Abstract (database part)

15. MySQL Optimization I

> **Objectives**
> – 能够根据数据访问的具体场景，设计数据库，包括数据库结构和索引

1. Optimization Overview(P3-6)
    a. Optimizing at the Database Level
    b. Optimizing at the Hardware Level
2. **Indexes(P7**
    a. 索引的定义&并不是越多索引越好（P7）
    b. **How MySQL Uses Indexes（P8-10）**
    c. **Primary Key Optimization（P11）**
    d. SPATIAL Index Optimization（不常用，不重要）**（P12）**
    e. **Foreign Key Optimization（P13）**
    f. **Column Indexes**
        i. **Index Prefixes（P14）**
        ii. FULLTEXT Indexes（不重要）（P15）
        iii. Spatial Indexes（不重要）（P16）
    g. **Multiple-Column Indexes**
        i. overview（P17）
        ii. MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index（P20）
        iii. 一些example (P18-20)
    h. **Comparison of B-Tree and Hash Indexes**

i. **B-Tree Index Characteristics（P21）**

ii. example（P22）

iii. **Hash Index Characteristics（P23）**

i. **Descending Indexes（P24-25）**

3. **Optimizing Database Structure（P26）**
4. **Optimizing Data Size（P27-34）**
   a. **Table Columns（P28）**
      - 使用更高效的（更小的）数据类型。*eg*：*INT→MEDIUMINT*

        eg：INT \rightarrow MEDIUMINT
      - 尽可能将 *Column* 声明为 *NOT NULL*。

   b. **Row Format（P29-30）**
      - 使用 *DYNAMIC* 为默认。
      - 可以使用 *REDUNDANT* , *COMPACT* , *DYNAMIC*, *COMPARESSED* 来减少使用空间。

   c. **Indexes（P31-32）**
      - *primary index* 应该尽可能短。可以仅索引不同部分，后缀相同的部分不需要，可以仅索引独一无二的前缀部分。
      - 只有需要频繁查询的地方才需要建立索引。频繁插入和更新的地方慎重使用。

   d. **Joins（P33）**
      - 一个经常 *scan* 的表格可以拆分成两个表。
      - 不同表格中相同信息的 *column* 应该被声明为相同数据类型。
      - 保持 *column* 名字简单，方便跨表格使用。（简单名字，尽量不超过 18 个字符）

   e. **Normalization（范式化）（P34）**
      - 保持所有信息非冗余。
      - 为了提高速度，可以适当放弃精简冗余信息。

5. **Optimizing MySQL Data Types**
   a. **Optimizing for Numeric Data（P35）**

b. **Optimizing for Character and String Types（P36-37）**

- 使用 *binary collation order* 在快速比较和排序的操作上。
- 尽可能比较数据类型相同的列，减少类型转换。
- 对于少于 8 KB的 *column* 使用 *binary VARCHAR* 而不是 *BLOB*。
- 如果表包含字符串列，例如名称和地址，但许多查询不检索这些列
  - 考虑将字符串列拆分为单独的表，并在必要时使用带有外键的连接查询。
  - 当 MySQL 从行中检索任何值时，它会读取包含该行所有列的数据块（以及可能的其他相邻行）。 保持每行较小，只包含最常用的列，这样可以在每个数据块中容纳更多行。
  - 这种紧凑的表减少了常见查询的磁盘 I/O 和内存使用。
- 当使用随机生成的值作为 InnoDB 表中的主键时，尽量在其前面加上一个升序值，例如当前日期和时间。 当连续的主值物理存储在彼此附近时，InnoDB 可以更快地插入和检索它们。

c. **Optimizing for BLOB Types（P38）**

- 先考虑压缩。
- 对于具有多个列的表，为了减少不使用 BLOB 列的查询的内存需求，考虑将 BLOB 列拆分为单独的表，并在需要时使用连接查询引用它。
- 将 BLOB 特殊的表放在不同的存储设备甚至单独的数据库实例上。
- 将列值的散列存储在单独的列中，索引该列，然后在查询中测试散列值，而不是针对非常长的文本字符串测试相等性。

6. **Optimizing for Many Tables**

a. 实际打开的表比用户自己打开的多的原因（P40）

b. table_open_cache（P41）

c. MySQL closes an unused table and removes it from the table cache 的情景（P42）

d. When the table cache fills up, the server uses the following procedure to locate a cache entry to use（P43）

e. Disadvantages of Creating Many Tables in the Same Database（P45）

f. internal temporary tables（临时表）使用的情景（P46）

7. **Limits on Number of Databases and Tables（P47）**
8. **Limits on Table Size（P48）**
    a. full-table error（P49）
9. **Column Count Limits（P50）**
10. **Row Size Limits（P51）**
    a. **Row Size Limits Examples（P52-56）**

16. MySQL Optimization II

> **Objectives**
> – 能够根据数据访问的具体场景，设计数据库性能调优方案，包括索引优化、缓存设置和参
> 数调优等

1. **Optimizing for InnoDB Tables（P3-18)（另一份笔记有更详细的叙述）**

    1 Optimizing Storage Layout for InnoDB Tables
    2 Optimizing InnoDB Transaction Management
    3 Optimizing InnoDB Transaction Management
    4 Optimizing InnoDB Redo Logging
    5 Bulk Data Loading for InnoDB Tables
    6 Optimizing InnoDB Queries
    7 Optimizing InnoDB DDL Operations
    8 Optimizing InnoDB Disk I/O
    9 Optimizing InnoDB Configuration Variables
    10 Optimizing InnoDB for Systems with Many Tables

    中文翻译参考：Mysql优化（出自官方文档） - 第十篇（优化InnoDB表篇） (bbsmax.com)
2. **Optimizing for MEMORY Tables（P25）**
    a. 考虑使用hash还是B-tree index
3. InnoDB Buffer Pool Optimization(P26-32)
    - Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)

- Configuring Buffer Pool Flushing
- Making the Buffer Pool Scan Resistant
- Configuring Multiple Buffer Pool Instances
- Saving and Restoring the Buffer Pool State
- Configuring InnoDB Buffer Pool Size

参考：MySql 中文文档 - 14.8.3 InnoDB 缓冲池配置 | Docs4dev

17. MySQL Backup & Recovery

> **Objectives**
> – 能够根据数据访问的具体场景，设计提高数据库访问性能和灾备能力的方案，包括集群部署和备份机制

1. **Backup and Recovery overview（P3）**
   a. why It is important to back up your databases
2. **Backup and Recovery Types(P4-14）**
   a. 什么是Physical backups（P4）
   b. 什么是Logical backups（P4）
   c. Physical backup characteristics: 优点(P5) 缺点（P6）
   d. Logical backup characteristics（P7）
   e. Online & Offline Backups (P8-9)
   f. Local & Remote Backups(P10)
   g. Snapshot Backups(P11)
   h. Full & Incremental Backups(P12)
   i. **Full Point-in-Time (Incremental) Recovery的方法（P13）**
   j. Backup Scheduling, Compression, and Encryption（P14）
3. **Database Backup Methods（P15-**
   a. Making a Hot Backup with MySQL Enterprise Backup（P15）

b. Making Backups with mysqldump（P16）
c. Making Backups by Copying Table Files（P16）
d. Making Delimited-Text File Backups（P17）
e. Making Incremental Backups by Enabling the Binary Log(P18)
f. Making Backups Using Replicas(P19)
g. Recovering Corrupt Tables(P20)
h. Making Backups Using a File System Snapshot(P20)

4. **Example Backup and Recovery Strategy(P20**
   a. 讨论的情况及分析（P22）：
      i. Operating system crash
      ii. Power failure
      iii. File system crash
      iv. Hardware problem (hard drive, motherboard, and so forth)
   b. 一个建立backup&recovery的例子（P23-26）
   c. Backup Strategy Summary（P27）
5. **Using mysqldump for Backups（P28-**
   a. A dump file的用途（P28）
   b. two types of output（P29）
   c. How to make a copy of a database（P35）
   d. Copy a Database from one Server to Another(P36)
   e. Dumping Stored Programs(P37)
   f. Dumping Table Definitions and Content Separately(P38)
   g. Using mysqldump to Test for Upgrade Incompatibilities(P39)
6. **Point-in-Time (Incremental) Recovery（P41-48）**
   a. 比较详细的说明，简洁的例子可以看P13

As an example, suppose that around 20:06:00 on March 11, 2020, an SQL statement was executed that deleted a table.

- You can perform a point-in-time recovery to restore the server up to its state right before the table deletion.

These are some sample steps to achieve that: 最近的全量备份

- Restore the last full backup created before the point-in-time of interest (call it t_p, which is 20:06:00 on March 11, 2020 in our example). When finished, note the binary log position up to which you have restored the server for later use, and restart the server.
- Find the precise binary log event position corresponding to the point in time up to which you want to restore your database.
  - In our example, given that we know the rough time where the table deletion took place (t_p), we can find the log position by checking the log contents around that time using the **mysqlbinlog** utility.
  - Use the --start-datetime and --stop-datetime options to specify a short time period around t_p, and then look for the event in the output. 找到 delete 表的语句跳过这一句，执行前后的

三种log补充：

什么是binlog

binlog用于记录数据库执行的写入性操作(不包括查询)信息，以二进制的形式保存在磁盘中。binlog是mysql的逻辑日志，并且由Server层进行记录，使用任何存储引擎的mysql数据库都会记录binlog日志。

可以简单理解为：存储着每条变更的SQL语句（可能不止SQL，还有XID「事务Id」等等）

binlog是通过追加的方式进行写入的，可以通过max_binlog_size参数设置每个binlog文件的大小，当文件大小达到给定值之后，会生成新的文件来保存日志。

用途

主要有两个作用：`复制` 和 `恢复数据`

- MySQL使用一主多从结构时，从服务器需要与主服务器的数据保持一致，通过binlog来实现的
- 可以通过binlog对数据进行恢复

redo log是在innodb即 `存储引擎层` 产生的，物理日志。

redo log包括两部分：一个是内存中的日志缓冲(redo log buffer)，另一个是磁盘上的日志文件(redo log file)。mysql每执行一条DML语句，先将记录写入redo log buffer，后续某个时间点再一次性将多个操作记录写到redo log file。这种先写日志，再写磁盘的技术就是MySQL里经常说到的WAL(`Write-Ahead Logging`) 技术。

在计算机操作系统中，用户空间(user space)下的缓冲区数据一般情况下是无法直接写入磁盘的，中间必须经过操作系统内核空间(kernel space)缓冲区(OS Buffer)。因此，redo log buffer写入redo log file实际上是先写入OS Buffer，然后再通过系统调用fsync()将其刷到redo log file中，过程如下：

功能

redo log是为持久性、恢复数据用的，而binlog不仅可以用于恢复数据，还用于主从复制。

redo log是一个环形结构，是有限的，数据刷到磁盘后redo log就无效，如果redo log满了，mysql还得停一下来刷新数据到磁盘，也就是刷脏页。

redo log	binlog	
文件大小	redo log的大小是固定的。	binlog可通过配置参数max_binlog_size设置每个binlog文件的大小。
实现方式	redo log是InnoDB引擎层实现的，并不是所有引擎都有。	binlog是Server层实现的，所有引擎都可以使用binlog日志
记录方式	redo log 采用循环写的方式记录，当写到结尾时，会回到开头循环写日志。	binlog 通过追加的方式记录，当文件大小大于给定值后，后续的日志会记录到新的文件上
适用场景	redo log适用于崩溃恢复(crash-safe)	binlog适用于主从复制和数据恢复

undo log

undo log主要有两个作用：`回滚`和`多版本控制(MVCC)`

在数据修改的时候，不仅记录了redo log，还记录undo log，如果因为某些原因导致事务失败或回滚了，可以用undo log进行回滚（保证了原子性）

undo log主要存储的是`逻辑日志`，用来回滚的`相反操作日志`。比如我们要insert一条数据了，那undo log会记录的一条对应的delete日志。我们要update一条记录时，它会记录一条对应相反的update记录。

因为undo log存储着修改之前的数据，相当于一个前版本，MVCC实现的是读写不阻塞，读的时候只要返回前一个版本的数据就行了。

参考：(80条消息) Mysql的各种log（binlog、redo log、undo log）_CodeMan22的博客-CSDN博客_mysql 几种log

18. MySQL Partitioning

> **Objectives**
> –能够根据数据访问的具体场景，设计数据库分区方案，并能够对分区数据进行有效管理和控制

1. **Overview of Partitioning in MySQL（P3-6）**
   a. horizontal partitioning
   b. advantages of partitioning（P6）

> 分区使在一个 table 中存储的数据比单个磁盘或文件系统分区中存储的数据更多。
>
> 通常，通过删除仅包含该数据的一个或多个分区，可以轻松地从分区 table 中删除失去其用途的数据。相反，在某些情况下，通过添加一个或多个用于专门存储该数据的新分区，可以大大简化添加新数据的过程。
>
> 由于满足给定 `WHERE` 子句的数据只能存储在一个或多个分区上，因此可以大大优化某些查询，这会自动从搜索中排除任何剩余的分区。由于可以在创建分区 table 后更改分区，因此您可以重新组织数据以增强在首次设置分区方案时可能不经常使用的频繁查询。排除不匹配分区(因此包含的任何行)的能力通常称为分区修剪。有关更多信息，请参见第 22.4 节“分区修剪”。

另外，MySQL 支持显式的分区选择查询。例如，选择*从 t 分区(p0，p1)中 c <5 仅选择分区 `p0` 和 `p1` 中与 `WHERE` 条件匹配的那些行。在这种情况下，MySQL 不会检查 table `t` 的任何其他分区；当您已经知道要检查的分区时，这可以大大加快查询速度。数据修改语句DELETE，INSERT，REPLACE，UPDATE和LOAD DATA，LOAD XML也支持分区选择。有关更多信息和示例，请参见这些语句的描述。

2.2 表分区有如下优点：
1）与单个磁盘或文件系统分区相比，可以存储更多的数据。
2）对于那些已经失去保存意义的数据，通常可以通过删除与那些数据有关的分区，很容易地删除那些数据。
相反地，在某些情况下，添加新数据的过程又可以通过为那些新数据专门增加一个新的分区，来很方便地实现。
同样的，你可以很快的通过删除分区来移除旧数据。你还可以优化、检查、修复个别分区。
3）一些查询可以得到极大的优化。 可以把一些归类的数据放在一个分区中，可以减少服务器检查数据的数量加快查询。
这主要是借助于满足一个给定WHERE语句的数据可以只保存在一个或多个分区内，这样在查找时就不用查找其他剩余的分区。
PS：因为分区可以在创建了分区表后进行修改，所以在第一次配置分区方案时还不曾这么做时，可以重新组织数据，来提高那些常用查询的效率。
4）涉及到例如SUM()和COUNT()这样聚合函数的查询，可以很容易地进行并行处理。

2、劣势

限制暂且归位劣势。

- 一个表最多只能有1024个分区（mysql5.6之后支持8192个分区）。
- 在mysql5.1中分区表达式必须是整数，或者是返回整数的表达式，在5.5之后，某些场景可以直接使用字符串列和日期类型列来进行分区（使用varchar字符串类型列时，一般还是字符串的日期作为分区）。
- 如果分区字段中有主键或者唯一索引列，那么所有主键列和唯一索引列都必须包含进来，如果表中有主键或唯一索引，那么分区键必须是主键或唯一索引。
- 分区表中无法使用外键约束。
- mysql数据库支持的分区类型为水平分区，并不支持垂直分区，因此，mysql数据库的分区中索引是局部分区索引，一个分区中既存放了数据又存放了索引，而全局分区是指的数据库放在各个分区中，但是所有的数据的索引放在另外一个对象中
- 目前mysql不支持空间类型和临时表类型进行分区。不支持全文索引。

2. **Partitioning Types（P7）**
    a. **RANGE Partitioning（P9）**
    b. **LIST Partitioning（P16）**
    c. **COLUMN Partitioning（P21-23）**
    d. **HASH Partitioning（P35）**
        i. **LINEAR HASH Partitioning（P38-39）（比较复杂的数学，没细看）**
        ii. **KEY Partitioning（P41）**

- RANGE分区：基于属于一个给定连续区间的列值，把多行分配给分区。
- LIST分区：类似于按RANGE分区，区别在于LIST分区是基于列值匹配一个离散值集合中的某个值来进行选择。
- HASH分区：基于用户定义的表达式的返回值来进行选择的分区，该表达式使用将要插入到表中的这些行的列值进行计算。这个函数可以包含MySQL 中有效的、产生非负整数值的任何表达式。
- KEY分区：类似于按HASH分区，区别在于KEY分区只支持计算一列或多列，且MySQL 服务器提供其自身的哈希函数。必须有一列或多列包含整数值。

3. **Subpartitioning（P42-43）**
4. **How MySQL Partitioning Handles NULL（P44-54）**

mysql 不禁止在分区键值上使用 null， 分区键可能是一个字段或者一个用户定义的额表达式。一般情况下，mysql 的分区把 null 当作零值，或者一个最小值进行处理。
range 分区中，null 值会被当作最小值来处理；
list 分区中，null 值必须出现在枚举列表中，否则不被接受；
hash/key 分区中，null 值会被当作零值来处理。

5. **Partition Management（P55-**
    a. **Management of RANGE and LIST Partitions（P57-58）**
    b. **Management of HASH and KEY Partitions（P65）**
6. **Exchanging Partitions and Subpartitions with Tables（P66-70）**

3.1 常见使用场景

- 当数据量很大(过T)时，肯定不能把数据再如到内存中，这样查询一个或一定范围的item是很耗时。另外一般这情况下，历史数据或不常访问的数据占很大部分，最新或热点数据占的比例不是很大。这时可以根据有些条件进行表分区。
- 分区表的更易管理，比如删除过去某一时间的历史数据，直接执行truncate，或者狠点drop整个分区，这比detele删除效率更高
- 当数据量很大，或者将来很大的，但单块磁盘的容量不够，或者想提升IO效率的时候，可以把没分区中的子分区挂载到不同的磁盘上。
- 使用分区表可避免某些特殊的瓶颈，例如Innodb的单个索引的互斥访问..
- 单个分区表的备份很恢复会更有效率，在某些场景下

总结：可伸缩性，可管理性，提高数据库查询效率。

3.2 业务场景举例

项目中需要动态新建、删除分区。如新闻表，按照时间维度中的月份对其分区，为了防止新闻表过大，只保留最近6个月的分区，同时预建后面3个月的分区，这个删除、预建分区的过程就是分区表的动态管理。

优缺点

优点

- 可以让单表存储更多的数据。
- 分区表的数据更容易维护，可以通过清除整个分区批量删除大量数据，也可以增加新的分区来支持新插入的数据。另外，还可以对一个独立分区进行优化，检查和修复等操作。

- 通过跨多个磁盘来分散数据查询，来获得更大的查询吞吐量。
- 分区表的数据还可以分布在不同的物理设备上，从而高效利用多个硬件设备。
- 可以使用分区表来避免某些特殊瓶颈，例如 InnoDB 单个索引的互斥访问、ext3 文件系统的 inode 锁竞争。
- 可以备份和恢复单个分区。

缺点

- 一个表最多只能有 1024 个分区。
- 如果分区字段中有主键或唯一索引的列，那么所有主键列和唯一索引列都必须包含进来。
- 分区表无法使用外键约束。
- `NULL` 值会使分区过滤无效。
- 所有分区必须使用相同的存储引擎。
- MySQL 数据仅支持水平分区，并不支持垂直分区。
- 目前 MySQL 不支持空间类型和临时表类型进行分区。不支持全文索引。

19. NoSQL & MongoDB

> ## Objectives
>
> – 能够根据数据特性，设计综合运用NoSQL数据库和关系型数据库的数据存储
> 方案，以实现数据访问性能的优化
> – 能够通过分层架构设计并实现跨类型数据存储机制下的数据访问

1. 对于大数据：read and write data in parallel to or from multiple disks
    a. 出现的问题：hardware failure sol: replication
    b. most analysis tasks need to be able to combine the data in some way sol: MapReduce
2. 关系型数据库的问题（P8-15）
    a. lock
    b. split tables horizontally，不支持split tables vertically

c. Partition the data within a single table based on rows

d. For updating the majority of a database, a B-Tree is less efficient

e. 不好拆分

f. 三种数据类型：*Structured data/Semi-structured data/Unstructured data*

3. MongoDB features（P19)
4. **MongoDB specification(P20)**
5. **Basic concept of MongoDB（P21）**
   a. **Document（P22-24）**
   b. **Collection（P25-27）**
      i. Why *should* we use more than one collection（P26）
   c. **Database（P28-29）**

MongoDB 的逻辑结构是一种层次结构。主要由：
文档(document)、集合(collection)、数据库(database)这三部分组成的。逻辑结构是面向用户的，用户使用 MongoDB 开发应用程序使用的就是逻辑结构。
（1）MongoDB 的文档（document），相当于关系数据库中的一行记录。
（2）多个文档组成一个集合（collection），相当于关系数据库的表。
（3）多个集合（collection），逻辑上组织在一起，就是数据库（database）。
（4）一个 MongoDB 实例支持多个数据库（database）。

下表是MongoDB与MySQL数据库逻辑结构概念的对比

MongoDb	关系型数据库Mysql
数据库(databases)	数据库(databases)
集合(collections)	表(table)
文档(document)	行(row)

6. **Querying（P67-70）**
7. **Indexing（P71-74）**
8. **Aggregation（聚合工具）（P75）**
9. **Sharding（P82-83）**
   a. **When to Shard ?（P85）**
   b. **Shard Keys（P86）**

10. **Chunks（P87）**

四、适用哪些场景

```
更高的写负载
不可靠环境保证高可用
数据量超大规模，大尺寸，低价值的数据
基于位置的数据查询
非结构化数据的爆发增长
常用的场景包括
        Web应用程序
        敏捷开发
        分析和日志(目标原子更新，定长集合)
        缓存
        可变Schema
```

五、不适用哪些场景

```
高度事务性，强一致性业务系统(银行，证券等)
传统商业智能应用
极为复制的业务逻辑查询
```

非关系型数据库的优势：

1. 非关系型数据库存储数据的格式可以是 key-value 形式、文档形式、图片形式等。使用灵活，应用场景广泛，而关系型数据库则只支持基础类型。

2. 速度快，效率高。 NoSQL 可以使用硬盘或者随机存储器作为载体，而关系型数据库只能使用硬盘。

3. 海量数据的维护和处理非常轻松，成本低。

4. 非关系型数据库具有扩展简单、高并发、高稳定性、成本低廉的优势。

5. 可以实现数据的分布式处理。

非关系型数据库存在的不足：

1. 非关系型数据库暂时不提供 SQL 支持，学习和使用成本较高。

2. 非关系数据库没有事务处理，无法保证数据的完整性和安全性。适合处理海量数据，但是不一定安全。

3. 功能没有关系型数据库完善。

4. 复杂表关联查询不容易实现。

关系型数据库的优势：

1. 采用二维表结构非常贴近正常开发逻辑（关系型数据模型相对层次型数据模型和网状型数据模型等其他模型来说更容易理解）；

2. 支持通用的SQL（结构化查询语言）语句；

3. 丰富的完整性大大减少了数据冗余和数据不一致的问题。并且全部由表结构组成，文件格式一致；

4. 可以用SQL句子多个表之间做非常繁杂的查询；

5. 关系型数据库提供对事务的支持，能保证系统中事务的正确执行，同时提供事务的恢复、回滚、并发控制和死锁问题的解决。

6. 数据存储在磁盘中，安全可靠。

关系型数据库存在的不足：

> **随着互联网企业的不断发展，数据日益增多，因此关系型数据库面对海量的数据会存在很多的不足。**

1. 高并发读写能力差：网站类用户的并发性访问非常高，而一台数据库的最大连接数有限，且硬盘 I/O 有限，不能满足很多人同时连接。

2. 海量数据情况下读写效率低：对大数据量的表进行读写操作时，需要等待较长的时间等待响应。

3. 可扩展性不足：不像web server和app server那样简单的添加硬件和服务节点来拓展性能和负荷工作能力。

4. 数据模型灵活度低：关系型数据库的数据模型定义严格，无法快速容纳新的数据类型（需要提前知道需要存储什么样类型的数据）。

20. Neo4J & Graph Computing

> ## Objectives
>
> – 能够根据数据特性和数据访问模式，识别适合图数据库存储的数据，设计并实现
> 其在图数据库中的存储和访问方案

1. **Graph Databases overview(P4)**
2. **The Labeled Property Graph Model(P11)**

- *nodes*, *relationships*, *properties*, and *labels*

3. **Querying Graphs: Cypher（P12）**

   参考：图数据库查询语言Cypher - 知乎 (zhihu.com)

4. **A Comparison of Relational and Graph Modeling（P15-18）（还需要重新看回放）**
5. **Describe the Model in Terms of the Application's Needs（P20）**
6. **Nodes for Things, Relationships for Structure（P21）**
7. **Embedded v.s. Server(P24）**
   a. In embedded mode, Neo4j runs in the same process as our application
8. **Cluster（P24）**
9. **how Neo4j stores graph data: node & relationship example(P28-29)**
10. 图计算理论（P41-52）

图数据库的优点：

1. **关系型数据库不擅长处理数据之间的关系，而图数据库在处理数据之间关系方面灵活且高性能**
2. **数据之间的关系越来越重要**
3. **使用图的方式表达现实世界中的很多事物更直接，更直观，也更易于理解**
4. **专业的图分析算法为实际场景提供解决方案**

图数据库在处理关联关系上具有完全的优势，特别是在我们这个社交网络得到极大发展的互联网时代。例如我们希望知道谁LIKES（喜欢）谁（喜欢可以是单向或双向），也想知道谁是谁的FRIEND_OF（朋友），谁是所有人的LEADER_OF（领导）。除了在关联查询中尤为明显的优越性，图数据库还有如下优势：
a) 用户可以面向对象的思考，用户使用的每个查询都有显式语义；
b) 用户可以实时更新和查询图数据库；
c) 图数据库可以灵活应对海量的关系变化，如增加删除关系、实体等；
d) 图数据库有利于实时的大数据挖掘结果可视化。

图数据库虽然弥补了很多关系型数据库的缺陷，但还有一些不足地方，如：
a) 不适合记录大量基于事件的数据（例如日志条目）；
b) 二进制数据存储。
c) 并发性能要求高的项目。
d) 目前相关图查询语言比较多，尚未有很好统一。
e) 图数据库相关的一些书籍文档偏少。
f) 相关生态还在不断完善。

21. Timeseries Database

> ## Objectives
>
> – 能够根据数据特性和数据访问模式，识别适合时序数据库存储的数据，设计并实
> 现其在时序数据库中的存储和访问方案

1. **time series database overview(P3)**
2. **Why is a time series database important now(P4)**
3. **InfluxDB（P6-**
    a. **InfluxDB key concepts(P8-12）**
        a. Timestamp
        b. Measurement
        c. Fields
        d. Tags
        e. Why your schema matters(P13-14)
        f. Bucket schema(P15)
        g. Series(P16)
        h. Bucket(P17)
        i. Organization(P17)

概念	MySQL	InfluxDB
数据库（同）	database	database
表（不同）	table	measurement（测量; 度量）
列（不同）	column	tag(带索引的，非必须)、field(不带索引)、timestemp(唯一主键)

2.Influxdb相关名词

> database：数据库；
> measurement：数据库中的表；
> points：表里面的一行数据。
> influxDB中独有的一些概念：Point由时间戳（time）、数据（field）和标签（tags）组成。

Point相当于传统数据库里的一行数据，如下表所示：

Point属性	传统数据库中的概念
time（时间戳）	每个数据记录时间，是数据库中的主索引(会自动生成)
fields（字段、数据）	各种**记录值（没有索引的属性）也就是记录的值**：温度， 湿度
tags（标签）	各种**有索引的属性**：地区，海拔

注意

在influxdb中，**字段必须存在。**因为字段是没有索引的。如果使用字段作为查询条件，会扫描符合查询条件的所有字段值，性能不及tag。类比一下，fields相当于SQL的没有索引的列。
tags是可选的，但是强烈建议你用上它，因为tag是有索引的，tags相当于SQL中的有索引的列。**tag value只能是string类型。**

Serices包含**series**以及**series key**两个概念。

首先，一个**series key**是一系列点的集合，这些数据点共享同一个measurement、tag set以及field key。例如，示例数据中包含下面两个不同的**series key**：

_measurement	tag set	_field
census	location=klamath,scientist=anderson	bees
census	location=portland,scientist=mullen	ants

换句话说，一个**series key**所包含数据有相同的measurement、tag set以及field key，这些数据最大的不同就是field value。这这一组相同的measurement、tag set以及field key就组成了一个series的**series key**。

而**series**是一个series key对应的数据序列，序列中的数据包含 `timestamp` 以及 `field value` 。下面是一对series key - series的例子：

参考：(80条消息) InfluxDB详解_顺其自然~的博客-CSDN博客_influxdb

b. **InfluxDB design principle（P18-20）**

   a. Time-ordered data

   b. Strict update and delete permissions

   c. Handle read and write queries first

   d. Schemaless design

   e. Datasets over individual points

   f. Duplicate data

Time-ordered data

为提升性能，数据以时间升序的顺序写入。

Strict update and delete permissions

为增加查询和写入性能，InfluxDB严格限制了**更新**和**删除**操作的权限。它所写入的时序数据几乎都是不会修改的最新的数据。因此**更新**和**删除**这两个动作在时序数据库中显得有些特殊。

Handle read and write queries first

相对于强一致性而言，InfluxDB会优先处理读写请求。任何**影响查询数据**的事务执行优先级都是靠后的，以确保数据的最终一致性。例如，我们写入数据的频率特别高，每毫秒要写入多条数据，那么在写数据过程中读数据，就有可能读不到最新的数据。

Schemaless design

InfluxDB使用"schemaless"的设计来更好的管理断断续续的数据。例如，一个程序运行几十分钟然后结束了，我们所记录的数据也就在这几十分钟范围内。

Datasets over individual points

通常讲，时序数据集整体比单个点的数据要重要。InfluxDB实现了强大的工具来聚合数据和处理大型数据集。而每条数据通过**timestamp**以及**series**来区分，所以InfluxDB中没有传统场景中的IDs（或者理解为主键）这一概念。

Duplicate data

为简化冲突的解决和提高写性能，InfluxDB对**相同的point**不会存储两次。如果某个point的一个新的field value被提交，InfluxDB会将该point对应的field value设置为最新的那一个。在极少数情况下数据可能会被覆盖。关于重复数据的更多信息看这里。

c. **InfluxDB storage engine(P21-**
   a. goal(P21)
   b. Writing data from API to disk(P22)
   c. Write Ahead Log (WAL)(P23)
   d. Cache(P24)
   e. Time-Structured Merge Tree (TSM)(P25)
   f. Time Series Index (TSI)(P26)
   g. InfluxDB file structure(P27-28)

d. **InfluxDB shards and shard groups(P29)**
   a. Shard group duration(P30)

b. Shard group diagram(P31)

c. create/deletion/insert(P32-34)

二. shard

shard是InfluxDB存储引擎的实现，负责数据的编码存储、读写服务等。将InfluxDB中时间序列化的数据按照时间的先后顺序存入到shard中，每个shard中都负责InfluxDB中一部分的数据存储工作，并以tsm文件的表现形式存储在物理磁盘上，每个存放了数据的shard都属于一个shard group。

三. shard group

shard group可以理解为存放shard的容器，所有的shard在逻辑上都属于这个shard group，每个shard group中的shard都有一个对应的时间跨度和过期时间，每一个shard group都有一个默认的时间跨度，叫做shard group duration

Retention Policy’s DURATION	Shard Group Duration
< 2 days	1 hour
>= 2 days and <= 6 months	1 day
> 6 months	7 days

三. 三者之间的关系

在一个RP中，如果指定的保留时间为24小时，那么每个shard的duration为1小时，即每个shard的时间跨度为1小时，那么总共会有24个跨度为1小时的shard，在触发数据的RP后，删除最早时间跨度的shard。

补充：

对于时序数据，我们总结了以下特点：

1.数据特点：数据量大，数据随着时间增长，相同维度重复取值，指标平滑变化（某辆车的某个设备上传上来平滑变化的轨迹坐标）。

2.写入特点：高并发写入，且不会更新（轨迹不会更新）。

3.查询特点：按不同维度对指标进行统计分析，存在明显的冷热数据，一般只会查询近期数据（一般我们只会关心近期的轨迹数据）。

针对特点可以有如下改进：

第一个特点，数据量大，相同维度重复取值，我们可以将这些相同的维度压缩存储（因为是重复的），减少存储成本，比如将重复的host和port只存储一份就好了。

第二个特点，高并发写入，和hbase一样，我们可以采用LSM代替B树

第三个特点，聚合以及冷热数据，我们可以对于冷数据降低精度存储，也就是对历史数据做聚合，节省存储空间。

1. 时序数据库的特点

- 数据写入
  时序数据会按照指定的时间粒度持续写入，支持实时、高并发写入，无须更新或删除操作。
- 数据读取
  写多读少，多时间粒度、指定维度读取，实时聚合。
- 数据存储
  按列存储，通过查询特征发现时序数据更适合将一个指标放在一起存储，任何列都能作为存储，读取数据时只会读取所需要的维度所在的列；以不同时间粒度存储，将最近时间以一个比较细的粒度存储，可以将历史数据聚合成一个比较粗的粒度。

influxDb 是一个高性能的时序数据库，主要特性：

- 专为时序数据设计的高性能数据仓储，TSM引擎可以实现高吞吐速度与数据压缩率；
- 完全使用go编写并且被编译为一个独立的二进制文件，没有任何其它外部依赖；
- 简单、性能优良的http API；
- 通过插件可以实现对其它数据库协议的接入，如Graphite, collectd, and OpenTSDB；
- 为方便数据聚合查询而定制设计的类似于SQL的查询语言；
- Tags使得series能够被迅速而高效地检索；
- 可以高效地淘汰过期数据的保留策略；
- 连续查询（continuous queries）功能可以自动定时聚合数据，使得需要频繁执行的检索更加高效。

二、表设计原则

作为一款schema-less的数据库，influxdb有着不同于mysql的表设计原则。

建议的原则

把元数据（meta data）设计进tag中

tags会被索引，但是fields不会，因此基于tags的检索会比基于fields的检索会更快。

这里有几条通用的用来判断是使用tag还是field来存储数据的方法：

- 如果字段是经常被作为检索条件的元数据，设计为tag；
- 如果字段经常要作为group by的参数，设计为tag；
- 如果字段需要被用来放在influxQL的函数的参数，设计为field；
- 如果出于某些考虑，字段不方便作为string类型保存，则应该使用field，因为tags总是以string类型来存储。

23. Clustering

> ## Objectives
>
> – 能够根据业务需求和系统被访问的模式，设计合理的集群部署方式和负载均衡策略

1. A large-scale system的特点（P3）
2. Essential requirements on large-scale systems（P4）
    a. RAS的区别和联系

RAS - Reliability, Availability and Serviceability

Reliability: 可靠性。指的是系统必须尽可能的可靠，不会意外的崩溃，重启甚至导致系统物理损坏，这意味着一个具有可靠性的系统必须能够对于某些小的错误能够做到自修复，对于无法自修复的错误也尽可能进行隔离，保障系统其余部分正常运转。

Availability：可用性。指的是系统必须能够确保尽可能长时间工作而不下线，即使系统出现一些小的问题也不会影响整个系统的正常运行，在某些情况下甚至可以进行 Hot Plug 的操作，替换有问题的组件，从而严格的确保系统的 downtime 时间在一定范围内。

Serviceability：指的是系统能够提供便利的诊断功能，如系统日志，动态检测等手段方便管理人员进行系统诊断和维护操作，从而及早的发现错误并且修复错误。

RAS 作为一个整体，其作用在于确保整个系统尽可能长期可靠的运行而不下线，并且具备足够强大的容错机制。这对于像大型的数据中心，网络中心如股票证券交易所，电信机房，银行的数据库中心等应用环境是不可或缺的一部分。

3. **Clustering定义（P5）**
4. The main principle behind clustering is that of redundancy（P6）
5. **Load balancing and Failover（P7）**
   a. session stickiness
   b. Request-level failover(P8)
   c. Session failover(P8)
6. **The concept of idempotence(P9)**
7. **nginx Load balancing methods**(P15-19)
   a. Session persistence(P18)
   b. Weighted load balancing(P19)

- 在轮询中，如果服务器down掉了，会自动剔除该服务器。
- 缺省配置就是轮询策略。
- 此策略适合服务器配置相当，无状态且短平快的服务使用。

weight：

- 权重越高分配到需要处理的请求越多。
- 此策略可以与least_conn和ip_hash结合使用。
- 此策略比较适合服务器的硬件配置差别比较大的情况。

ip_hash：

- 在nginx版本1.3.1之前，不能在ip_hash中使用权重（weight）。
- ip_hash不能与backup同时使用。
- 此策略适合有状态服务，比如session。
- 当有服务器需要剔除，必须手动down掉。

least_conn：

- 此负载均衡策略适合请求处理时间长短不一造成服务器过载的情况。

8. **MySQL InnoDB Cluster(P35)(没细看，感觉不会怎么考）**

InnoDB Cluster是MySQL官方推出的高可用方案，一个集群由至少3个MySQL Server组成，它提供了MySQL服务的高可用性和可伸缩性，InnoDB Cluster包含下面三个关键组件：

1、MySQL Shell，它是MySQL的高级管理客户端

2、MySQL Server和MGR，使得一组MySQL实例能够提供高可用性，对于MGR，Innodb Cluster提供了一种更加易于编程的方式来处理MGR

3、MySQL Router，一个轻量级的中间件

补充：

- **反向代理**

  那么，随着请求量的爆发式增长，服务器觉得自己一个人始终是应付不过来，需要兄弟服务器们帮忙，于是它喊来了自己的兄弟以及代理服务器朋友。此时，来自不同客户端的所有请求实际上都发到了代理服务器处，再由代理服务器按照一定的规则将请求分发给各个服务器。

  这就是反向代理（Reverse Proxy），反向代理隐藏了服务器的信息，它代理的是服务器端，代其接收请求。换句话说，反向代理的过程中，客户端并不知道具体是哪台服务器处理了自己的请求。如此一来，既提高了访问速度，又为安全性提供了保证。