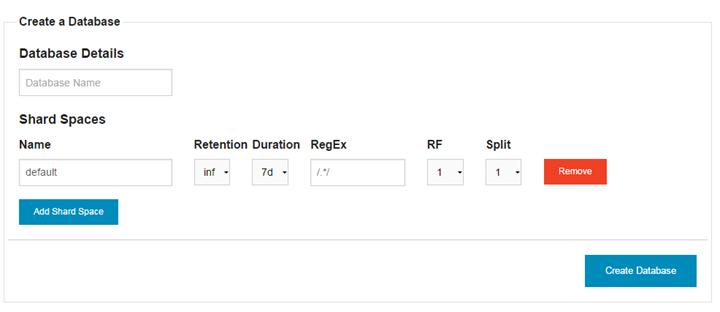
# Influxdb的存储引擎 - 蝈蝈俊 - 博客园

http://www.cnblogs.com/ghj1976/p/4146949.html

创建Influxdb数据库时，我们可以看到下面选项，每个选项的含义就是本文要描述的：

[](http://images.cnitblog.com/blog/120296/201412/051618559835844.png)

Influxdb内部数据的存储可以使用不同的存储引擎。当前0.8.7版本支持的是LevelDB, RocksDB, HyperLevelDB, 和 LMDB。

这几个数据库都是kv类型的数据库，相关信息如下：

LevelDB 是一个google实现的非常高效的kv数据库，目前的版本1.2能够支持billion级别的数据量了。   
LevelDB 是单进程的服务，性能非常之高，在一台4核Q6600的CPU机器上，每秒钟写数据超过40w，而随机读的性能每秒钟超过10w。   
此处随机读是完全命中内存的速度，如果是不命中 速度大大下降   
LevelDB 只是一个 C/C++ 编程语言的库, 不包含网络服务封装, 所以无法像一般意义的存储服务器(如 MySQL)那样, 用客户端来连接它. LevelDB 自己也声明, 使用者应该封装自己的网络服务器.

RocksDB 是一个来自 facebook 的可嵌入式的支持持久化的 key-value 存储系统，也可作为 C/S 模式下的存储数据库，但主要目的还是嵌入式。RocksDB 基于 LevelDB 构建。

HyperLevelDB 是 HyperDex 开发的一个数据存储引擎，改进自 Google 的 LevelDB 以满足 HyperDex 的业务需要。   
HyperLevelDB 主要在 LevelDB 上改进了：   
1. 改进并行机制，使用更细粒度的内部锁控制来提供多 writer 线程的高吞吐量   
2. 改进数据压缩

LMDB 是一个快而小的 key-value 数据存储服务，是由 OpenLDAP 项目的 Symas 开发的。使用内存映射文件，因此读取的性能跟内存数据库一样。其大小受限于虚拟地址空间的大小。

Influxdb 官方试验了这三个引擎，发现RocksDB性能好，所以Influxdb的默认存储引擎是RocksDB。

Influxdb 的数据存储可以支持多碎片存储，每个碎片可以是一种存储引擎，如下图，一个数据库可以有多个碎片。

[](http://images.cnitblog.com/blog/120296/201412/051618585925913.png)

每个碎片存储都有下面属性，跟上面图的内容项对应：

{

"name": "high\_precision",

"database": "pauls\_db",

"retentionPolicy": "7d",

"shardDuration": "1d",

"regex": "/^[a-z].\*/",

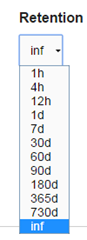
"replicationFactor": 1,

"split": 1

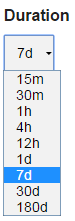
}

在配置参数中， 我们可以看到 "database": "pauls\_db" 标示 每个碎片存储都只能属于一个特定的数据库，一个数据库可以有多个 Shard Space。

"retentionPolicy": "7d" 表示数据被保存的时间（最少保存时间）， 图中的 Retention 就是这个， 下图是系统界面中，对这个时间的设置， inf 标示永久。

[](http://images.cnitblog.com/blog/120296/201412/051618597951212.png)

"shardDuration": "1d",    表示 多长时间做次清理。

[](http://images.cnitblog.com/blog/120296/201412/051619004671841.png)

shardDuration 的值应该小于 retentionPolicy， 大于我们查询时的group by time() 的值。

上面配置的例子中 "retentionPolicy": "7d", "shardDuration": "1d",   会导致我们保存 7-8 天的数据， 每天都会清理，把7天前的数据清理掉一次。

"replicationFactor": 1, 每个存储碎片保存到几台服务器的设置；

"split": 1 给定的时间间隔内，有多少个存储碎片。

注意，这里有下面一个隐含的关系： replicationFactor \* split == 服务器的数量。

数据被分配到那个碎片空间是基于下面的算法：

* Look up the shard spaces for the InfluxDB database
* Loop through the spaces and use the first one that matches the series name
* Lookup the shards for the given time interval
* If no shards exist, create N shards for the interval based on split
* Assign the data to a given shard in the interval using the algorithm  hash(series\_name) % N

使用 shard spaces 的最佳实践是把高精度，大数据的数据 每个时间段写一个 shard spaces 。在使用时把他们再合成一起。

**参考资料：**

Influxdb Storage Engines   
<http://influxdb.com/docs/v0.8/advanced_topics/sharding_and_storage.html>

分类: [Influxdb](http://www.cnblogs.com/ghj1976/category/629197.html)

# influxdb和boltDB简介

——MVCC+B+树，Go写成，Bolt类似于LMDB，这个被认为是在现代kye/value存储中最好的，influxdb后端存储有LevelDB换成了BoltDB - bonelee - 博客园 http://www.cnblogs.com/bonelee/p/6236234.html

**influxdb**

influxdb是最新的一个时间序列数据库，最新一两年才产生，但已经拥有极高的人气。influxdb 是用Go写的，0.9版本的influxdb对于之前会有很大的改变，后端存储有LevelDB换成了BoltDB，读写的API也是有了很大的变化，也将支持集群化，continuous query，支持retention policy,读写性能也是哇哇的，可以说是时间序列存储的完美方案，但是由于还很年轻，可能还会存在诸多的问题，就像现在正在开发的0.9一样，发布一拖再拖，就是由于还有些技术壁垒没有攻陷。

对于influxdb我不想多说些什么，之后打算开一个专题，专门详细来说一说这个玩意，因为我看国内几乎没有详细的文章来讲influxdb的。

如果你想让你的Go应用中的数据持久化，大多数人会使用一些数据库。最简单最方便的选择是嵌入式数据库，有很多嵌入式数据库都是C写的，然而对于Go开发者来说，更希望使用纯粹的Golang的解决方案。

Bolt就是这么一个纯粹的Go语言版的嵌入式key/value的数据库，而且在Go的应用中很方便地去用作持久化。Bolt类似于LMDB，这个被认为是在现代kye/value存储中最好的。但是又不同于LevelDB，BoltDB支持完全可序列化的ACID事务，也不同于SQLlite，BoltDB没有查询语句，对于用户而言，更加易用。

BoltDB将数据保存在一个单独的内存映射的文件里。它没有wal、线程压缩和垃圾回收；它仅仅安全地处理一个文件。

**LevelDB和BoltDB的不同**

LevelDB是Google开发的，也是一个k/v的存储数据库，和BoltDB比起起来有很大的不同。对于使用者而言，最大的不同就是LevelDB没有事务。在其内部，也有很多的不同：LevelDB实现了一个日志结构化的merge tree。它将有序的key/value存储在不同文件的之中，并通过“层级”把它们分开，并且周期性地将小的文件merge为更大的文件。这让其在随机写的时候会很快，但是读的时候却很慢。这也让LevelDB的性能不可预知：但数据量很小的时候，它可能性能很好，但是当随着数据量的增加，性能只会越来越糟糕。而且做merge的线程也会在服务器上出现问题。LevelDB是C++写的，但是也有些Go的实现方式，如[syndtr/goleveldb](https://code.google.com/p/leveldb-go/)、[leveldb-go](https://github.com/syndtr/goleveldb)。

BoltDB使用一个单独的内存映射的文件，实现一个写入时拷贝的B+树，这能让读取更快。而且，BoltDB的载入时间很快，特别是在从crash恢复的时候，因为它不需要去通过读log（其实它压根也没有）去找到上次成功的事务，它仅仅从两个B+树的根节点读取ID。

按照官方说法，boltDB特点：

**Comparison with other databases**

**Postgres, MySQL, & other relational databases**

Relational databases structure data into rows and are only accessible through the use of SQL. This approach provides flexibility in how you store and query your data but also incurs overhead in parsing and planning SQL statements. Bolt accesses all data by a byte slice key. This makes Bolt fast to read and write data by key but provides no built-in support for joining values together.

Most relational databases (with the exception of SQLite) are standalone servers that run separately from your application. This gives your systems flexibility to connect multiple application servers to a single database server but also adds overhead in serializing and transporting data over the network. Bolt runs as a library included in your application so all data access has to go through your application's process. This brings data closer to your application but limits multi-process access to the data.

**LevelDB, RocksDB**

LevelDB and its derivatives (RocksDB, HyperLevelDB) are similar to Bolt in that they are libraries bundled into the application, however, their underlying structure is a log-structured merge-tree (LSM tree). An LSM tree optimizes random writes by using a write ahead log and multi-tiered, sorted files called SSTables. Bolt uses a B+tree internally and only a single file. Both approaches have trade-offs.

**If you require a high random write throughput (>10,000 w/sec) or you need to use spinning disks then LevelDB could be a good choice. If your application is read-heavy or does a lot of range scans then Bolt could be a good choice**.

One other important consideration is that LevelDB does not have transactions. It supports batch writing of key/values pairs and it supports read snapshots but it will not give you the ability to do a compare-and-swap operation safely. Bolt supports fully serializable ACID transactions.

**LMDB**

Bolt was originally a port of LMDB so it is architecturally similar. Both use a B+tree, have ACID semantics with fully serializable transactions, and support lock-free MVCC using a single writer and multiple readers.

The two projects have somewhat diverged. LMDB heavily focuses on raw performance while Bolt has focused on simplicity and ease of use. For example, LMDB allows several unsafe actions such as direct writes for the sake of performance. Bolt opts to disallow actions which can leave the database in a corrupted state. The only exception to this in Bolt is DB.NoSync.

There are also a few differences in API. LMDB requires a maximum mmap size when opening an mdb\_env whereas Bolt will handle incremental mmap resizing automatically. LMDB overloads the getter and setter functions with multiple flags whereas Bolt splits these specialized cases into their own functions.

参考：

http://www.opscoder.info/boltdb\_intro.html

https://github.com/boltdb/bolt

# InfluxDB引擎浅析 - 微店技术团队 - SegmentFault

https://segmentfault.com/a/1190000005977485

**引言**

InfluxDB是一款Go语言写的时序数据库。时序数据库主要用于存储基于时间序列的指标数据，例如一个Web页面的PV、UV等指标，将其定期采集，并打上时间戳，就是一份基于时间序列的指标。时序数据库通常用来配合前端页面来展示一段时间的指标曲线。

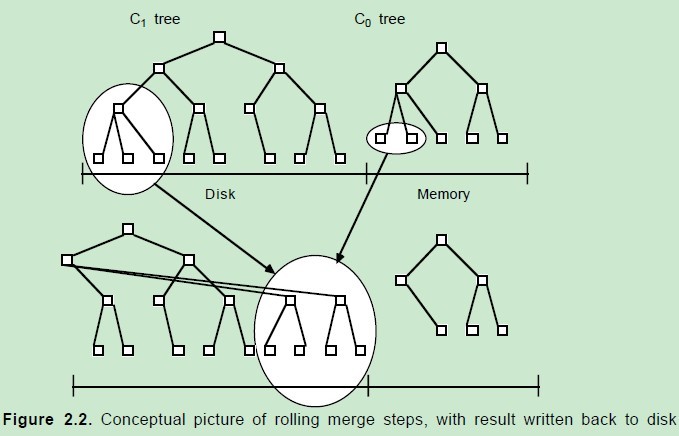
**为什么需要时序数据库**

时序数据库较传统的关系型数据库以及NoSQL究竟有什么优势，下面会结合相关模型的特性进行分析

**LSM Tree**

LSM tree是基于Google的BigTable架构，数据以K-V方式存储。

写数据首先会插入到内存中的树。当内存中的树中的数据超过一定阈值时，会进行合并操作。合并操作会从左至右遍历内存中的树的叶子节点与磁盘中的树的叶子节点进行合并，当被合并的数据量达到磁盘的存储页的大小时，会将合并后的数据持久化到磁盘，同时更新父亲节点对叶子节点的指针。



这种机制保证了写入的效率，因为数据会在合并后顺序写入磁盘页。但会推迟磁盘回写，因此为保障读数据的一致性，会先在内存中查询，如果内存中没有，则到磁盘上查询。

删除数据时，在内存（C0）中查找，如果没有，则在内存中新建一个索引，将键值设置删除标记（创建墓碑），这样后续的滚动合并操作时，再有查询操作，就会被直接返回该键值不存在。 数据会在之后的Compaction当中从数据文件中删除。

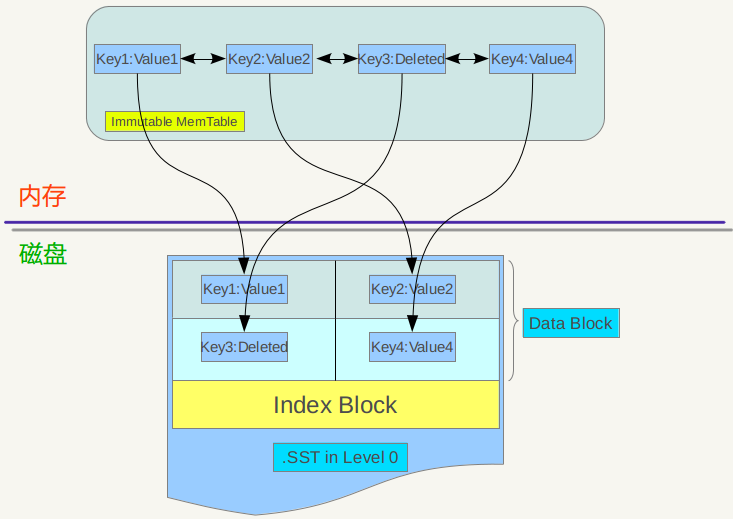
**Compaction**

当日志文件超过一定大小的阈值是 (默认为 1MB):

建立一个新的memtable和日志文件，以后的操作都是用新的memtable和日志文件

后台进行如下操作:

1. 将旧的 memtable写到SSTable中（过程为先转为immtable\_table，然后遍历写入)
2. 废弃旧的 memtable
3. 删除旧的 memtable和日志文件
4. 将新的SSTable加到level 0中.



对于时序数据而言，LSM tree的读写效率很高。但是热备份以及数据批量清理的效率不高。

**B+ Tree**

B+ Tree，很多关系型数据库像 Berkerly DB , sqlite , mysql 数据库都使用了B+树算法处理索引。B+ Tree的特点是数据按照索引有序排放，牺牲一定写入性能，保证了读取效率。但数据量很大时（GB），查询效率就会很低。因为数据量越大，树分叉就越多，遍历时的开销就越大。

**TSM**

influxdb在v0.9.5版本引入TSM引擎，该引擎修改自LSM

**预写日志**

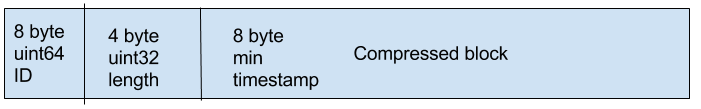
当前日志文件达到2MB大小后封闭，并开始写新的日志文件

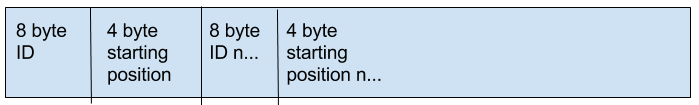
写数据时，日志文件落盘(fsync)且数据索引加入内存表后返回成功。这样的设计保证了数据的一致性。同时对写盘的吞吐性能提出要求，建议批量提交数据（influxdb提供了批量提交的API）。日志遵循TLV格式，并采用较精简的数据结构，来减少写操作的开销。

**数据文件**

文件结构  
  
一个文件的中数据块按照时序进行排列

对照LevelDB的结构，增加了min和max time, 基于一段时间范围的数据提取会非常简单

Data Block结构  
  
ID由存放的key (measurement name + tagset) 以及 field name进行hash(fnv64-a hash)生成  
Compressd block当中会存储metric值，数据压缩算法后面会进行详述

Index Block结构  


**读取数据**

首先会根据查询请求的时间范围，在数据文件中进行二进制搜索，找到符合范围的文件。之后在内存中的映射表根据查询指标项HASH获取ID，并通过索引找到数据块的起始地址。之后根据数据块及其下一数据块的timestamp我们可以推算出需要取出多少个数据块，最后将数据块中的数据解压，得到结果

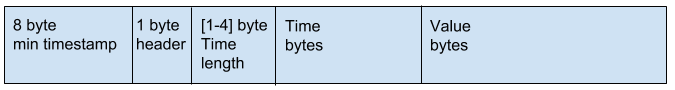
**更新数据**

如果多个更新在同一个时间范围内，预写日志会缓存起来一起更新。

**删除数据**

两阶段式处理，第一阶段，预写日志会将其持久化在日志中，并通知索引维护内存中的墓碑. 此时查询数据，就会返回不存在。第二阶段，预写日志写索引文件，会优先处理删除，之后再处理删除操作之后的其他插入(包括删除的序列以及其他序列)，并清除内存中的墓碑。

**数据压缩**

数据压缩的目的是为了减少存储空间以及降低写磁盘的开销  
  
每个压缩数据块当中会包含一个系列的点（压缩时间戳、压缩值), 因为时间戳是一个单调递增的序列，因此压缩时填入的时间的偏移量

**总结**

influxdb的数据存储结构实现了数据基于系列以及时间戳2个维度的有序存取。并通过压缩数据来降低I/O开销。在取一个系列在一定时间范围内的数据这个场景下，能够提高处理速度。 由于数据按时间进行归并，对Retention操作而言，可以以数据文件为单位进行操作，效率会比较高。

**硬件需求**

| **负载** | **写/秒** | **读/秒** | **唯一系列** |
| --- | --- | --- | --- |
| 低 | < 5千 | < 5 | < 10万 |
| 标准 | < 10万 | < 25 | < 100万 |
| 高 | > 10万 | > 25 | > 100万 |
| 极限 | 50万 | 100 | 1000万 |

低配

* CPU: 2-4核
* RAM: 2-4 GB
* IOPS: 500

标配

* CPU: 4-6核
* RAM: 8-32 GB
* IOPS: 500-1000

高

* CPU: 8核+
* RAM: 32+ GB
* IOPS: 1000+

内存大小与唯一系列（指标项）数量相关。

文章参考

1. [InfluxDB concept文档](https://docs.influxdata.com/influxdb/v0.13/concepts/storage_engine/)
2. [LevelDB 原理](http://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html)