持久化数据，并发（通过事务控制），共享数据库集成

关系模型和内存中的数据结构之间存在差异，关系元组中的值必须简单才行，不能包含嵌套记录或列表等任何结构，如果在内存中使用了较为丰富的数据结构，那么要把它保存到磁盘之前，必须先将其转换成“关系”形式。

关系数据库 + ORM (对象-关系映射框架)

关系型数据库并不是设计给集群用的，关系型数据库可以把数据划分为几个集合，并将其分别放在各自独立的服务器上运行，于是就能有效地对数据库分片。这么做虽然能将负载分散到多个服务器之中，但是应用程序必须控制所有分片，它要知道数据库中的每份小数据都存放在哪个服务器里才行。而且查询、参照完整性、事务、一致性控制等操作也都无法以跨分片的方式执行。

大多数NoSQL数据库的研发动机，都是为了要在集群环境中运行。关系型数据库使用ACID事务来保持整个数据库的一致性，而这种方式本身与集群环境相冲突

选用NoSQL的两个主要原因：在很多应用程序的开发过程中，大量精力和时间都放在了内存数据结构和关系型数据库之间的映射上面；关系数据库是为独立运行的计算机而设计的，许多NoSQL数据库是为集群环境而设计

共享数据库集成 -> web服务集成？

如何用SQL交互，那么必须使用关系型的数据结构；然而用web服务交互时，则可以使用嵌套记录及列表等更丰富的数据结构。这些数据通常存放在XML文档中，最近也流行以json格式来保存它们。

使用web服务集成，大多数使用传输文本信息的http协议。然而如果交互过程对性能要求很高，可以考虑使用二进制协议。

由于内部数据加与外部通信服务之间已经解耦，所以外界并不关心数据如何存储，这样就可以选用非关系型数据库了

## 聚合

在领域驱动设计中，我们想把一组相互关联的对象视为一个整体单元来操作，我们通过原子操作更新聚合的值，并且在与数据存储通信时，也以聚合为单位。

关系映射能够秀好地捕捉数据元素及其关系，但是它没有“聚合实体”概念，在关系模型中，可以用“外键”来表达这种关系，但是那样做无法区分某个关系是否表示聚合

选用面向聚合模型的决定性因素：就在于它非常适合在集群上运行。在集群上运行时，我们需要把采集数据时所需的节点数降至最小。如果在数据库中明确包含聚合结构，那么这就可以根据这一重要信息，知道哪些数据需要一起操作了，而且这些数据应该放在同一个节点中。

聚合对事务处理有影响：关系型数据库允许把任意表格中的任意行组合起来，放在一个事务中操作。具有原子性、一致性、隔离性和持久性。重点是原子性，在单一操作中更新跨越多张表的数个行。该操作要么完全成功，要么彻底失败，而且并发执行的多个操作之间是彼此隔离的；面向聚合的数据库确实不支持跨越多个聚合的ACID事务，它每次只能在一个聚合结构上执行原子操作。也就是说，如果我们想以原子方式操作多个聚合，那么就必须自己组织应用程序的代码，在实际应用中，大多数原子操作都可以局限于某个聚合结构内部

在集群上运行时，聚合是中心环节，因为数据库必须保证将聚合内的数据存放在同一个节点上。聚合还是“更新”操作的最小数据单位，对事务控制来说，以聚合为操作单元，其大小正合适。

大多数NoSQL数据库的关键特征在于它们都使用聚合。

面向聚合数据库获取数据时以聚合为单元，因此，它只能保证单一聚合内部内容的原子性。如果一次更新多个聚合，那就必须设法应对中途发生的错误。

如果待处理的数据中存在大量关系，那么这意味着你更应该选用关系型数据库，而不是NoSQL型数据库。若关系很多，关系型数据库(通过join)查询多个表效率很低，则需要采用图数据库。图数据库遍历关系非常迅速，原因是由于图数据库会多花一些时间用于插入关系数据，以此来缩短遍历关系时所需的时间。

## 物化视图

关系型数据库提供“视图”，其展示数据的角度与数据库存储数据的方式有所不同。视图就好比一张关系表，只不过它是通过基表算出来，在访问视图时，数据库会算出视图中的数据。

生成某些视图需要大量的计算工作，于是产生了“物化视图“（预先算好并缓存在磁盘中的视图）。如果数据读取非常频繁，而访问者又不介意略陈旧的数据，那么使用物化视图效率比较高

如何构建物化视图？

* 一旦基础数据有变动，那么立即更新物化视图。（比如适合读多写少）
* 定期通过批处理更新物化视图

面向聚合的数据库通常能够用不同方式重组同一聚合的数据，以计算出各种”物化视图“

## 分布式模型

面向聚合数据库非常适用于横向扩展方式，因为聚合此时就自然成了数据分布单元。

数据分布有两种：复制（同一份数据copy到多个节点）和分片（不同数据存放在不同节点中）

分片：一般来说，数据库的繁忙体现在：不同用户需要访问数据集中的不同部分。在这种情况下，我们把数据的各个部分存放于不同的服务器中，以此实现横向扩展。为了获得近乎理想的效果（N台服务器分担1/N负载），必须保证需要同时访问的那些数据都存放在同一节点上，所以怎样存放数据，才能保证用户基本上只需要从一台服务器中获取它。考虑使用面向聚合的数据库，把聚合作为分布数据的单元。

NoSQL数据库提供”自动分片“功能，可以让数据库自己负责把数据分布到各分片，并且将数据访问请求引导至适当的分片上。

分片可以提升读取和写入效率。

复制可以提升读取性能，同时改善数据库的”故障恢复能力“。

主从复制：数据由主节点复制到从节点，所有写入请求都由主节点处理，读取请求可以由主节点处理，也可以交给从节点

主从复制有助于提升数据访问性能；可以增强”读取操作的故障恢复能力“

采用自动指派方案，不仅配置起来较为简单，而且当主节点出错时，集群可以自动指派新的主节点，以减少停机时间。

主从复制缺点：数据的不一致性，如果数据更新还没有全部通知给从节点，那么不同的客户端就可能于不同的从节点中读出内容各异的值。主节点是系统的瓶颈与弱点。可以通过”对等复制“解决问题.

“对等复制”:没有主副节点概念，所有节点都可以接受写入请求和读取操作。节点将自身的写入操作通知给其他节点

对等复制缺点：数据的一致性。由于两个不同的节点可以同时处理写入操作，可能导致“写入冲突”, 当然也会有数据读取的不一致。

如何解决“写入冲突”？

当某节点写入数据时，各节点之间相互协调，确保不发生冲突.

任意节点先写入数据，然后处理冲突----合并写入操作

## 一致性

集中式关系型数据库（强一致性） -> 面向集群的NoSQL数据库(CAP, 最终一致性)

更新一致性 =写冲突： 两个客户端同时修改一份数据

悲观方式：避免发生冲突，采用写入锁。

乐观方式：先让冲突发生，然后检测冲突并处理。方法1：采用条件更新，任意客户在执行更新操作之前，都要先测试数据的当前值和其上一次读入的读是否相同；方法2: 保存两份更新数据，采用版本控制系统相同的方法合并有冲突的更新，如用户手动处理或根据规则自动合并

大家一般会采用悲观方式来处理并发，并发编程涉及一个根本问题，那就是在安全性（避免更新冲突之类的错误）与响应能力之间权衡。悲观方式通常会降低系统响应能力。

读取一致性（逻辑一致性） =读写冲突： 某客户端在另一个客户端执行写入操作的过程中读取数据

为了避免“读写冲突”造成的逻辑不一致，关系型数据库支持事务概念。客户端执行写入操作的过程封装到一个事务中，那么另一客户端读取的要么是更新之前的值，或更新之后的值。

面向聚合的数据库通常支持单一聚合内部的事务。不支持聚合之间。

复制一致性： 要求从不同副本中读取同一个数据项时，所得到的值相同。

最终一致性：在任意时刻，节点中都可能存在”复制不一致“，然而只要不再继续执行其他更新操作，那么上一次更新操作的结果最终将会反映到全部节点中去。（比如通过机器重启或网络重连后，记录在日志或队列里未完成的操作会重试，复制到其余节点）

CAP定理：Consistency（一致性）、Availability(可用性)、Partition tolerance（分区耐受性）。

可用性：如果客户可以同集群中的某个节点通信，那么该节点就必然能够处理读取及写入操作。

分区耐受性：如果发生通信故障，导致整个集群被分割成多个无法互相通信的分区时，集群仍然可用。

当系统可能会遭遇”分区“状况时（比如分布式系统），我们需要在”一致性“与”可用性“之间进行权衡。

若确保一致性，必须访问所有节点。但若网络发生故障，则无法确定是否有房间（因为节点A可能有，而节点B已被预订完），系统失去了”可用性“。

若改善可用性，即使网络断了，任节点可以操作，然后网络连上，更新不一致如访问两个节点的客户都订到了最后一间客房。

根据业务领域，可以解决写入一致，从而提高可用性和性能：

Case1: 旅行公司允许一定数量的超卖；某些旅馆总是会在全部订满的名额之外多留出几间客房；有些旅馆选择发现预订冲突之后向客户致歉并取消此预订。这些做法行得通，是因为这些方案所付出的代价，要比因为网络故障而彻底无法预订的代价小。

Case2: 多个购物车。即使网络有故障，你池总是能够修改购物车中的商品。这么做有可能导致多个购物车出现。而结账过程则会将两个购物车合并。因为客户在下单之前有机会检视一下购物车中的东西。

根据业务领域，可以处理读取不一致：

媒体网站中发帖子，那么旧页面持续几分钟也没多大关系。

牺牲一些”持久性“以换取更好的性能：

某个数据库大部分时间都在内存中运行，更新操作也直接写入内存，并且定期将数据变量写回磁盘，那么，它就可以大大提高响应请求的速度了。

大型网站可能会有很多用户来访问，而网站要将每个用户正在做的事情作为临时信息，以某种”会话状态“的形式保存起来。关键问题：会话数据就算丢了也无所谓，与整个网站访问速度都变慢相比，它造成的损失要小很多。

处理请求所用的节点越多，避免”不一致“问题的能力就越强。要想保证”强一致性“，需要使用多少个节点才行？

若执行读取操作时所需联系的节点R,确认写入操作时所需征询的节点数W,以及复制因子N，只有R+W>N,才能保证读取操作的”强一致性“。若复制因子N=3, 写入操作就要得到全部节点的确认才行W=3, 只需联系一个节点R=1就能完成读取操作

## 版本戳

乐观离线锁的条件更新？

客户端执行操作时，将重新读取”商业事务“所依赖的信息，并检测该信息在首次读取之后是否一直没有变动，若一直未变，则将其展示给用户。实现此技术有个好办法：那就是保证数据库中的记录都有某种形式的”版本戳“。版本戳是一个字段，每当记录中的底层数据改变时，其值也随之改变。读取数据时可以记下版本戳，这样的话，在写入数据之前，就可以先检查一下数据版本是否已经变了。

构建版本戳的四种方法：计数器，GUID, 根据资源内容生成hash,使用上一次更新时的时间戳。

在分布式系统中，可以采用”由版本戳构成的数组“来检测不同节点之间是否发生了”相互冲突的更新操作“

## 数据库业务逻辑

集中式数据库，处理计算逻辑：一种是在数据库服务器上，另一种是在客户端计算机上。若客户端，要把大量数据从数据库服务器搬到客户端。

集群：map-reduce

## 键值数据模型与文档数据模型的区别？

在键值数据库中，基本上都是通过键来搜索聚合内容，而在文档数据库中，我们提交的查询关键词往基于文档的内部结构。

键值数据库是一张简单的hash table, 只有针对单个键的操作才具备”一致性“。使用键值数据库时，大部分精力都要花在设计键名上：可以用某种算法生成键吗？可以使用户信息（例如ID、电子邮件地址等）当键吗？或者可以用时间戳等数据库以外的值来生成吗？

键值数据库并不关心键值对里的值。它可以是二进制块、文本、json、xml等

适合场景：保存会话，购物车数据、用户配置等信息

不适合场景：数据间关系，含有多项操作的事务，查询数据，操作关键字集合

文档数据库：副本集通常用于处理”数据冗余“， “自动故障切换“、”读取能力扩展“，”无需停机的服务器维护“和“灾难恢复”等事项。

适合场景：事件记录，内容管理系统及博客平台，网站分析与实时分析，电子商务应用程序

不适合场景：包含多项操作的事务

## 模式迁移

关系型数据库中的大数据集很难迁移其模式，因为移动大量数据并改变其结构，通常需要锁定数据库表。

NoSQL可以无模式，但应用程序却必须有模式，因为从数据库中读取数据时，应用程序必须解析数据流。

## 混合持久化

各种数据库引擎都有其适合操作的数据结构及数据量。

应该以多种语言混合编写同一应用程序，以这些语言各自的优势来解决其中不同的问题。复杂的应用程序通常要面对各类问题，针对每个问题，选用适当的语言来解决。

应用程序没有必要非得用一种数据库应对所有需求，因为不同数据库的设计目标不同，并非所有问题都能用一种数据库优雅地解决

将直接数据库操作封装为服务：实际上Riak and Neo4J等诸多NoSQL数据库都提供了现成的REST API.

促使NoSQL数据库发展的原因：需要迅速访问大量数据。面向聚合的数据库读出或取得聚合的速度要比关系型数据库快很多，因为后者的数据散布在许多表中。因为NoSQL数据库便于在集群中”分片“与”复制“，所以可以对其进行水平扩展。联系紧密的数据，用图数据库来获取，要快于使用join命令的关系型数据库

# Sqlite3

SQLite是一个进程中关系数据库，SQLite设计为嵌入在应用中，而不是作为一个单独的数据库服务器程序

支持数据库大小至2TB

ACID事务

$sudo apt-get install sqlite3 安装sqlite3

## 创建sql文件

create table project (

name text primary key,

description text,

deadline date

);

create table task (

id integer primary key autoincrement not null,

priority integer default 1,

details text,

status text,

deadline date,

completed\_on date,

project text not null references project(name)

);

insert into project (name, description, deadline)

values ('pymotw', 'Python MOdule of the Week', '2010-11-01');

insert into task (details, status, deadline, project)

values('write about select', 'done', '2010-10-03', 'pymotw');

insert into task (details, status, deadline, project)

values('write about random', 'waiting', '2010-10-10', 'pymotw');

insert into task (details, status, deadline, project)

values('write about sqlite3', 'actie', '2010-10-17', 'pymotw');

## 打开或创建数据库

$sqlite3 todo.db3

## 导入sql文件

sqlite> .read todo\_schema.sql; 执行sql文件

## CRUD操作

sqlite> select \* from task; 查询

sqlite> delete from task where id=2; 删除

sqlite> insert into task (details, status, deadline, project) 插入

values('write about random', 'waiting', '2010-10-10', 'pymotw');

sqlite> update task set status='done' where id=4; 更新

## 事务操作

sqlite> begin;

sqlite> … //crud operation

sqlite> commit; 递交

(或者sqlite> rollback; 撤销)

# MySQL

## install MySQL

//安装过程会要求输入root用户的密码，所以有user: root, password: q

$sudo apt-get install mysql-server

//客户端程序访问MySQL服务器

$mysql -u root -p 本机访问

输入root密码

$mysql -h hostname -P 3306 -u root -p 远程访问

若无法远程连接，需要如下排查

1. 排除网络或防火墙问题

$ping 192.168.1.12 假定该地址是mysql数据库ip

$telnet 192.168.1.12 若连接失败，配置防火墙

配置防火墙，开启3306端口

vi /etc/sysconfig/iptables

-A INPUT -m state –state NEW -m tcp -p tcp –dport 3306 -j ACCEPT（允许3306端口通过防火墙）

/etc/init.d/iptables restart（重启防火墙使配置生效）

2. 检查MySQL配置

如果开启了防火墙，telnet还是失败，通过netstat查看3306的端口状态：

netstat -apn|grep 3306

tcp6 0 0 127.0.0.1:3306 :::\* LISTEN 13524/mysqld

注意红色的地方，这说明3306被绑定到了本地。检查一下/etc/mysql/my.cnf的配置，这里可以配置绑定ip地址。

bind-address=addr

不配置或者IP配置为0.0.0.0，表示监听所有客户端连接。

3. 检查用户访问权限

MySQL建用户的时候会指定一个host，默认是127.0.0.1/localhost，那么这个用户就只能本机访问， 其它机器用这个用户帐号访问会提示没有权限，host改为%，，表示允许所有机器访问。

#mysql -u root -p

mysql> use mysql;

mysql> select host,user from user; 查看用户权限

mysql> update user set host=’%’ where user=’root’;

mysql> exit;

$sudo service mysql restart 重启mysql服务

$sudo /etc/init.d/mysql start run mysql

## Issue

### 若记不住命令格式，请查询help

mysql>help;

mysql>help create;

mysql>help create database;

### 若存在字符集问题

mysql> SHOW CREATE TABLE your\_table; 查看表模式

查看数据库字符集

mysql>use yourdb;

mysql> show variables like 'character\_set\_database';

mysql> SHOW FULL COLUMNS FROM my\_tablename; Getting the collation of columns

若查询参数是uft8,而表默认是latin,会产生collate冲突，如果解决？

mysql> drop database books; 删除数据库

若出现错误，则直接删除文件夹

>sudo rm -rf /var/lib/mysql/books;

mysql> drop table orders; 删除表

创建数据库

mysql> CREATE DATABASE knowledge CHARACTER SET utf8 COLLATE utf8\_general\_ci;

创建数据表

knowledge.sql

$mysql -u root -p knowledge < knowledge.sql //create table

$sudo cp nameid.tsv /var/lib/mysql/knowledge

mysql>load data infile “nameid.tsv” into table nameid ignore 1 lines; //import data

mysql>select \* from nameid;

mysql>select \* into outfile “nameid.csv’ fields terminated by ‘,’ from nameid; //output data

## mysql命令

**mysql>help 显示帮助信息**

mysql>status 从MySQL服务器提取状态信息

mysql>exit 退出

SQL语句不区分大小写，但数据库和表的名称区分大小写

### -设置用户与权限 (参数<PHP and MySQL Web开发>p171

root用户通常只用作管理目的，对于每个需要使用该系统的用户，应该为他们创建一个帐号和密码

权限存储在名为mysql的数据库中的5个系统表中，分别是mysql.user, mysql.db, mysql.host, mysql.tables\_priv, mysql.columns\_priv

除了GRANT命令，可以直接修改这些表，手动更新，MySQL服务器不会注意到已经修改，需要运行mysql> flush privileges;

创建用户user,密码为password使用数据库test，有所有权限

mysql> GRANT ALL on test.\* to user IDENTIFIED by 'password';

创建用户qzlin,密码123，使用所有数据库的所有权限，且允许他向其他人授予这些权限

mysql> grant all on \* to qzlin identified by '123' with grant option;

删除用户

mysql> revoke all privileges, grant from qzlin;

授予适当的权限

mysql> grant select, insert, update, delete, index, alter, create, drop on books.\* to qzlin;

减少一些权限

mysql> revoke alter, create, drop on books.\* from qzlin;

当建立一个连接之后，对于所发送的每一个请求，MySQL都会检查是否有执行该请求的权限级别。系统首先将检查全局权限（在user表中），如果这些还不够，系统将再检查db表和host表。如果仍然没有足够的权限，MySQL将检查tables\_priv表，如果权限还不够，最后将检查columns\_priv表

### 数据库

#### 数据库管理操作

显示所有数据库

mysql> show databases ;

创建数据库

MySQL数据库系统可以支持许多不同的数据库。通常每个应用程序需要一个数据库

mysql> CREATE DATABASE test;

mysql> USE test; 切换数据库test

显示当前数据库的所有表

mysql> show tables ;

查看某表的详细信息

mysql> describe customers;

mysql> show tables from books; 查询数据库中的所有表

mysql> show columns from books.tables; 查询所有列

mysql> show grants for user; 查看用户所有拥有的权限

mysql> explain

select \* from customers; 获取查询的工作过程

备份MySQL数据库

$mysqldump -u root -p --all-databases > all.sql

**$mysqldump -u root -p db1> db1.sql**

$mysqldump -u root -p --databases db1 db2 > dbs.sql

$mysqldump -u root -p db table1 table2 > db\_tables.sql

恢复数据库

**$mysql -u root -p db < db.sql**

导入SQL文件方式创建表

>mysql -h host -u user -D books -p < book.sql

其中-D指定数据库,book.sql里含sql语句

从csv文件导入

**mysql> load data infile "newbooks.txt" into table books;**

在默认情况下，文件中的数据字段必须通过Tab键分隔，而且必须包括在单引号内，同时每一行都必须由换行\n符进行间隔。特殊字符必须用\进行转义

#### CRUD操作

book.sql

create table customers

( customerid int unsigned not null auto\_increment primary key,

name char(50) not null,

address char(100) not null,

city char(30) not null

);

create table orders

( orderid int unsigned not null auto\_increment primary key,

customerid int unsigned not null,

amount float(6,2),

date date not null

);

create table books

( isbn char(13) not null primary key,

author char(50),

title char(100),

price float(4, 2)

);

create table order\_items

( orderid int unsigned not null,

isbn char(13) not null,

quantity tinyint unsigned,

primary key (orderid, isbn)

);

use books;

insert into customers values

(NULL, 'qzlin', '25 Oak Street', 'Airport West'),

(NULL, 'tom', '375 North Road', 'Yahoo');

字段为AUTO\_INCREMENT,意味着若该列赋NULL或没指定值，MySQL会自动生成唯一

以命令行方式创建表，及CRUD操作

mysql> CREATE TABLE employees (

empid int not null,

lastname varchar(30),

firstname varchar(30),

salary float,

primary key (empid) );

mysql> INSERT INTO employees VALUES (1, ‘ Blum’, ‘Rich’, 25000.00); 插入

mysql> DELETE FROM employees WHERE empid = 2; 删除

mysql> SELECT \* FROM employees WHERE salary>40000; 查询

mysql>update customers 更新

set address = '250 Road'

where customerid = 4;

修改表

改列数据类型

mysql> alter table customers

modify name char(70) not null;

添加列

mysql> alter table orders

add tax float(6,2) after amount;

删除列

mysql> alter table orders

drop tax;

mysql> drop table orders; 删除表

mysql> drop database books; 删除数据库

查询Julie Smith的订单

select orders.orderid, orders.amount, orders.date

from customers, orders

where customers.name = 'Julie Smith'

and customers.customerid = orders.customerid;

查询哪些顾客已经订购了关于Java的图书

select customers.name

from customers, orders, order\_items, books

where customers.customerid = orders.customerid

and orders.customerid = order\_items.customerid

and order\_items.isbn = books.isbn

and books.title like '%Java%';

查询哪些顾客没有订单

select customers.name

from customers left join orders

using (customerid)

where orders.orderid is null;

合计信息

查询哪些顾客的平均订单总金额超过50

select customerid, avg(amount)

from orders

group by customerid

having avg(amout)>50;

MySQL中的数据库将被映射到具有某种文件结构的目录，而表则映射到文件

BLOB类型存储二进制数据，如图像，声音等

LONGTEXT类型存储长TEXT字段，字符数2^32-1

在脚本中使用数据库

MYSQL=` which mysql` 查找mysql客户端程序

$MYSQL test -u test -ptest <<EOF

SHOW TABLES;

SELECT \* FROM employees WHERE salary > 4000;

EOF

## 数据库优化

* 数据库的数据越小越好，故最小化冗余
* 最小列数据类型，尽可能少NULL, 主键短
* 简化权限 提高查询速度
* 随着更新和删除操作，数据支离破碎，周期优化mysql> optimize table tablename;
* 使用默认值，只在与默认值不同的时候才插入数据
* 简化索引

存储引擎和事务

数据库每个表可以使用不同的存储引擎

mysql> create table orders type=innodb;

mysql> alter table orders type=innodb;

MyISAM: 默认类型，Indexed Sequential Access Method（有索引的顺序访问方法），存储记录和文件和标准方法。

**InnoDB: 事务安全，还支持外键**

当事务非常重要，如存储财务数据的表格，或在insert and select语句是交错执行的情况下，如在线的消息栏或论坛系统，应该使用InnoDB

# Redis (remote dictionary service)

也称为**数据结构服务器，发布订阅系统**，作为有用的数据结构算法和程序的工具包。作为栈，队列或优先队列， 对象存储系统（通过hashtable），集合；支持集合操作（并，交，差），提供原子命令，和多步命令的事务机制；键到期及缓存

1. Download and install [Redis-x64-3.0.500-rc2.msi](https://github.com/MSOpenTech/redis/releases/download/win-3.0.500-rc2/Redis-x64-3.0.500-rc2.msi) from <https://github.com/MSOpenTech/redis/releases>
2. Run Redis server:

C:\Program Files\redis > redis-server redis.windows.conf

1. Run Redis cli:

C:\Program Files\redis > redis-cli

127.0.0.1:6379> ping

>keys \* 取出当前所有key

>type mykey 查询key对应的类型是list, string, map…

>del mykey

>exists mykey 是否存在指定key

// 键值

>set key value //mset key1 value1 key2 value2 …

> get key //mget key1 key2

//hash table (map) 嵌套的Redis对象

mset key1 value1 key2 value2

mget key1 key2

hmset key key1 value1 key2 value2 …

hvals key

hkeys key

hmget key key1 key2

for example:

//使用:作为分离的键

mset user:qzlin:name “qizhong lin” user:qzlin:password changeit

//不使用分离的键

hmset user:qzlin name “qizhong lin” password changeit

hvals user:qzlin

hkeys user:qzlin

// 列表

rpush qzlin:wishlist google yahoo baidu //建立列表

llen qzlin:wishlist //长度

lrange qzlin:wishlist 0 -1 //取子集

lrem qzlin:wishlist 0 google //删除元素

rpoplpush qzlin:wishlist qzlin:visited //从左列表弹出元素， 在右列表压入元素

// set 集合

sad key value1 value2 … //创建集合

smembers key //获取集合

sinterstore key key1 key2 //求交集

sdiffstore key key1 key2 //差集

sunionstore key key1 key2 //并集key1, key2, 结果存入key中

for example:

sad news nytimes.com baidu.com

smembers news

// 有序集合 (随机存取的优先级队列)

zadd key score1 key1 score2 key2

zrange key idx0 idx 1

zrangebyscore key score0 score1

// 加权后求并集

zunionstore destination numkeys key1 key2 … [weights weight1 weight2 …] [aggregate sum|min|max]

for example:

zadd visits 500 google 9 yahoo 10 baidu

zunionstore importance 2 visits votes weights 1 2 aggregate sum

事务: multi … exec ; multi和exec内多命令按事务执行，在使用multi命令时，命令在实际上并不执行，而是排入队列，然后按顺序执行。

到期：有助于避免总的键集无限增长，做法是安排Redis经过一定的时间就删除一个键-值对

expire key seconds

setex key second value // 设置key value的生命

ttl key //获取key的生命

persist key //最近使用缓存算法(MRU, Most Recently Used)： 每当你检索一个值时，更新它的到期时间，确保你最近使用的键将继续保留在Redis中，而被忽视的键将正常到期

Redis命名空间称为数据库(database),以数字为键。默认的命名空间0（也称为数据库0）交互

// select 0

set greeting hello

get greeting

select 1 //通过select切换到数据库1

get greeting //返回(nil)

所有的数据库都运行在同一服务器实例内，Redis就允许用move命令，在不同命令空间之间移动键. 若单个Redis服务器运行不同的应用程序，又要允许这些应用程序相互之间交换数据

move greeting 2

select 2

get greeting

别的操作命令： rename, type, del, flushdb, flushall.

//发布-接收模式

127.0.0.1:6379> brpop comments 300 //某客户端监听 key=comments, 超时为5分钟,

127.0.0.1:6379> lpush comments “hello world!” //某客户端发布 key-values

Key points:

redis server启动后，独占进程，能否改为后台服务？

redis server服务是单线程的，能否在同一台机器上启动多个实例？也就是修改默认端口6379？

$redis-server --port 6378 --daemonize yes 服务端

$redis-cli -p 6378 客户端

Redis允许在运行的过程中，在不重启服务器的情况下更改服务器配置

127.0.0.1:6378> config set tcp-keepalive 60

127.0.0.1:6378> config set port 6380

127.0.0.1:6380>

注意：先把心跳包发送时间间隔修改成60秒。

Redis主从复制

特性：

* redis使用异步复制
* 一个master拥有多个slave
* master死了，会自举一个slave作为master

做法：

打开三个终端启动三个redis 服务实例，然后启动三个redis client连接

$redis-server --daemonize yes

$redis-server --daemonize yes --slaveof 127.0.0.1 10000

$redis-server --daemonize yes --slaveof 127.0.0.1 10000

从而两个slave会同步一个主

Redis集群

集群技术是构建高性能网站架构的重要手段，试想在网站承受高并发访问压力的同时，还需要从海量数据中查询出满足条件的数据，并快速响应，我们必然想到的是将数据进行切片，把数据根据某种规则放入多个不同的服务器节点，来降低单节点服务器的压力。

集群主要是利用切片技术来组建的集群

集群要实现的目的是要将不同的 key 分散放置到不同的 redis 节点，这里我们需要一个规则或者算法.

Redis 集群中内置了 16384 个哈希槽，当需要在 Redis 集群中放置一个 key-value 时，redis 先对 key 使用 crc16 算法算出一个结果，然后把结果对 16384 求余数，这样每个 key 都会对应一个编号在 0-16383 之间的哈希槽，redis 会根据节点数量大致均等的将哈希槽映射到不同的节点。

使用哈希槽的好处就在于可以方便的添加或移除节点。

当需要增加节点时，只需要把其他节点的某些哈希槽挪到新节点就可以了；

当需要移除节点时，只需要把移除节点上的哈希槽挪到其他节点就行了；

在新增或移除节点的时候不要让我们先停掉所有的 redis 服务

做法：

先启动三个redis服务实例

创建集群，使三个实例互相通讯, （具体见网上）

$./redis-tri.rb create –replicas 0 127.0.0.1:9001 127.0.0.1:9002 127.0.0.1:9003

连接127.0.0.1:9001的客户端，设置或获取的数据，可能被redis cluster自动重定向到127.0.0.1:9003。

# MongoDB

## 安装

Install and set env

install mongoDB in ubuntu

https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/

Import the public key used by the package management system.

$sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv EA312927

Create a list file for MongoDB

$echo "deb http://repo.mongodb.org/apt/ubuntu trusty/mongodb-org/3.2 multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-3.2.list

Reload local package database.

$sudo apt-get update

Install the MongoDB packages.

$sudo apt-get install -y mongodb-org

## 启动mongod --ubuntu

The MongoDB instance stores its data files in /var/lib/mongodb and its log files in /var/log/mongodb by default, and runs using the mongodb user account. You can specify alternate log and data file directories in /etc/mongod.conf. See systemLog.path and storage.dbPath for additional information.

Start MongoDB

$sudo service mongod start

Verify that the mongod process has started successfully by checking the contents of the log file at /var/log/mongodb/mongod.log for a line reading

[initandlisten] waiting for connections on port <port>

Stop MongoDB

$sudo service mongod stop

Restart MongoDB

$sudo service mongod restart

类似window方式使用

卸载mongo包

$dpkg -l | grep mongo

$sudo apt-get remove --purge mongodb-org

$ sudo apt-get remove --purge mongodb-org-mongos

$ sudo apt-get remove --purge mongodb-org-server

$ sudo apt-get remove --purge mongodb-org-server

$ sudo apt-get remove --purge mongodb-org-tools

wget https://fastdl.mongodb.org/linux/mongodb-linux-x86\_64-ubuntu1404-3.2.8.tgz

tar zxf mongodb...gz

$ gedit ~/.bashrc 配置环境变量

PATH=$PATH:~/Downloads/mongo…/bin

export PATH

$source ~/.bashrc

$mkdir -p /data/db

$chown -R $USER:$USER /data/db

启动mongod --window

>mongod --dbpath ./data 默认是c:/data/db

默认情况下，MongoDB监听27017端口，mongod还启动一个HTTP服务器，监听数字比主端口号高1000的端口，也就是28017端口

$curl localhost:28017

## MongoDB shell

MongoDB自带JavaScript shell,可以shell中使用命令行与MongoDB实例交互。Shell是一个功能完备的JavaScript解释器，可运行任意JavaScript程序

>mongo localhost:27017/test

启动时，shell会连到MongoDB服务器的test数据库，并将数据库连接赋值给全局变量db.这个变量是通过shell访问MongoDB的主要入口点

### 命令行

为了方便习惯SQL shell用户，shell包含一些非JavaScript语法的扩展

>help

> show dbs //show all database

> use test // select database

> db.dropDatabase() // remove database

> show collections //show collections of selected database

> db.customer.drop() //delete collection

//read

第一个参数查询条件，第二个参数投影

db.customer.find() // read all documents of one collection of one database

db.customer.find({‘firstname’:’Alice’}) //根据查询条件检索

db.users.find({"usrname": "joe", "age": 27}); //查询条件=条件1 and 条件2

db.users.find({}, {"email": 1, "\_id": 0}); //0表示剔除，1表示想要

查询条件：$lt, $lte, $gt, $gte, $in, $nin, $or,

db.users.find({"age": {"$gte": 18, "$lte", 30}});

db.raffle.find({"$or": [{"ticket\_no": 725}, {"winner": true}]});

正则表达式查询

db.users.find({"name": /joe/i});

数组查询

找到既有appble,又有banana的文档

db.food.find({"fruit": {"$all": ["appble", "banana"]}});

db.food.find({"fruit": {"$size": 3}}) 根据数组长度

db.blog.posts.findOne(criteria, {"comments": {"$slice": -10}}); 返回后10条评论

内嵌文档查询

条件：子文档完全匹配

db.people.find("name": {"first": "Joe", "last": "Schemoe"});

内嵌文档特定键

db.people.find("name.first": "Joe", "name.last": "Schmoe");

// update

update是一种特殊的更新，若没有找到符合更新条件的文档，就会以这个条件和更新文档为基础创建一个新文档。如果找到了匹配的文档，则正常更新。说明同一套代码既可以用于创建文档又可以更新文档

常规程序：

blog = db.analytics.findOne({url: "/blog"});

if (blog) {

blog.pageviews ++;

db.analytics.save(blog);

} else {

db.analytics.save({url: "/blog", pageviews: 1});

}

Mongo update

db.analytics.update({"url": "/blog"},

{"$inc": {"pageviews": 1}}, true);

默认情况下，更新只能对符合匹配条件的第一个文档执行操作。要是有多个文档符合条件，只有第一个文档会被更新，其他文档不会发生变化。要更新所有匹配的文档，可以将update的第4个参数设置为true

文档替换： 第二个参数作为文档替换第一个参数查询到的文档

>db.customer.update({‘firstname’:’Alice’}, {‘firstname’:’QiZhong’, ‘lastname’:’Lin’})

模式改变

{

“\_id”: …,

“name”: “joe”,

“friends”: 32,

“enemies”: 2

}

改成

{

“\_id”: …,

“username”: “joe”,

“relationships”: {

“friends”: 32,

“enemies”: 2

}

}

命令如下：

var joe = db.users.findOne({"name": "joe"});

joe.relationships = {"friends": joe.friends, "enemies": joe.enemies};

joe.username = joe.name;

delete joe.friends;

delete joe.enemies;

delete joe.name;

db.user.update({"name": "joe"}, joe);

部分更新：第一个参数是查询文档，第二个参数是修改器

若字段不存在，则创建它

增加值

db.analytics.update({"url": "www.example.com"},

{"$inc": {"pageviews": 1}});

修改某字段值

db.users.update({"\_id": ...},

{"$set": {"favorite book": "War and Peace"}});

删除某字段

db.users.update({"name": "joe"},

{"$unset":{"favorite book": 1}});

数组修改器

db.blog.posts.update({"title": "A blog post"},

{"$push", {"comments": {

"name": "joe",

"email": "joe@example.com",

"content": "nice post."

}}});

加入到集合中

db.users.update({"\_id", ...},

{"$addToSet": {"emails", "joe@gmail.com"}});

数组元素访问

db.blog.update({"post": post\_id},

{"$inc": {"comments.0.votes": 1}});

>db.customer.remove({‘lastname’:’Lin’}) //delete

### 基本数据类型

Null, true/false, 数值（默认使用64位浮点型，对于整数用NumberInt()/NumberLong()）, 字符串，日期(new Date()), 正则表达式(/foobar/i), 数组([“a”, “b”]), 内嵌文档({“foo”: “bar”}, 对象id(ObjectId()), 二进制数据，代码(Javascript代码function(){…})

### 使用shell执行脚本

$mongo ip:port/db script1.js script2.js script3.js

>load(“script1.js”)

注意：在脚本中，可以访问db变量，以及其他全局变量，但不能使用shell语法糖（如use db; show collections;等），只能使用对应的shell辅助函数

db.getSisterDB(“foo”) use foo

db.getMongo().getDBs() show dbs

db.getCollectionNames() show collections

## 索引

数据库索引与书籍索引类似

不使用索引的查询称为全表扫描

Explain()函数查看mongodb在执行查询相关的的信息

db.users.find({"username": "user01"}).explain()

### 创建索引，基于字段或多个字段

由于机器性能和集合大小的不同，创建索引有可能需要花几分钟时间。

使用索引的查询可以瞬间完成，然而使用索引是有代价的：对于添加的每一个索引，每次写操作（插入，更新，删除）都将耗费更多的间。这是因为，当数据发生变动时，mongodb不仅要更新文档，还要更新集合上的所有索引。通常在一个特定的集合上，不应该拥有两个以上的索引。

db.users.ensureIndex({"username": 1});

如果有一个拥有N个键的索引，那么你同时免费得到了所有这N个键的前缀组成的索引

通常，一个字段的基数越高（字段拥有不同值的数量），这个键上的索引就越有用，比如id

所有的数据库索引信息都存储在system.indexes集合中。这是一个保留集合，不能在其中插入或者删除文档。只能通过ensureIndex or dropIndexes对其进行操作

查看给定集合上的所有索引信息

db.collectionName.getIndexes()

默认情况下，mongodb会尽可能快地创建索引，阻塞所有对数据库的读请求和写请求，一直到索引创建完成

### 固定集合

Mongodb中的普通集合是动态创建的，而且可以自动增长以容纳更多的数据

固定集合相当于循环队列，如果已经没有空间了，最老的文档会被删除以释放空间，新插入的文档会占据这块空间。

db.createCollection("my\_collection", {"capped": true, "size": 10000})

### TTL索引

time-to-live index具有生命周期的索引，这种索引允许为每一个文档设置一个超时时间。一个文档到达预设置的老化程序之后就会被删除。这种类型的索引对于缓存问题（比如会话的保存）非常有用

db.foo.ensureIndex({"lastUpdated": 1}, {"expiredAfterSecs": 60\*60\*24});

mongodb每分钟对TTL索引进行一次清理，所以不应该依赖以秒为单位的时间保证索引的存活状态

### 全文本索引

应该在离线状态下创建全文本索引

全文本索引也会导致比“普通”索引更严重的性能问题，因为所有字符都需要被分解、分词、并且保存到一些地方

### 地理空间索引

2dsphere索引：用于地球表面类型的地图

2d索引：用于平面地图和时间连续的数据

2dsphere允许使用GeoJSON格式指定点、线和多边形。占可以用形如[longitude, latitude]经度纬度表示

查询类型：交集intersection,包含within,接近nearness

### GridFS存储文件

存储大型二进制文件

使用GridFSg来代替独立的文件存储工具

GridFS会自动平衡已有的复制或者为mongodb设置的自动分片

通常来说，如果你有一 些不常改变但是经常需要连续访问的大文件，那么使用GridFS再合适不过了

可以用mongofiles工具在GridFS中上传文件、下载文件、查看文件列表、搜索文件、以及删除文件。

mongofiles --help

mongofiles put foo.txt 上传

mongofiles list 列表

mongofiles get foo.txt 下载

mongofiles search

mongofiles delete

GridFS背后的理念：可以将大文件分割为多个比较大的块，将每个块作为独立的文档进行存储。除了将文件的每一个块单独存储之外，还有一个文档用于将这些块组织在一起并存储该文件的元信息

GridFS中的块会被存储到专用的集合中。块默认使用的集合是fs.chunks，不过可以修改为其他集合。在块集合内部，各个文档的结构如下：

{

"\_id": ..., 块id

"n": 0, 块在文件中的位置

"data": BinData("..."), 块所包含的二进制数据

"files\_id": ... 所属文件

}

每个文件的元信息被保存在一个单独的集合中，默认情况下这个集合是fs.files.这个文件集合中的每一个文档表示GridFS中的一个文件，文档中可以包含与这个文件相关的任意用户自定义信息。文件元信息结构如下：

{

"\_id": ...,

"length": ,

"chunkSize": ...,

"uploadDate": ...

"md5": ...

}

## 聚合

数据提取后，一般需要进行数据分析，可以用聚合工具

### 聚合框架

对集合中的文档进行变换和组合

基本上，可以用多个构件创建一个管道pipeline,用于对一连串的文档进行处理。这些构件包括筛选filtering, 投射projecting,分组grouping,排序sorting,限制limiting,和跳过skipping

找出发表文章最多的那个作者

db.articles.aggregate(

{"$project": {"author": 1}}, 将每篇文章文档中的作者投射

{"$group": {"\_id": "$author", "count": {"$sum": 1}}}, 将作者按名字group,统计次数

{"$sort": {"count": -1}}, 降序排列

{"$limit": 5} 限制返回结果

)

数学运算： $add, $subtract, $multiply, $divide, $mod

db.employees.aggregate(

{

"$project": {

"totalPay": {

"$subtract": [{"$add": ["$salary", "$bonus"]}, "$401k"]

}

}

}

);

日期表达式：$year, $month, $week, $dayOfMonth, $dayOfWeek, $dayOfYear, $hour, $minute, $second

字符串表达式：$substr, $concat, $toLower, $toUpper

逻辑表达式：$cmp, $strcasecmp, $eq, $ne, $gt, $gte, $lt, $lte, $and, $or, $not, $ifNull

分组：$group, $sum, $avg, $max, $min, $first, $last,

数组操作符：$addToSet, $push

大部分操作符的工作方式都是流式的，只要有新文档进入，就可以对新文档进行处理，但是$group and $sort必须要等收到所有的文档之后，才能对文档进行分组，然后才能将各个分组发送给管道中的下一个操作符

### MapReduce

使用JavaScript作为查询语言，因此它能够表达任意复杂的逻辑。然而，这种强大是有代价的：MapReduce非常慢，不应该用在实时的数据分析中

MapReduce能够在多台服务器之间并行执行。它会将一个大问题拆分为多个小问题，将各个小问题发送到不同的机器上，每台机器只负责完成一部分工作。

map -> shuffle -> reduce

计算集合中所有文档所有键出现的次数

map = function() {

for (var key in this) { //this指当前文档

emit(key, {count: 1})

}

};

reduce = function(key, emits) {

total = 0;

for (var i in emits) {

total += emits[i].count;

}

return {"count": total};

}

mr = db.runCommand({

"mapreduce": "foo",

"map": map,

"reduce": reduce);

网页分类，找出哪个主题最为热门

map = function() {

for (var i in this.tags) {

var recency = 1/(new Date() - this.date);

var score = recency \* this.score;

emit(this.tags[i], {"urls": [this.url], "score": score});

}

};

reduce = function (key, emits) {

var total = {urls: [], score: 0}

for (var i in emits) {

emits[i].urls.forEach(function(url) {

total.urls.push(url);

});

total.score += emits[i].score;

}

};

### 聚合命令

db.foo.count(); 返回集合中的文档数量

db.runCommand({"distinct": "people", "key": "age"}); 给定键的所有值 （去重）

## 复制

创建副本集

副本集是一组服务器，其中有一个主服务器(primary),用于处理客户端请求；还有多个备份服务器(secondary)，用于保存主服务器的数据副本。如果主服务器崩溃了，备份服务器会自动将其中一个成员升级为新的主服务器

关键点：

客户端在单台服务器上可以执行的请求，都可以发送到主节点执行（读，写，执行命令，创建索引等）

客户端不能在备份节点上执行写操作

默认情况下，客户端不能从备份节点中读取数据。在备份节点上显式地执行setSlaveOk之后，客户端就可以从备份节点中读取数据了

### 单台服务器副本集

第一个mongo shell

前提条件：创建数据目录/data/db，同时用root帐户启动shell

mongo --nodb 启动mongo shell, 但不连接到任何mongod

replicaSet = new ReplSetTest({"nodes": 3}) 创建三个服务器的副本集

replicaSet.startSet() 启动副本集

replicaSet.initiate() 配置复制功能

现在有3个mongod进程，分别运行在20000、20001、20002端口

第二个mongo shell，客户端1

conn1 = new Mongo("localhost:21000")

testReplSet:PRIMARY> primaryDB = conn1.getDB("test")

>primaryDB.isMaster() 查看副本集的状态

conn1 = new Mongo("qzlin-virtualBox: 20000");

primaryDB = conn1.getDB("test");

//insert

for (i = 0 ; i < 1000; ++ i) {

primaryDB.coll.insert({count: i})

}

primaryDB.coll.count()

备份节点可能会落后于主节点，可能没有最新写入的数据，所以备份节点在默认情况下会拒绝读取请求，以防止应用程序意外拿到过期的数据

这是为了保护应用程序，以免意外连接到备份节点，读取到过期数据。如果希望从备份节点读取数据，需要设置“从备份节点读取数据没有问题”标识。

conn2 = new Mongo("localhost:20001");

conn2.setSlaveOk(); 设置“从备份节点读取数据没有问题”标识

secondaryDB = conn2.getDB("test");

secondaryDB.coll.find();

自动故障转移automatic failover, 如果主节点挂了，其中一个备份节点会自动选举为主节点

primaryDB.adminCommand({"shutdown": 1}) 关闭某一节点

secondaryDB.isMaster() 查看主节点

replicaSet.stopSet() 关闭副本集

### 多台服务器副本集

前提条件：

多台机器互通, $ping server1, $ping server2 （若是多虚拟机，见linux.docx）

创建空的mongod.conf and mongod.log文件

启动Server1上的mongod

$mongod --replSet spock -f mongod.conf --fork --logpath "mongod.log"

启动server2上的mongod

$ssh server2

$mongod --replSet spock -f mongod.conf --fork --logpath "mongod.log"

只是这几个服务器mongod无法知道彼此存在，为了让每个mongod知道彼此，需要创建一个配置文件，在配置文件中列出每一个成员，并且将配置文件发送给任何一个服务器，该服务器会负责将配置文件传播给其他成员

创建配置文件

>config = {

"\_id": "spock",

"members": [

{"\_id": 0, "host": "server1:27017"},

{"\_id": 0, "host": "server2:27017"}

]

}

配置副本集，从而关联服务器，server1会解析这个配置对象，然后向其他成员发送消息，提醒它们使用新的配置。所有成员都配置完成之后，它们会自动选出一个主节点，然后就可以同单机一样，读写请求了

>db = (new Mongo("server1:27017")).getDB("test");

>rs.init

修改副本集配置，重新配置副本集时，主节点需要先退化为普通的备份节点，以便接受新的配置，然后会恢复。

rs.add("server3:27017");

rs.remove("server1:27017");

rs.config();

var config = rs.config();

config.members[1].host = "server4:27017";

rs.reconfig(config)

## 分片

几乎所有数据库软件都能进行手动分片，应用需要维护与若干不同数据库服务器的连接，每个连接还是完全独立的。应用程序管理不同服务器上不同数据的存储，还管理在合适的数据库上查询数据的工作。这种方法可以很好地工作，但是非常难以维护，比如向集群添加节点或从集群删除节点都很困难，调整数据分布和负载模式也不轻松。

MongoDB支持自动分片，可以使数据库架构对应用程序不可见，也可以简化系统管理对应用程序而言，好像始终在使用一个单机的MongoDB服务器一样。MongoDB自动处理数据在分片上的分布，容易添加和删除分片

分片与复制的区别？

复制是让多台服务器都拥有同样的数据副本，每一台服务器都是其他服务器的镜像。而每一个分片都有其他分片拥有不同的数据子集

分片的目标之一：创建一个拥有很多机器的集群，整个集群对应用程序来说就像是一台单机服务器。为了对应用程序隐藏数据库架构的细节，在分片之前要先执行mongos进行一次路由过程。这个路由服务器维护着一个“内容列表”，指明了每个分片包含什么数据内容。应用程序只需要连接到路由服务器，就可以像使用单机服务器一样进行正常的请求了。

### 单台服务器集群

$mongo --nodb

创建3个分片(mongod进程）的集群，分别运行在20001,20002, 20003端口，默认情况下，ShardingTest会在20006端口启动mongos

>cluster = new ShardingTest({"shards": 3, "chunksize": 1})

客户端连接路由服务器Mongos

> db = (new Mongo("localhost:20006")).getDB("test")

mongos会自动将请求路由到合适的分片。客户端不需要知道分片的任何信息，比如分片数据和分片地址。只要有分™片存在，就可以向mongos发送请求，它会自动将请求转发到合适的分片上。

for (var i = 0 ; i < 100000; ++ i) {

db.users.insert({"username": "user"+i, "create\_at": new Date()});

}

>db.users.count()

查询集群的状态：分片摘要信息，数据库摘要信息，集合摘要信息

sh同rs一样，提供分片操作的辅助函数

>sh.help()

>sh.status()

数据库分片

>sh.enableSharding("test")

数据库内的集合分片

对集合分片时，要选择一个片键. MongoDB根据这个键拆分数据。只要被索引过的键才能够作为片键

>db.users.ensureIndex({"username": 1}); 键 -> 索引

>sh.shardCollection("test.users", {"username": 1}) 基于键分片

在分片之前，集合实际上是一个单一的数据块。分片依据片键将集合拆分为多个数据块

查看MongoDB的查询过程

>db.users.find({username: "user12345"}).explain() 定向查询

>db.users.find().explain() 分散-聚集查询

>cluster.stop()

多台服务器集群

通常，分片用来：

* 增加可用RAM
* 增加可用磁盘空间
* 减轻单台服务器的负载
* 处理单个mongod无法承受的吞吐量

由于迁移数据、维护元数据、路由等开销，少量分片的系统与未分片的系统相比，通常延迟更大，吞吐量甚至可能会更小。因此，至少应该创建3个或以上的分片

配置服务器

普通的mongod服务器，保存着集群的配置信息：集群中有哪些分片、分片的是哪些集合、以及数据块的分布

配置服务器相当于集群的大脑，保存着集群和分片的元数据

应该首先建立配置服务器，鉴于它所包含数据的极端重要性，必须启用其日志功能，并确保其数据保存在非易失性驱动器上。每个配置服务器都应位于单独的物理机器上，最好是分布在不同地理位置的机器上

Mongos会向所有配置服务器发送写请求，执行一个两步提交类型的操作，以确保服务器拥有相同的数据，所以这些配置服务器是可的

$mongod --configsvr --dbpath /var/lib/mongodb -f mongod.conf

若没有权限,执行$chown -R $USER:$USER /var/lib/mongodb

$ssh server2

$mongod --configsvr --dbpath /var/lib/mongodb -f /var/lib/config/mongod.conf

--configsvr选项指定mongod为新的配置服务器。

配置服务器并不需要太多的空间和资源。配置服务器的1kb空是约等于200MB真实数据，它保存的只是数据的分布表，由于配置服务器不需要太多的资源，因此可将其部署在运行着其他程序的机器上，如应用服务器、分片的mongod服务器、或mongos进程的服务器上

要常对配置服务器做数据备份。应常在执行集群维护操作之前备份配置服务器的数据

mongos进程

配置服务器均处于运行状态后，启动一个mongos进程供应用程序连接

$mongos --configdb spock/server1:27019, server2:27019 -f /var/lib/mongos.conf

默认情况下mongos运行在27017端口。注意，并不需要指定数据目录(mongos自身并不保存数据，它会在启动时从配置服务器加载集群数据）

可启动任意数量的mongos进程。通常的配置是每个应用程序服务器使用一个mongos进程（与应用服务器运行在同一台机器上）

将副本集转换为分片

该副本集会成为第一个分片

>sh.addShard("spock/server1:27017, server2:27017")

Mongos能够自动检测到没有包含在副本集成员表中的成员。

副本集名称spock被用作分片名称

将副本集作为分片添加到集群后，就可以将应用程序设置从连接到副本集改为连接到mongos.添加分片后，mongos会将副本集内的所有数据库注册为分片的数据库，因此所有查询都会被发送到新的分片上。

增加集群容量

可通过增加分片来增加集群容量。为添加一个新的、空的分片，可先创建一个副本集。确保副本集的名字与其他分片不同。副本集完成初始化并拥有一个主节点后，可以Mongos上运行addShard()命令,将副本集作为分片添加到集群中，在参数中指定副本集的名称和主机名作为种子。

如有多个现存的副本集没有作为分片，只要它们没有同名的数据库，就可将它们作为新分片全部添加到集群中

数据分片

除非明确指定规则,否则MongoDB不会自动对数据进行拆分。如有必要，必须明确告知数据库和集合

>db.enableSharding("music")

>sh.shardCollection("music.artists", {"name": 1})

shardCollection()命令会将集合拆分为多个数据块，这是mongodb迁移数据的基本单元。命令成功执行后，mongodb会均衡地将集合数据分散到集群的分片上。

MongoDB如何追踪集群数据

MongoDB将文档分组为块(chunk)，每个块由给定片键特定范围内的文档组成。一个块只存在于一个分片上，所以MongoDB用一个比较小的表就能够维护块跟分片的映射

进行写操作时，块内的文档数量和大小可能会发生改变

当一个块增长到特定大小时，mongodb会自动将其拆分为两个较小的块。

块范围

新分片的集合起初只有一个块，所有文档都位于这个块中。随着块的增长，mongodb会自动将其分在两个块。

块信息保存在config.chunks集合中。

拆分块过程

Mongos会记录在每个块中插入了多少数据，一旦达到某个阈值，就会检查是否需要对块进行拆分

收到客户端发起的写请求时，Mongos会检查当前块的拆分阈值点

如果达到了拆分阈值点，mongos就会向分片发起一个针对该拆分点的拆分请求

分片计算块的拆分点，并将这些信息发回mongos

Mongos选择一个拆分点，然后试图将这些信息通知给配置服务器

均衡器balancer

负责数据的迁移，它会周期性地检查分片间是否存在不均衡，每隔几秒钟，Mongos就会尝试变身为均衡器。如果没有其他可用的均衡器，mongos就会对整个集群加锁，以防止配置服务器对集群进行修改，然后做一次均衡。均衡并不会影响mongos的正常路由操作，所以使用mongos的客户端不会受到影响

Mongos成为均衡器后，就会检查每个集合的分块表，从而查看是否有分片达到了均衡阈值balancing threshold.

使用集群的应用程序无需知道数据迁移：在数据迁移完成之前，所有读写读请求都会被路由到旧的块上。如果元数据更新完成，那么所有试图访问旧位置数据的mongos进程都会得到一个错误，这些错误对客户端不可见，mongos会对这些错误做静默处理，然后在新的分片上重新执行之前的操作

# PostgreSQL

## install PostgreSQL

### Window

Download and install postgresql-9.5.0-rc1-windows-x64.exe

安装过程会要求设置管理员密码：qzlin （默认管理员:postgre）

数据库默认地址: C:\Program Files\PostgreSQL\9.5\data

Search -> service -> postgresql -> 右击 重启, 停止, …

开放远程连接

C:\PostgreSQL\... data\postgresql.conf

listen\_addresses = ‘\*’ 远程连接

listen\_addresses = ‘localhost’ 本地连接

### Linux

$sudo apt-get install postgresql

//启动postgresql服务器

$sudo /etc/init.d/postgresql start

$sudo /etc/init.d/postgresql stop

### 连接数据库

pgAdmin and psql

pgAdmin是可视化工具，psql是命令行工具

推荐使用psql来连接数据库进行备份和恢复db，同样大小的db，使用psql来restore会比pgAdmin快上数倍！

Open pgAdmin

连接服务器： pgAdmin 4 (localhost:5432) -> 连接服务器

数据库 -> 创建数据库 //创建数据库

登录角色 -> 创建登录角色 //创建登录角色

Right click mimic -> restore - > import sql

cmd -> psql

点击psql （输入server, database, port, usename and password） #window连接方式

or

cmd -> psql –u postgres mimic

$sudo -u postgres psql #linux 连接方式

mimic=# help （假定连接的是mimic数据库）

mimic=# \l #list所有数据库

mimic =# CREATE DATABASE test; #创建数据库

**mimic =# \c test #connect（切换）数据库**

mimic =#\dt #list数据库中的所有表

mimic =#\d admission #查看admission表结构

mimic=#\q #退出

mimic=#SHOW search\_path;

restore

**mimic=#SET search\_path TO mimiciii; #切换当前schema**

mimic=#\cd C:/Users/320029857/Documents/projects/icu/mimic/mimic-code/concepts

mimic=#\i sepsis/angus.sql

or

**mimic=#\i C:/…./durations/ventilation-durations.sql**

增删改

test=# CREATE ROLE rich login; 创建登录角色

test=# GRANT SELECT, INSERT, DELETE, UPDATE ON public.employees TO rich;

test=# CREATE ROLE qzlin LOGIN PASSWORD ‘pass@123’ VALID UNTIL ‘infinity’;

test=# CREATE DATABASE blog;

test=# CREATE TABLE table\_name (

<列名1> <数据类型> <该列所需约束>,

…

);

数据类型：INTEGER, CHAR, VARCHAR, DATE

DROP TABLE table\_name;

ALTER TABLE table\_name ADD COLUMN 列名 数据类型 该列所需约束;

ALTER TABLE table\_name DROP COLUMN 列名;

插入数据

BEGIN TRANSACTION;

INSERT INTO table\_name VALUES (val1, val2, …);

INSERT INTO table\_name (部分列col1, col2, …) VALUES (部分值val1, val2, …);

…

COMMIT;

删除全部数据行

DELETE FROM table\_name;

DELETE FROM table\_name

WHERE sale\_price >= 400;

更新数据行

UPDATE Product

SET sale\_price = sale\_price \* 10,

purchase\_price = purchase\_price / 2,

WHERE product\_type = ‘厨房用具’;

## 查询SQL语法

SELECT x.name, count(\*) AS count\_col 投影

FROM People x

INNER JOIN Country cn ON x.country\_code = cn.country\_code 联接

WHERE x.isactive 行过滤

GROUP BY venue\_id 表分组，然后对这些组执行筛选和聚合

HAVING count(\*) >= 2 AND venue\_id IS NOT NULL; 组过滤

ORDER BY venue\_id, count(\*) 排序

报告顺序：FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY

* SQL语句以分号;结尾
* 注释多行用/\* … \*/, 单行用--
* 字符串或日期用单引号如’abc’, ‘10/01/26’表示

SELECT ‘商品’ AS string, 38 AS number, ‘2009-02-24’ AS date, id, name

FROM Product;

* 使用聚合函数 或GROUP BY时，SELECT子句和HAVING子句只能接常数， 聚合键（即GROUP BY子句中指定的列名）

SELECT MIN(starts), MAX(ends), COUNT(id), SUM(price), AVG(price) 聚合函数

FROM Product;

WHERE starts >= ‘2012-04-01’ 查询4月1日之后发生的所有事件

* ORDER BY子句可以接表中任何列，也可以使用聚合函数
* 视图保存的是SELECT语句，不能使用ORDER BY
* ‘2012-10-25 10:00 AM’::Date 类型转换，等效写法CAST (‘2012-10-25 10:00 AM’ AS Date)

视图

CREATE VIEW view\_name (col1, col2, …)

AS

SELECT …

DROP VIEW view\_name CASCADE; 删除视图及其关联视图

子查询

SELECT type, product

FROM (

SELECT type, COUNT(\*) AS product

FROM Product

GROUP BY type

);

SELECT name, price

FROM Product

WHERE id in (

SELECT id

FROM ShopProduct

WHERE shop\_id = ‘000C’)

);

标量子查询 （仅返回1行1列的结果）

能够 使用常数或者列名的地方，无论是SELECT子句，GROUP BY子句，HAVING子句，还是ORDER BY子名，都可以使用

SELECT id, name, price

FROM Product

WHERE price > (

SELECT AVG(price)

FROM Product

);

SELECT type, AVG(price)

FROM Product

GROUP BY type

HAVING AVG(price) > (

SELECT AVG(price)

FROM Product

);

CASE 表达式

CASE WHEN <求值表达式> THEN <表达式>

WHEN <求值表达式> THEN <表达式>

…

ELSE <表达式>

END

若<求值表达式>为真，则执行THEN子句中的表达式

SELECT name,

CASE WHEN type=’衣服’ THEN ‘A’

WHEN type=’办公用品’ THEN ‘B’

ELSE NULL

END AS product\_type

FROM Product;

--对按照商品种类计算出的销售单价合计

SELECT SUM(CASE WHEN type = ‘衣服’ THEN price ELSE 0 END) AS price\_clothes,

SUM(CASE WHEN type = ‘厨房用具’ THEN price ELSE 0 END) AS price\_kitchen

FROM Product;

集合运算 (以行为单位对表进行运算，运算对象的记录列数必须相同)

UNION 并集

EXCEPT 差集

INTERSECT交集

UNION ALL 包含重复行的集合运算

SELECT id, name

FROM Product

UNION

SELECT id, name

FROM Product2;

联结 （以列为单位对表进行联结）

INNER JOIN内联结

LEFT OUTER JOIN 左联结

RIGHT OUTER JOIN 右联结

FROM table1

INNER JOIN table2

ON table1.id = table2.id

CROSS JOIN 笛卡儿积 (对两张表中的全部记录进行交叉组合)

窗口函数: SUM, AVG, COUNT, MAX, MIN, RANK, DENSE\_RANK, ROW\_NUMBER

窗口函数 OVER (PARTITION BY 列

ORDER BY 列)

通过PARTITION BY分组后的记录集合称为”窗口”

与GROUP BY区别：分组聚合，会使返回记录变少. PARTITION BY并不会减少记录

窗口函数只能在SELECT子句中使用

与ORDER BY区别：OVER子句中的ORDER BY只是用来决定窗口函数按照什么样的顺序进行计算，对结果的排列顺序并没有影响

SELECT name, type, price,

RANK() OVER (ORDER BY price) AS ranking,

DENSE\_RANK() OVER (ORDER BY price) AS dense\_ranking,

ROW\_NUMBER() OVER (ORDER BY price) row\_num

FROM Product;

RANK(): 跳过相同位次的记录，如有3条记录排在第1位时: 1, 1, 1, 4

DENSE\_RANK(): 不跳过相同位次的记录，如有3条记录排在第1位时：1,1,1,2

ROW\_NUMBER(): 连续位次，如有3条记录排在第1位时：1,2,3,4

累计统计方法：

SELECT id, name, price,

SUM(price) OVER (ORDER BY id) AS current\_sum 累计求和

FROM Product;

SELECT id, name, price,

AVG(price) OVER (ORDER BY id) AS current\_avg 累计均值

FROM Product;

移动平均：基于前一行，当前行，后一行的平均值

SELECT id, name, price,

AVG(price) OVER (ORDER BY id

ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS moving\_avg

FROM Product;

小计和合计

ROLLUP: 该 操作能够得到像从小计到合计这样，从最小的聚合级开始，聚合单位逐渐扩大的结果

SELECT type, regist\_date, SUM(price) AS sum\_price

FROM Product

GROUP BY ROLLUP(type, regist\_date);

GROUPING: 用来判断超级分级记录的NULL，该函数在其参数列的值为超级分级记录所产生的NULL时返回1, 其他情况返回0

SELECT CASE WHEN GROUPING(type)=1 THEN ‘商品种类 合计’

ELSE type

END AS type,

CASE WHEN GROUPING(regist\_date)=1 THEN ‘登记日期 合计’

ELSE **CAST(resit\_date AS VARCHAR(16)) --为了满足CASE分支返回值一致**

END AS regist\_date

SUM(price) AS sum\_price

FROM Product

GROUP BY ROLLUP(type, regist\_date);

**数据类型 + 函数(运算符)**

类型置换函数： CAST (val AS TYPE) 等效成val::TYPE

如：CAST (‘2012-10-25 10:00 AM’ AS DATE)

‘2012-10-25 10:00 AM’::DATE

将NULL转换为其他值

COALESCE(val1, val2, val3, …)

数值类型

GENERATE\_SERIES(start, end, step) 类似python的range函数

算术运算符: +, -, \*, /, ABS(val), MOD(被除数,除数), ROUND(val, 保留小数的位数)

比较运算符: =, <>, >=, >, IS NOT NULL, BETWEEN val1 AND val2

逻辑运算符: AND, OR, NOT, NOT IN (val1, val2, val3)

三值逻辑真值表

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | P | Q | P AND Q | | True | True | True | | True | False | False | | **True** | **unknown** | **unknown** | | False | True | False | | False | False | False | | **False** | **unknown** | **unknown** | | **unknown** | **True** | **unknown** | | **unknown** | **False** | **False** | | unknown | unknown | unknown | | |  |  |  | | --- | --- | --- | | P | Q | P OR Q | | True | True | True | | True | False | True | | **True** | **unknown** | **True** | | False | True | True | | False | False | False | | **False** | **unknown** | **unknown** | | **unknown** | **True** | **True** | | **unknown** | **False** | **unknown** | | unknown | unknown | unknown | |

### 字符串

char 固定长度, varchar, text 无固定长度，上限存储空间为1GB

WHERE type = ‘衣服’; #比较字符串

LENGTH(val),

REPEAT(val, n), --类似python的repeat and + 函数

LOWER(val), UPPER(val)

替换：REPLACE( 对象字符串，替换前的字符串，替换后的字符串)

连接字符串: ||

字符串拆分数组：STRING\_TO\_ARRAY

SELECT UNNEST(STRING\_TO\_ARRAY (‘abc.123.zsx’, ‘.’)) AS x; 返回[abc, 123, zsx]

提取子字符串:

SUBSTRING(对象字符串，截取的起始位置，截取的字符数)

SPLIT\_PART(对象字符串，分隔符，第i个元素)

SELECT SPLIT\_PART (‘abc.123.zsx’, ‘.’, 2) AS x; 返回123

填充: LPAD, RPAD

SELECT LPAD (‘ab’, 4, ‘0’) As ab\_lpad, RPAD (‘ab’, 4, ‘0’) AS ab\_rpad;

去空格: RTRIM, LTRIM, TRIM, BTRIM

SELECT a AS a\_before, TRIM(a) AS a\_trim, RTRIM(a) AS a\_rt

FROM (

SELECT REPEAT( ‘ ‘, 4) || i || REPEAT(‘ ‘, 4) AS a,

FROM GENERATE\_SERIES(0, 200, 50) AS I

)

字符串部分一致查询 LIKE

\_（下划线）：表示任意一个字符

%: 表示任意多个字符

正则表达式: ~\* ‘…’

regexp\_replace

SELECT COUNT(\*)

FROM movies

WHERE title !~\* ‘^the.\*’

正则表达式的匹配字串由~运算符开始，可以带可选 的!(表示不匹配)和\*（表示不区分大小写）。^the表示不以the开始的电影（不区分大小写）

全文检索 &&

SELECT title

FROM movies

WHERE title @@ ‘night & day’;

等价于

SELECT title

FROM movies

WHERE to\_tsvector(title) @@ to\_tsquery(‘engligh’, ‘night & day’)

### 时间类型

date, time, timestamp, timestamptz, timetz, interval, strange, tstzrage, daterange

SELECT CURRENT\_DATE; 获得当前日期

SELECT CURRENT\_TIMESTAMP; 获得当前日期和时间

生成时间序列数组

SELECT **(dt – interval ‘1 day’)** AS eom

FROM **generate\_series(‘2/1/2012’, ‘6/30/2012’, interval ‘1 month’)** AS dt;

EXTRACT(日期元素 FROM 日期) 截取日期元素

从日期时间类型中提取部分元素

SELECT CURRENT\_TIMESTAMP,

EXTRACT**(YEAR from** CURRENT\_TIMESTAMP**)** AS year,

EXTRACT**(MONTH from** CURRENT\_TIMESTAMP**)** AS month,

EXTRACT**(DAY from** CURRENT\_TIMESTAMP**)** AS day,

EXTRACT**(HOUR from** CURRENT\_TIMESTAMP**)** AS hour,

EXTRACT**(MINUTE from** CURRENT\_TIMESTAMP**)** AS minute,

EXTRACT**(SECOND from** CURRENT\_TIMESTAMP**)** AS second;

时间运算

SELECT ‘2012-10-25 10:00 AM’::timestamp + interval ‘1 hour’;

SELECT (‘2012-10-25 10:00 AM’::timestamp, ‘2012-10-27 10:00 PM::timestamp) OVERLAPS

(‘2012-10-26 10:00 AM’::timestamp, ‘2012-10-28 10:00 PM::timestamp) AS x,

### 数组: type[]

SELECT ARRAY[2001, 2002, 2003] AS years; 构造数组

SELECT ARRAY(

SELECT DISTINCT date\_part(‘year’, log\_ts)

FROM logs

ORDER BY date\_part(‘year’, log\_ts)

);

SELECT ‘{Ales, Sonia}’::text[] AS name, ’{43, 40}’::smallint[] AS age;

SELECT string\_to\_array(‘ca.ma.tx’, ‘.’) AS estados; 字符串转数组

SELECT array\_agg(log\_ts ORDER BY log\_ts) AS x

FROM logs

WHERE log\_ts BETWEEN ‘2011-01-01’::timestamptz AND ‘2011-01-15’::timestamptz;

引用数组元素

SELECT facts[1] AS first, facts[array\_upper(facts, 1)] AS last

FROM census.department;

SELECT facts[2:4]

FROM census.department;

SELECT facts[1:2] || facts[5:6]

FROM census.department;

SELECT unnest(‘{aaa, bbb, ccc}’::varchar[]) AS tic\_tac\_toe; 数组元素展开为记录行

SELECT \* FROM unnest(‘{a, b}’::text[], ‘{1,2,3}’::int[]) AS f(t, i);

### 区间类型：

int4range, int8range

numrange 用于描述小数，浮点数…

daterange

strange, tstzrange

SELECT ‘[2013-01-05, 2013-08-13]’::daterange;

判定区间是否有重叠： &&

判定区间是否包含值: @>

SELECT …

FROM employment AS e1 INNER JOIN employment AS e2

ON e1.period && e2.period

WHERE e1.employee <> e2.employee

GROUP BY e1.employee;

SELECT employee

FROM employment

WHERE period @> current\_date

GROUP\_BY employee;

## PostgreSQL专用语法

DISTINCT: 将结果集中**完全重复的记录**剔除

SELECT DISTINCT type, date #基于type and date列，去除重复数据

FROM Product;

DISTINCT ON: 将结果集中**指定字段值的重复记录**剔除 (支持基于多列)

SELECT left(tract\_id, 5)) AS county, tract\_id, tract\_name

DISTINCT ON (left(tract\_id, 5))

FROM census.tracts

ORDER BY county, tract\_id

LIMIT: 指定查询时仅返回指定数量的记录

OFFSET: 指定从第几条记录开始返回

ILIKE: 实现不区分大小写的查询

SELECT tract\_name

FROM census.tracts

WHERE tract\_name ILIKE ‘%duke%’

FILTER: 替代ANSI SQL标准语法的CASE WHEN子句

SELECT student,

AVG(CASE WHEN subject=’algebra’ THEN score ELSE NULL END) AS algebra,

AVG(CASE WHEN subject=’physics’ THEN score ELSE NULL END) AS physics

FROM test\_scores

GROUP BY student;

等价于

SELECT student,

AVG(score) FILTER (WHERE subject=’algebra’) AS algebra,

AVG(score) FILTER (WHERE subject=’physics’) AS physics

FROM test\_scores

GROUP BY student;

OVER窗口函数

**窗口：当前行可见的外部记录行的范围**

通过窗口函数可以把当前行的“窗口”区域内的记录的聚合运算结果附加到当前记录行。

SELECT tract\_id, val, AVG(val) **OVER()** as val\_avg

FROM centus.facts

WHERE fact\_type\_id = 86;

OVER子句限定窗口中的可见记录范围。本例中的OVER子句未设定任何条件，因此从该窗口中能看见全表所有记录， 所以AVG运算的是表中所有符合fact\_type\_id=86条件的记录中val字段的平均值

SELECT tract\_id, val, AVG(val) **OVER(PARTITION BY left(tract\_id, 5))** AS val\_avg\_county

FROM census.facts

WHERE fact\_type\_id = 2

ORDER BY tract\_id;

**PARTITION BY设置窗口可见记录范围**

SELECT **ROW\_NUMBER()** **OVER (ORDER BY tract\_name)** AS rnum, tract\_name

FROM census.lu\_tracts

ORDER BY rum LIMIT 4;

SELECT tract\_id, val,

SUM(val) OVER(PARTITION BY left(tract\_id, 5) ORDER BY val) AS sum\_county\_ordered

FROM census.facts

WHERE fact\_type\_id = 2

ORDER BY left(tract\_id, 5), val;

PARTITION BY 和ORDER BY联用，效果是对PARTITION BY指定的记录集进行排序

临时表

WITH cte1 AS (

SELECT tract\_id,

substring(tract\_id, 1, 5) AS county\_code,

COUNT(\*) OVER(PARTITION BY substring(tract\_id, 1, 5)) AS cnt\_tracts

FROM census.lu\_tracts

), cet2 AS (

SELECT MAX(tract\_id) AS last\_tract, county\_code, cnt\_tracts

FROM cte1

WHERE cnt\_tracts < 8

GROUP BY county\_code, cnt\_tracts

)

SELECT c.last\_tract, f.fact\_type\_id, f.val

FROM census.facts AS f INNER JOIN cte2 c ON f.tract\_id=c.last\_tract;

# XNAT

wiki.xnat.org/display/XNAT17/Running+XNAT+in+a+Vagrant+Virtual+Machine

Running XNAT in a Vagrant Virtual Machine

Prerequisites:

* Install Vagrant (1.8.7 or later is recommended)
* Install Git
* Install VirtualBox (5.1 or later is recommended)

$ git clone --branch master https://bitbucket.org/xnatdev/xnat-vagrant.git && cd xnat-vagrant && ./run xnat setup

Python接口

pythonhosted.org/pyxnat/