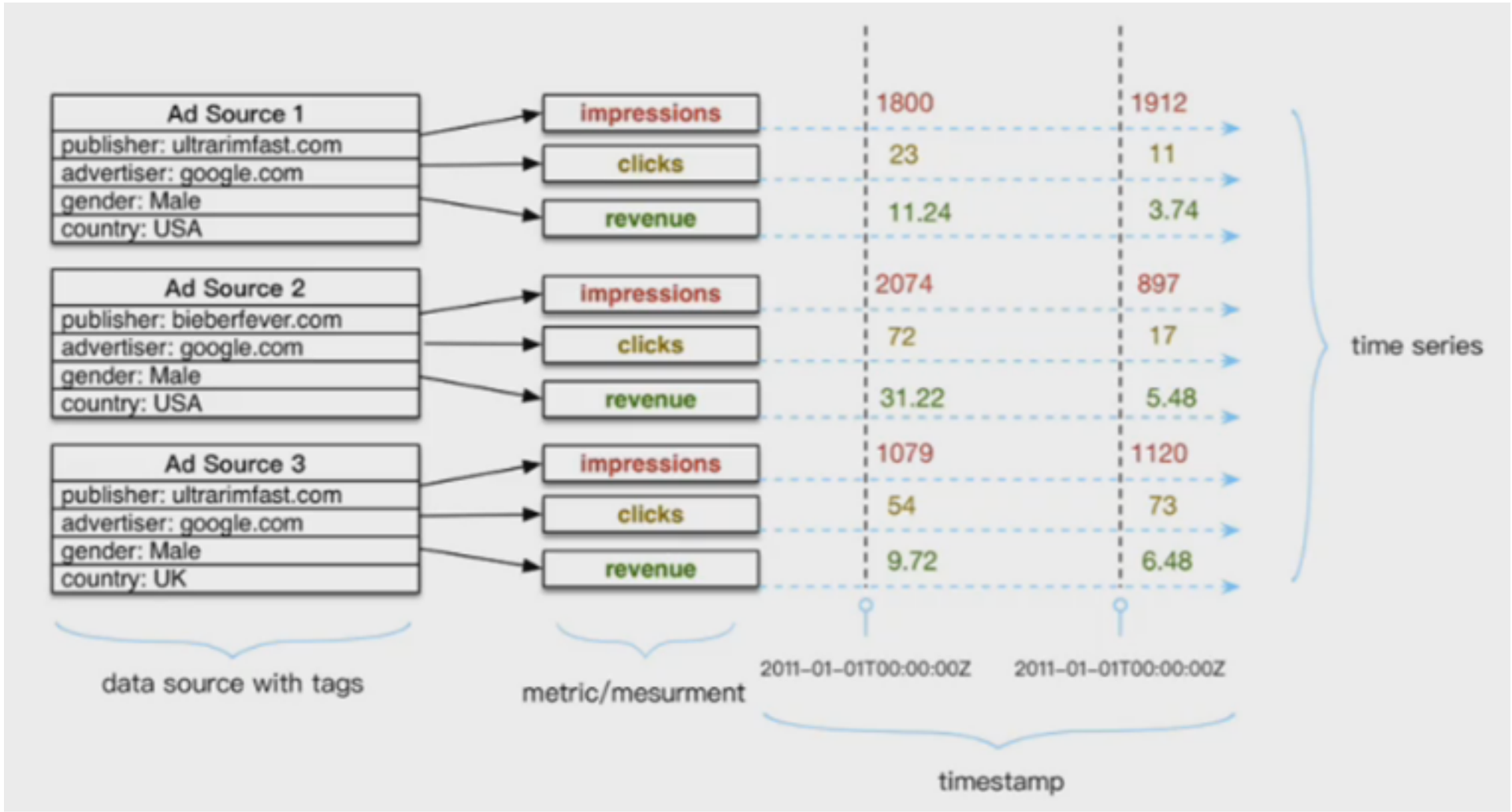
Beacon 时序数据流存储模型

设计初稿 中文版 V0.2

2019/09/27

背景知识（1）：典型的时序数据示意图



- 时序数据由两个维度坐标来表示，横坐标表示时间轴，随着时间的不断流逝，数据也会源源不断地吐出来；纵坐标由两种元素构成，分别是数据源和metric，数据源由一系列的标签（tag，也称为维度）唯一表示，图中数据源是一个广告数据源，这个数据源由publisher、advertiser、gender以及country四个维度值唯一表示，metric表示待收集的数据源指标。一个数据源通常会采集很多指标（metric），上图中广告数据源就采集了impressions、clicks以及revenue这三种指标，分别表示广告浏览量、广告点击率以及广告收入。
- 简单概括为：一个时序数据点（point）由datasource(tags)+metric+timestamp这三部分唯一确定。

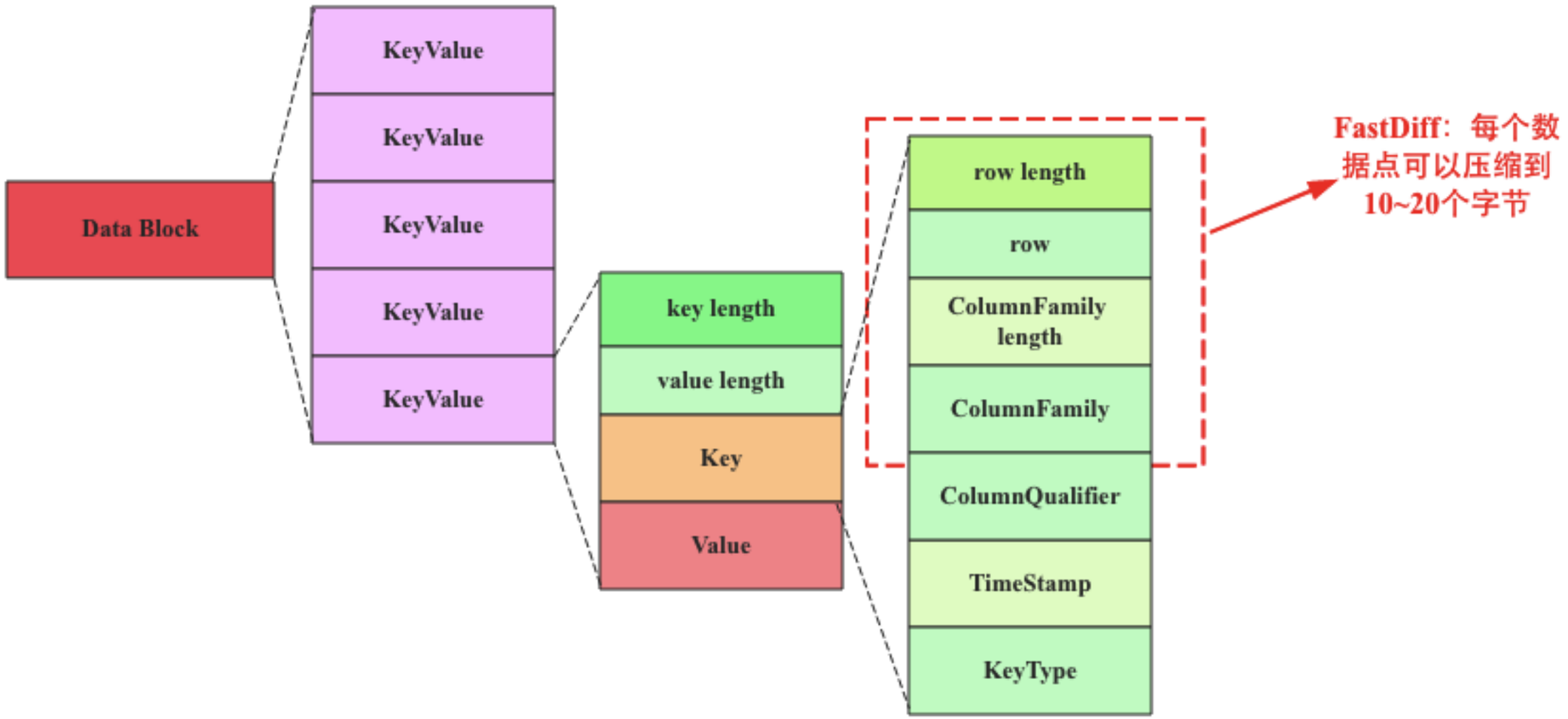
背景知识（2）：时序业务特点

- 持续产生海量数据，没有波峰波谷。
- 数据都是插入操作，基本没有更新删除操作。
- 近期数据关注度更高，未来会更关注流式处理这个环节，时间久远的数据极少被访问，甚至可以丢弃。
- 数据存在多个维度的标签，往往需要多维度联合查询以及统计查询。

关注的核心技术点：

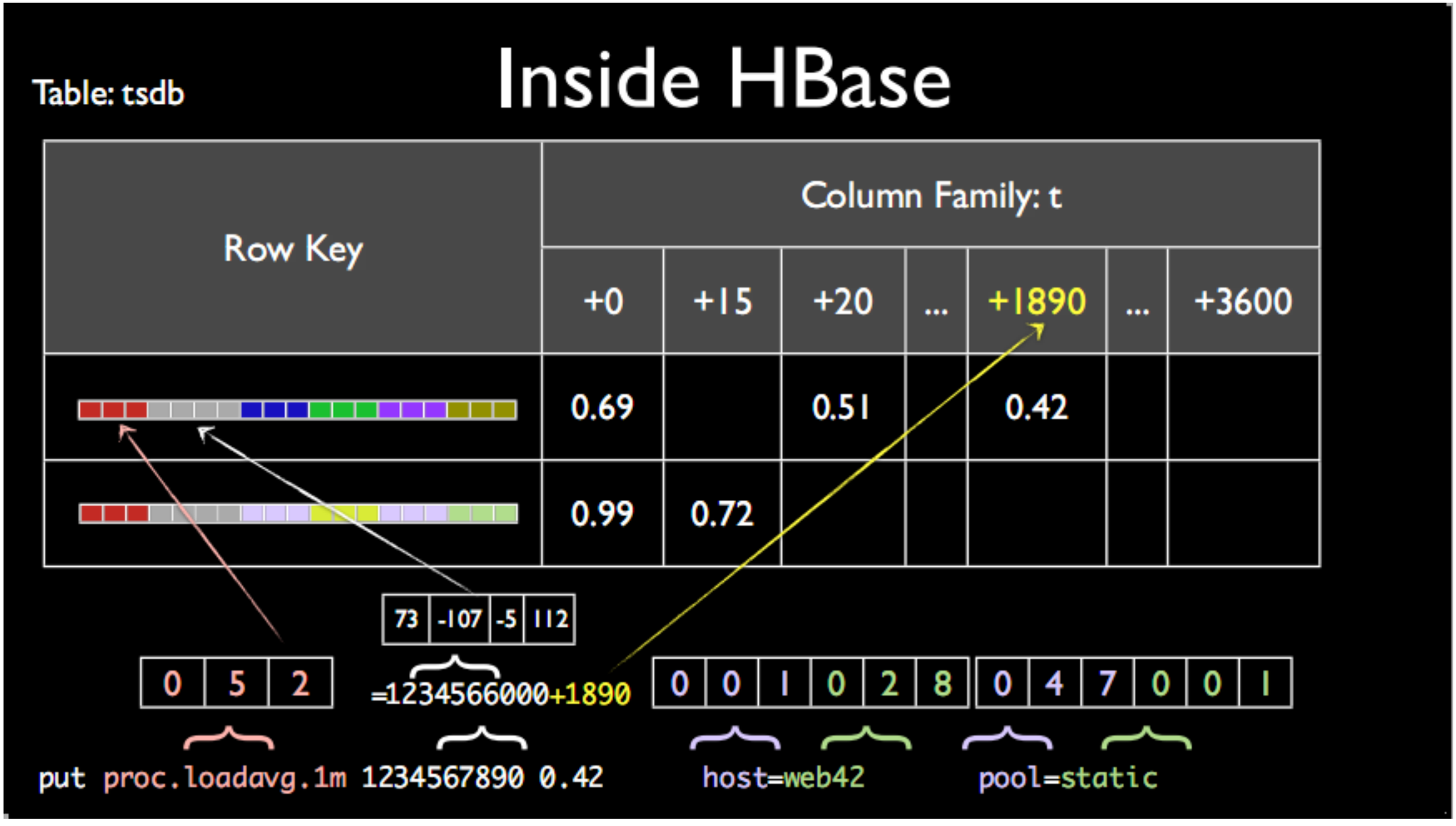
- **高吞吐量写入能力：**系统具有水平扩展性和单机LSM体系结构。LSM体系结构是用来保证单台机器的高吞吐量写入，LSM结构下数据写入只需要写入内存以及追加写入日志，这样就不再需要随机将数据写入磁盘。
- **数据分级存储/TTL：**时序数据冷热性质决定的技术特性。能够将最近小时级别数据放入内存，最近天级别数据放到SSD，更久远数据放到廉价HDD或直接使用TTL过期淘汰掉。
- **高压缩率**
- **多维度查询能力**
- **高效聚合能力**
- **未来技术点：**异常实时检测、未来预测等；

背景知识（3）：OpenTSDB(HBase)时序数据存储模型



- OpenTSDB基于HBase存储时序数据，RowKey规则：metric + timestamp + datasource(tags)
- 上图是HBase中一个存储KeyValue(KV)数据的数据块结构，一个数据块由多个KeyValue数据组成，在我们的事例中KeyValue就是一个时序数据点（point）。其中Value结构很简单，就是一个数值。而Key就比较复杂了，由rowkey+columnfamily+column+timestamp+keytype组成，其中rowkey等于metric+timestamp+datasource。
- 问题：(1)存在很多无用的字段；(2)数据源和采集指标冗余；(3)无法有效的压缩；(4)不能完全保证多维查询能力；

背景知识（4）：OpenTSDB(HBase)时序数据存储模型



- 优化一：timestamp细粒度并不是到秒级或毫秒级，而是精确到小时级别，然后将小时中每一秒设置到列上。
- 优化二：所有metrics以及所有标签信息（tags）都使用全局编码将标签值编码成更短bit，减少rowkey存储数据量。

  问题：存在很多无用的字段；数据源和采集指标冗余；无法有效的压缩；不能完全保证多维查询能力；

背景知识（5）：Druid时序数据存储模型

datasource(tags)　　metrics

Timestamp	publisher	advertiser	gender	country	impressions	clicks	revenue
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	USA	1800	23	11.24
2011-01-01T00:00:00	biberfever.com	google.com	Male	USA	2074	72	31.22
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	UK	1079	54	9.72

Druid是一个列式数据库，所以每一列都会独立存储，比如Timestamp列会存储在一起形成一个文件，publish列会存储在一起形成一个文件，以此类推。针对冗余这个问题，Druid和HBase的处理方式一样，都是采用编码字典对标签值进行编码，将string类型的标签值编码成int值。但和HBase不一样的是，Druid编码是局部编码，Druid和HBase都采用LSM结构，数据先写入内存再flush到数据文件，Druid编码是文件级别的，局部编码可以有效减小对内存的巨大压力。

- 列式存储模式好处：
  1. 数据存储压缩率高。每列独立存储，可以针对每列进行压缩，而且可以为每列设置对应的压缩策略，比如时间列、int、fload、double、string都可以分别进行压缩，压缩效果更好。
  2. 支持多维查找。Druid为datasource的每个列分别设置了Bitmap索引，利用Bitmap索引可以有效实现多维查找，比如用户想查找20110101T00:00:00这个时间点所有发布在USA的所有广告的浏览量，可以根据country=USA在Bitmap索引中找到要找的行号，再根据行号定位待查的metrics。
- 存储模型的问题：(1)数据依然存在冗余；(2)指定数据源的范围查找并没有OpenTSDB高效；

背景知识（6）：InfluxDB时序数据存储模型

datasource(tags)　　fields

Timestamp	publisher	advertiser	gender	country	impressions	clicks	revenue
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	USA	1800	23	11.24
2011-01-01T00:00:00	biberfever.com	google.com	Male	USA	2074	72	31.22
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	UK	1079	54	9.72

measurement：advertising platform

为了保证写入的高效，InfluxDB也采用LSM结构，数据先写入内存，当内存容量达到一定阈值之后flush到文件。InfluxDB在时序数据模型设计方面提出了一个非常重要的概念：seriesKey，seriesKey实际上就是measurement+datasource(tags)。需要特别注意的是，measurement和上文中提到的measurement并不是一回事，上文中measurement和metric意义相同，表示采集指标，而InfluxDB中measurement更像是表的概念，InfluxDB中使用fields表示指标，如上图所示。

- **时序数据写入内存之后按照seriesKey进行组织**

内存中实际上就是一个Map：<SeriesKey+fieldKey, List<Timestamp|Value>>，Map中一个SeriesKey+fieldKey对应一个List，List中存储时间线数据。数据进来之后根据measurement+datasource(tags)拼成SeriesKey，加上fieldKey，再将Timestamp|Value组合值写入时间线数据List中。

背景知识（7）：InfluxDB时序数据存储模型

内存中的数据flush至文件后，同样会将同一个SeriesKey中的时间线数据写入同一个Block块内，即一个Block块内的数据都属于同一个数据源下的同一个field。

这样设计的好处：

- 同一数据源的tags不再冗余存储。一个Block内的数据都共用一个SeriesKey，只需要将这个SeriesKey写入这个Block的Trailer部分就可以。大大降低了时序数据的存储量。
- 时间序列和value可以在同一个Block内分开独立存储，独立存储就可以对时间列以及数值列分别进行压缩。InfluxDB对时间列的存储借鉴了Beringei的压缩方式，使用delta-delta压缩方式极大的提高了压缩效率。而对Value的压缩可以针对不同的数据类型采用相同的压缩效率。
- 对于给定数据源以及时间范围的数据查找，可以非常高效的进行查找。这一点和OpenTSDB一样。

将datasource(tags)和measurement拼成SeriesKey，也不能实现多维查找。不过InfluxDB内部实现了倒排索引机制，即实现了tag到SeriesKey的映射关系，如果用户想根据某个tag查找的话，首先根据tag在倒排索引中找到对应的SeriesKey，再根据SeriesKey定位具体的时间线数据。InfluxDB的这种存储引擎称为TSM，全称为Timestamp-Structure Merge Tree，基本原理类似于LSM。

背景知识（8）：Beringei时序数据存储模型

Beringei是Facebook开源的一个时序数据库系统。InfluxDB时序数据模型设计很好地将时间序列按照数据源以及metric挑选了出来，解决了维度列值冗余存储，时间列不能有效压缩的问题。但InfluxDB没有很好的解决写入缓存压缩的问题：InfluxDB在写入内存的时候并没有压缩，而是在数据写入文件的时候进行对应压缩。时序数据最大的特点之一是最近写入的数据最热，将最近写入的数据全部放在内存可以极大提升读取效率。Beringei很好的解决了这个问题，流式压缩意味着数据写入内存之后就进行压缩，这样会使得内存中可以缓存更多的时序数据，这样对于最近数据的查询会有很大的帮助。

Beringei的时序数据模型设计与InfluxDB基本一致，也是提出类似于SeriesKey的概念，将时间线挑了出来。但和InfluxDB有两个比较大的区别：

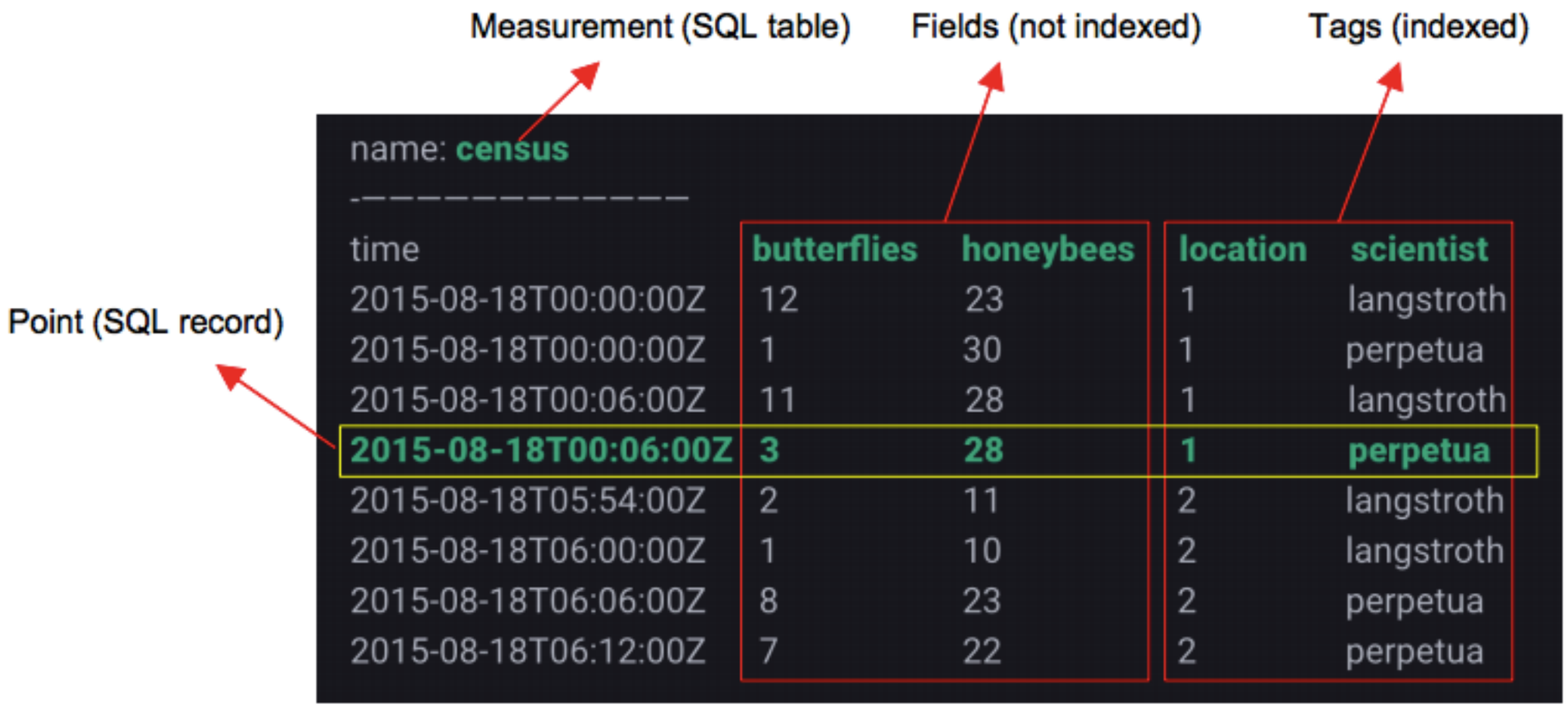
- 文件组织形式不同。Beringei的文件存储形式按照时间窗口组织，比如最近5分钟的数据全部写入同一个文件，这个文件分为很多block，每个block中的所有时序数据共用一个SeriesKey。Beringei文件没有索引，InfluxDB有索引。
- Beringei目前没有倒排索引机制，因此对于多维查询并不高效。

技术体系参考解读（1）：以InfluxDB为主

Rank Apr 2019	Rank Mar 2019	Rank Apr 2018	DBMS	Database Model	Score Apr 2019	Score Mar 2019	Score Apr 2018
1.	1.	1.	InfluxDB	Time Series	17.22	+1.04	+6.46
2.	2.	2.	Kdb+	Time Series, Multi-model	5.85	+0.25	+2.77
3.	3.	↑ 4.	Graphite	Time Series	3.12	+0.05	+0.93
4.	↑ 5.	↑ 7.	Prometheus	Time Series	2.91	+0.20	+1.86
5.	↓ 4.	↓ 3.	RRDtool	Time Series	2.70	-0.05	-0.05
6.	6.	↓ 5.	OpenTSDB	Time Series	2.37	+0.09	+0.67
7.	7.	↓ 6.	Druid	Multi-model	1.65	+0.07	+0.59
8.	8.	↑ 19.	TimescaleDB	Time Series, Multi-model	0.95	+0.04	+0.92
9.	9.	↓ 8.	KairosDB	Time Series	0.64	-0.02	+0.20
10.	↑ 11.	↓ 9.	eXtremeDB	Multi-model	0.40	+0.00	+0.08

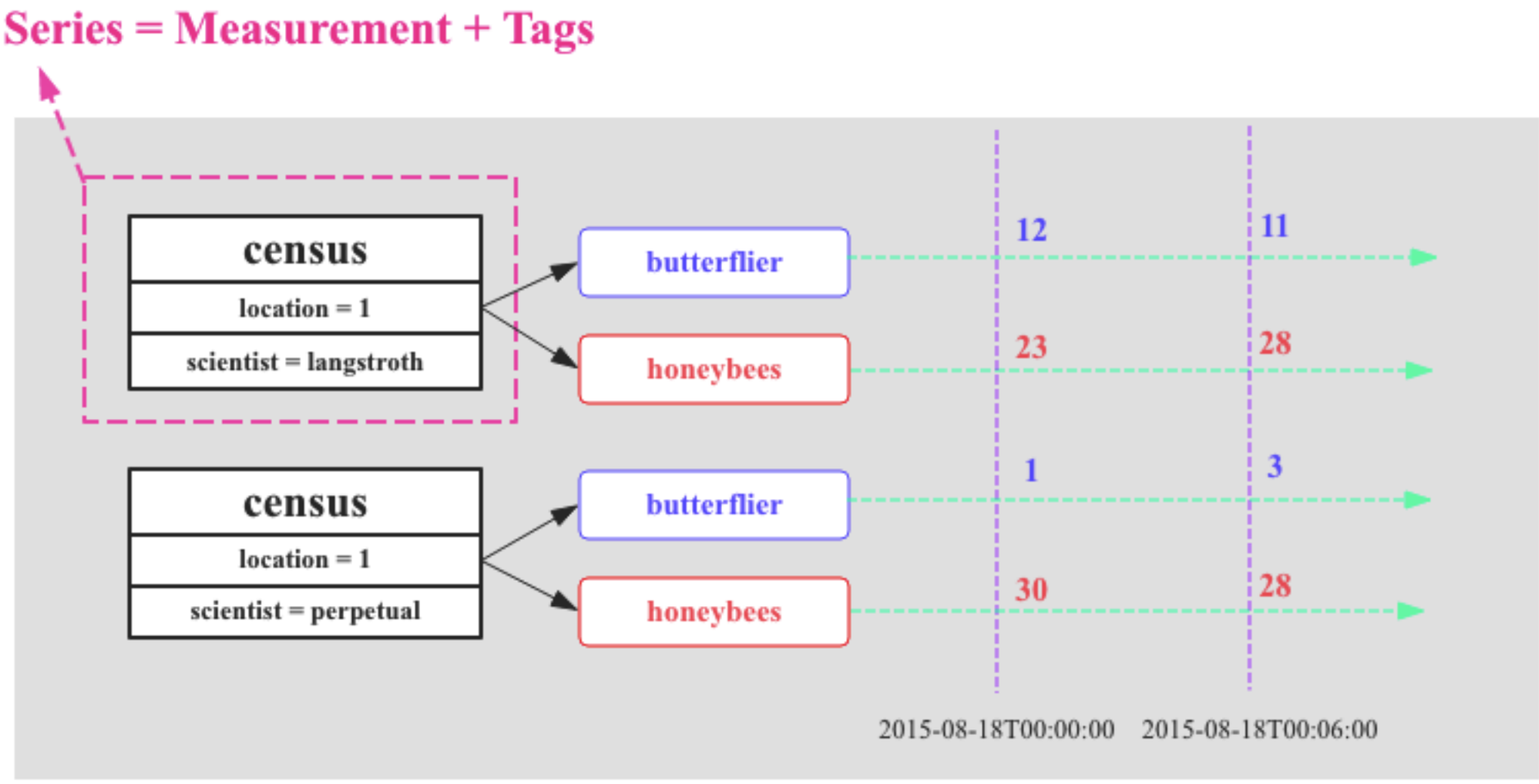
- 经过数年的发展，InfluxDB一枝独秀，在DB-Engines中，遥遥领先其他的时序数据库，成为最受用户欢迎的数据库之一。

技术体系参考解读（2）：InfluxDB表结构示意



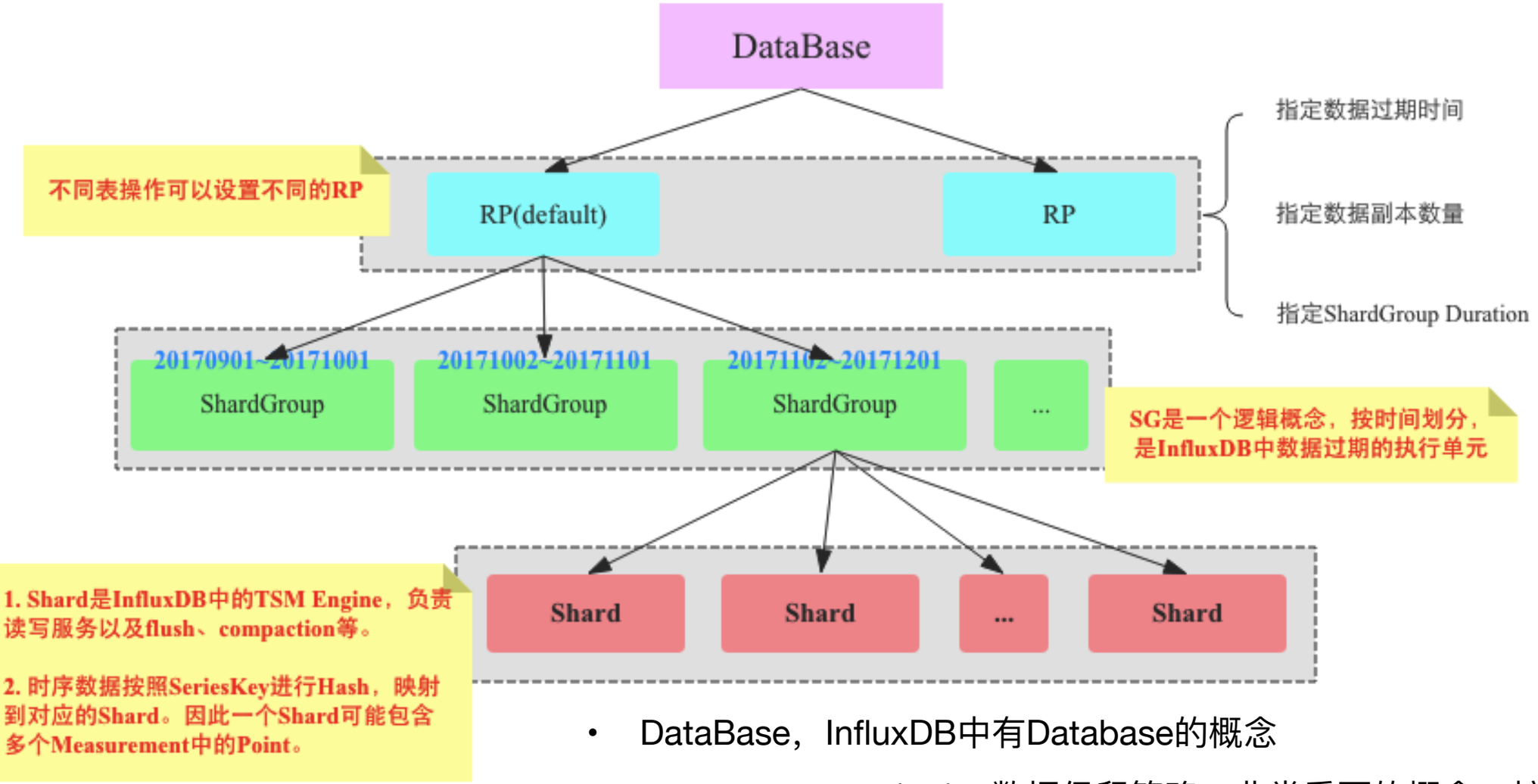
- Measurement：从原理上讲更像SQL中表的概念。
- Tags：维度列。上图中location和scientist分别是表中的两个Tag Key，其中location对应的维度值Tag Values为｛1, 2｝，scientist对应维度值Tag Values为{langstroth, perpetual}，两者组合TagSet有四种。表中Tags组合会被作为记录的主键，因此主键并不唯一，如上表中第一行和第三行记录主键都为’location=1,scientist=langstroth’。所有时序查询最终都会基于主键查询之后再经过时间戳过滤完成。
- Fields：数值列。数值列存放用户的时序数据。
- Point：类似SQL中一行记录，而并不是一个点。

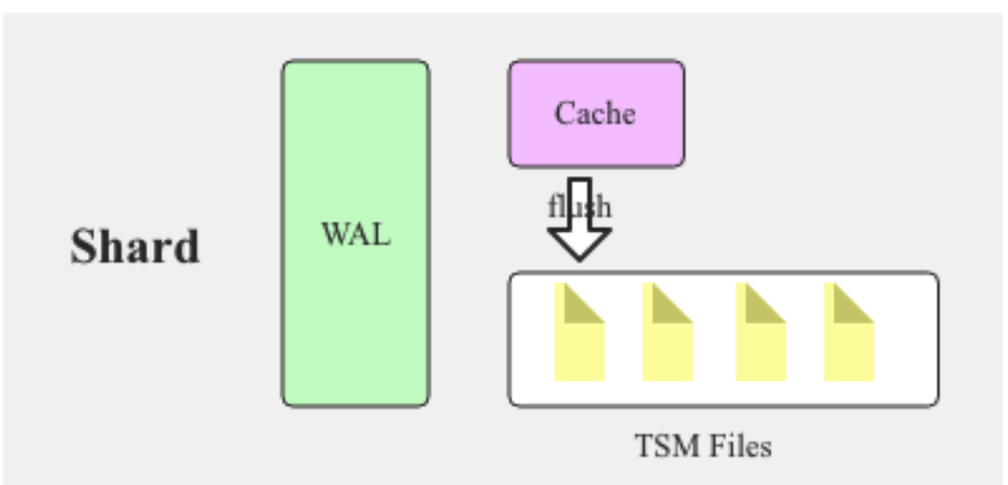
技术体系参考解读（3）：InfluxDB 核心概念Series



- 时序数据的时间线就是一个数据源采集的一个指标随着时间的流逝而源源不断地吐出数据，这样形成的一条数据线称之为时间线。
- 上图中有两个数据源，每个数据源会采集两种指标：butterflier和honeybees。

技术体系参考解读（4）：InfluxDB系统架构



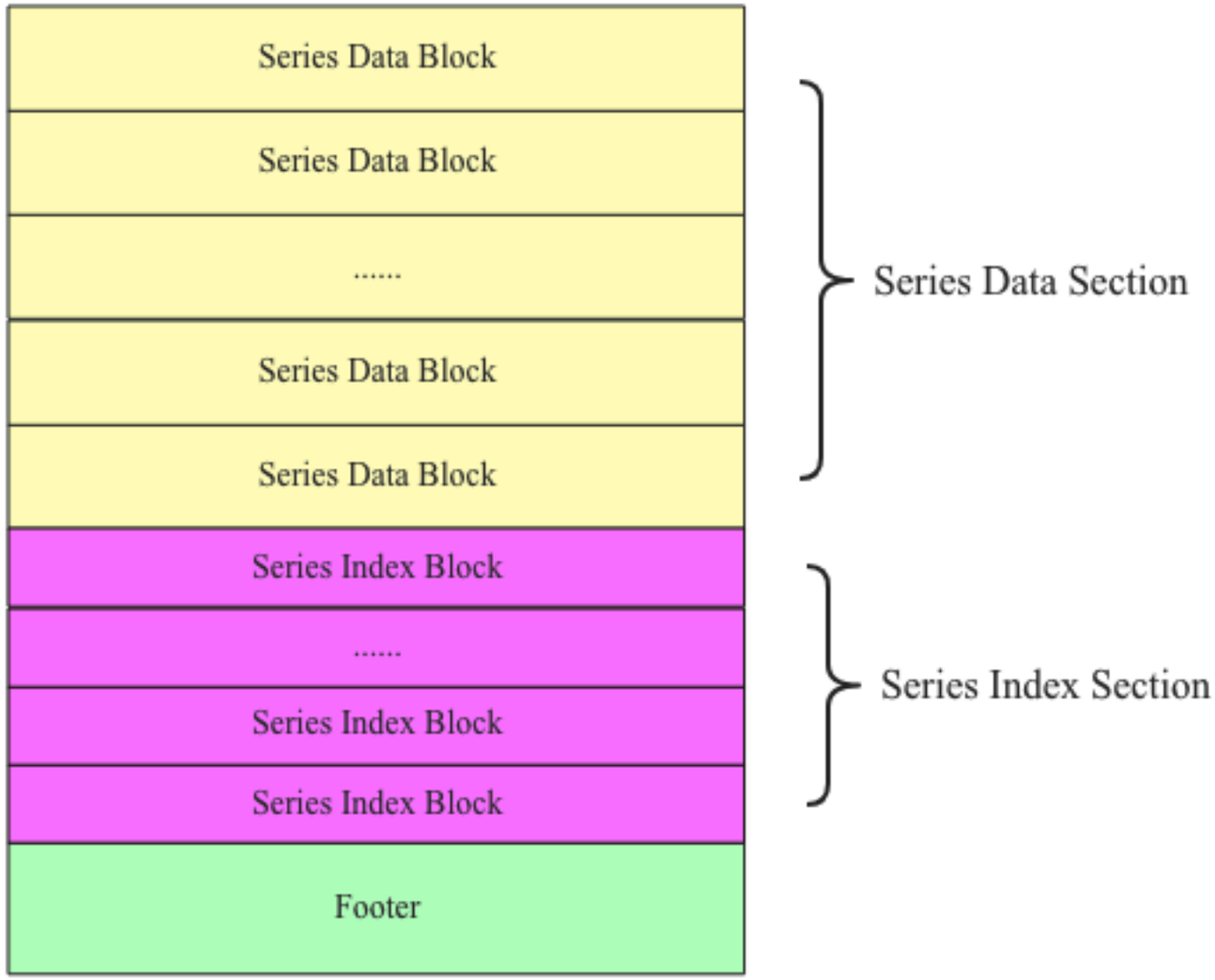


- DataBase，InfluxDB中有Database的概念
- Retention Policy(RP)，数据保留策略，非常重要的概念，核心作用有3个：指定数据的过期时间，指定数据副本数量以及指定ShardGroup Duration
- Shard Group是InfluxDB中一个重要的逻辑概念，每个Shard Group只存储指定时间段的数据，不同Shard Group对应的时间段不会重合。
- Shard是InfluxDB的存储引擎实现，负责数据的编码存储、读写服务等。TSM类似于LSM，包含Cache、WAL以及Data File等各个组件，也会有flush、compaction等这类数据操作。

技术体系参考解读（5）：InfluxDB TSM引擎

基于Map数据结构，时序数据写入内存流程可以表示为如下三步：

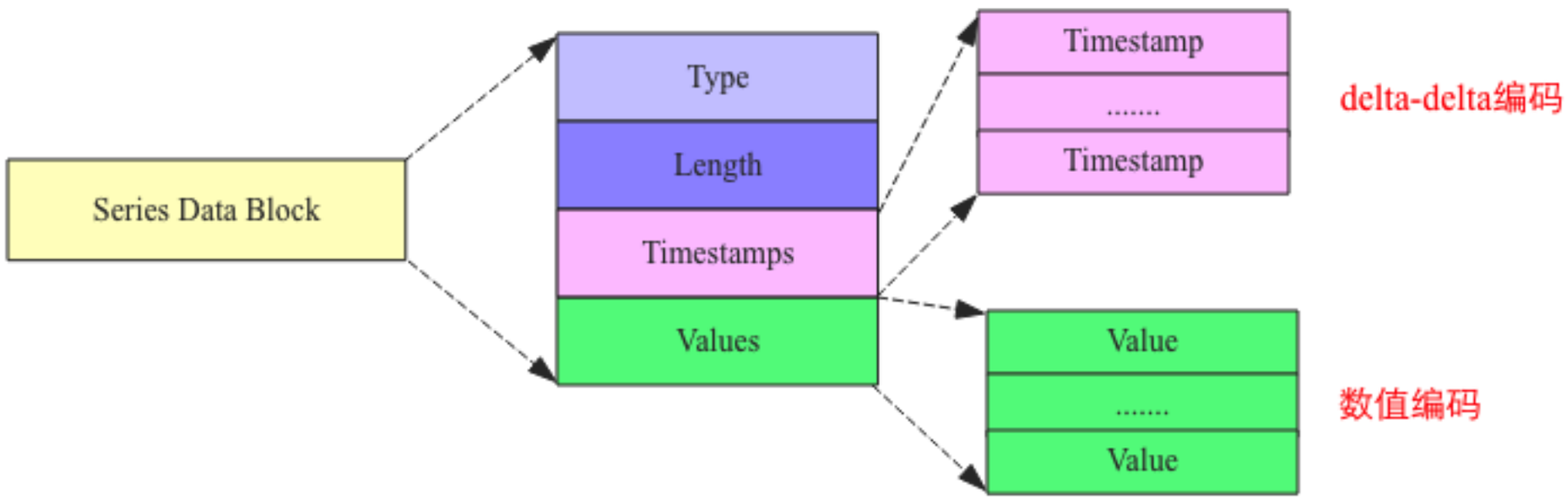
1. 时间序列数据进入系统之后首先根据measurement + datasource(tags)拼成seriesKey
2. 根据这个seriesKey以及待查fieldKey拼成Key，再在Map中根据Key找到对应的时间序列集合，如果没有的话就新建一个新的List
3. 找到之后将Timestamp|Value组合值追加写入时间线数据链表中



TSM文件

- TSM文件最核心由Series Data Section及Series Index Section两个部分组成，其中前者表示存储时序数据的Block，而后者存储文件级别B+树索引Block，用于在文件中快速查询时间序列数据块。

技术体系参考解读（6）：TSM引擎 Series Data Block



Series Data Block由四部分构成：Type、Length、Timestamps以及Values，分别表示意义如下：

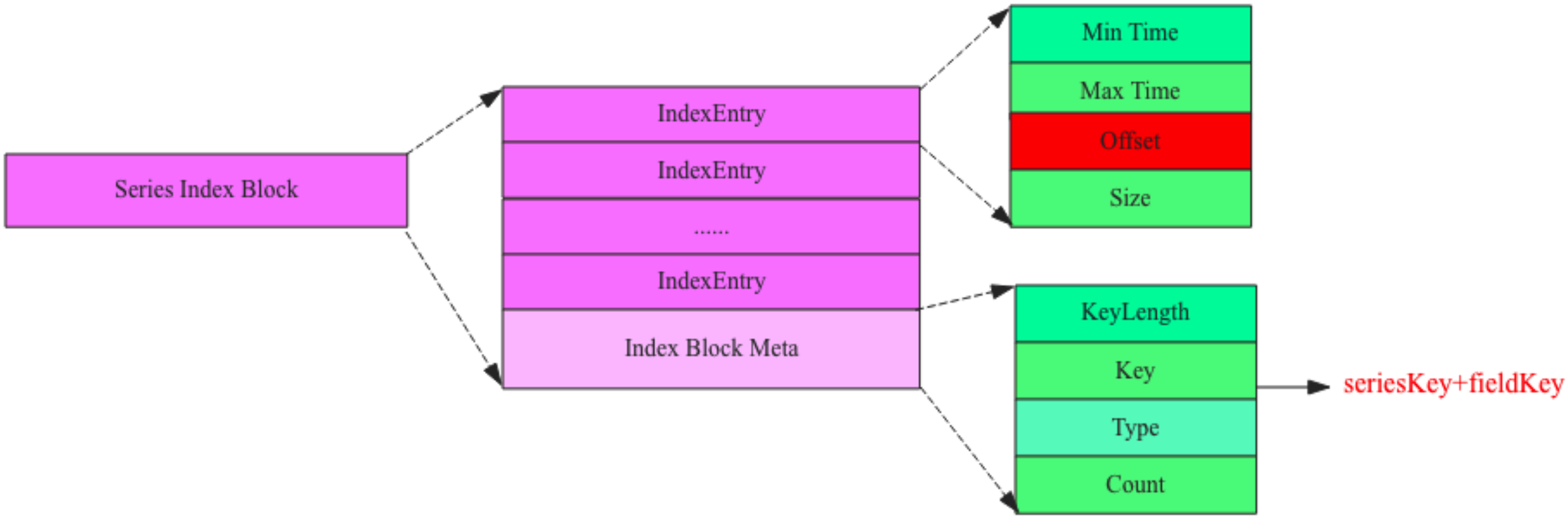
1. Type：表示该seriesKey对应的时间序列的数据类型，数值数据类型通常为int、long、float以及double等。不同的数据类型对应不同的编码方式。

2. Length：len(Timestamps)，用于读取Timestamps区域数据，解析Block。时序数据的时间值以及指标值在一个Block内部是按照列式存储的：所有的时间值存储在一起，所有的指标值存储在一起。使用列式存储可以极大提高系统的压缩效率。

3. Timestamps：时间值存储在一起形成的数据集，通常来说，时间序列中时间值的间隔都是比较固定的，比如每隔一秒钟采集一次的时间值间隔都是1s，这种具有固定间隔值的时间序列压缩非常高效，TSM采用了Facebook开源的Geringei系统中对时序时间的压缩算法：delta-delta编码。

4. Values：指标值存储在一起形成的数据集，同一种Key对应的指标值数据类型都是相同的，由Type字段表征，相同类型的数据值可以很好的压缩，而且时序数据的特点决定了这些相邻时间序列的数据值基本都相差不大，因此也可以非常高效的压缩。需要注意的是，不同数据类型对应不同的编码算法。

技术体系参考解读（7）：TSM引擎 Series Index Block

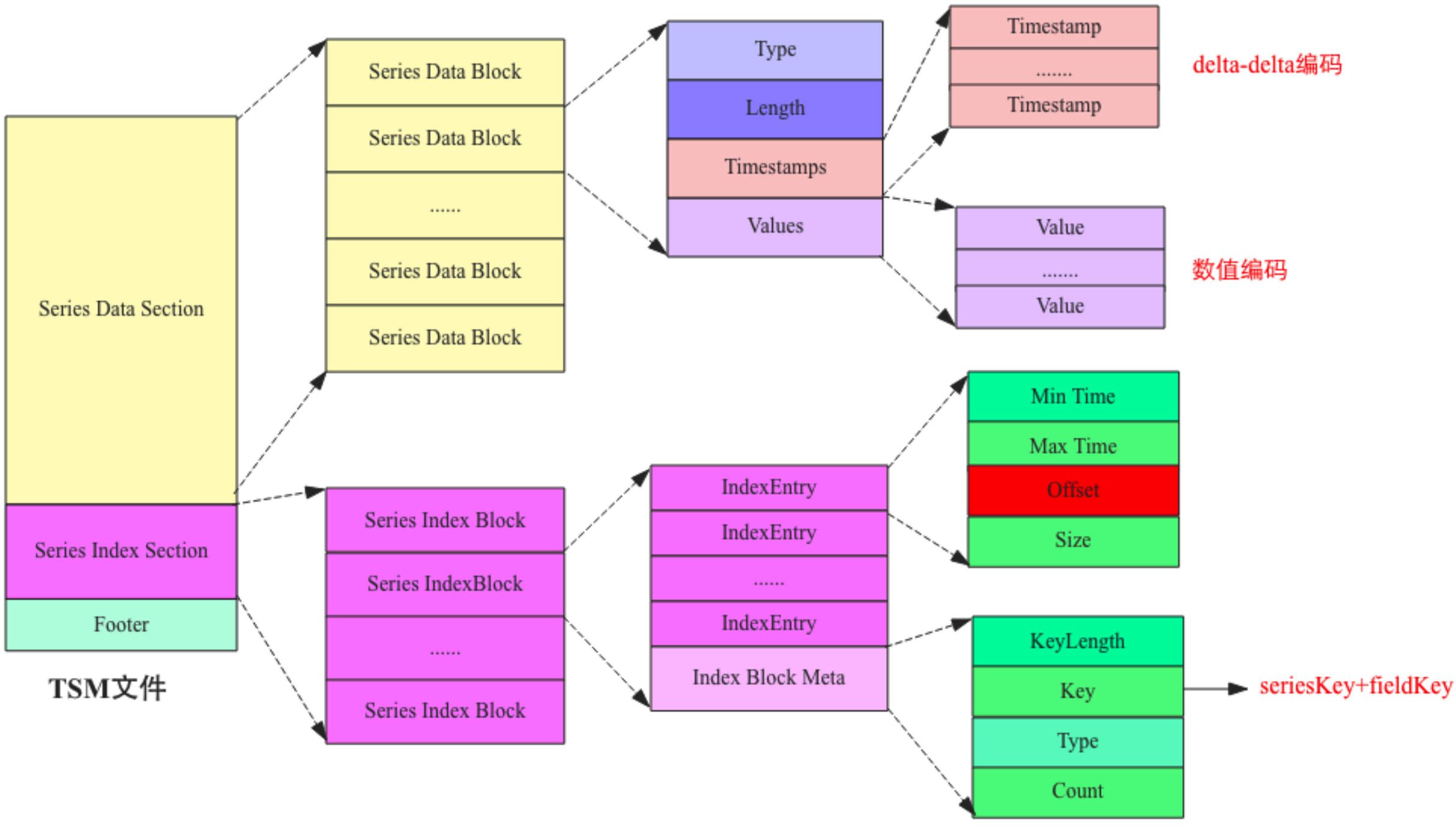


Series Index Block由Index Block Meta以及一系列Index Entry构成：

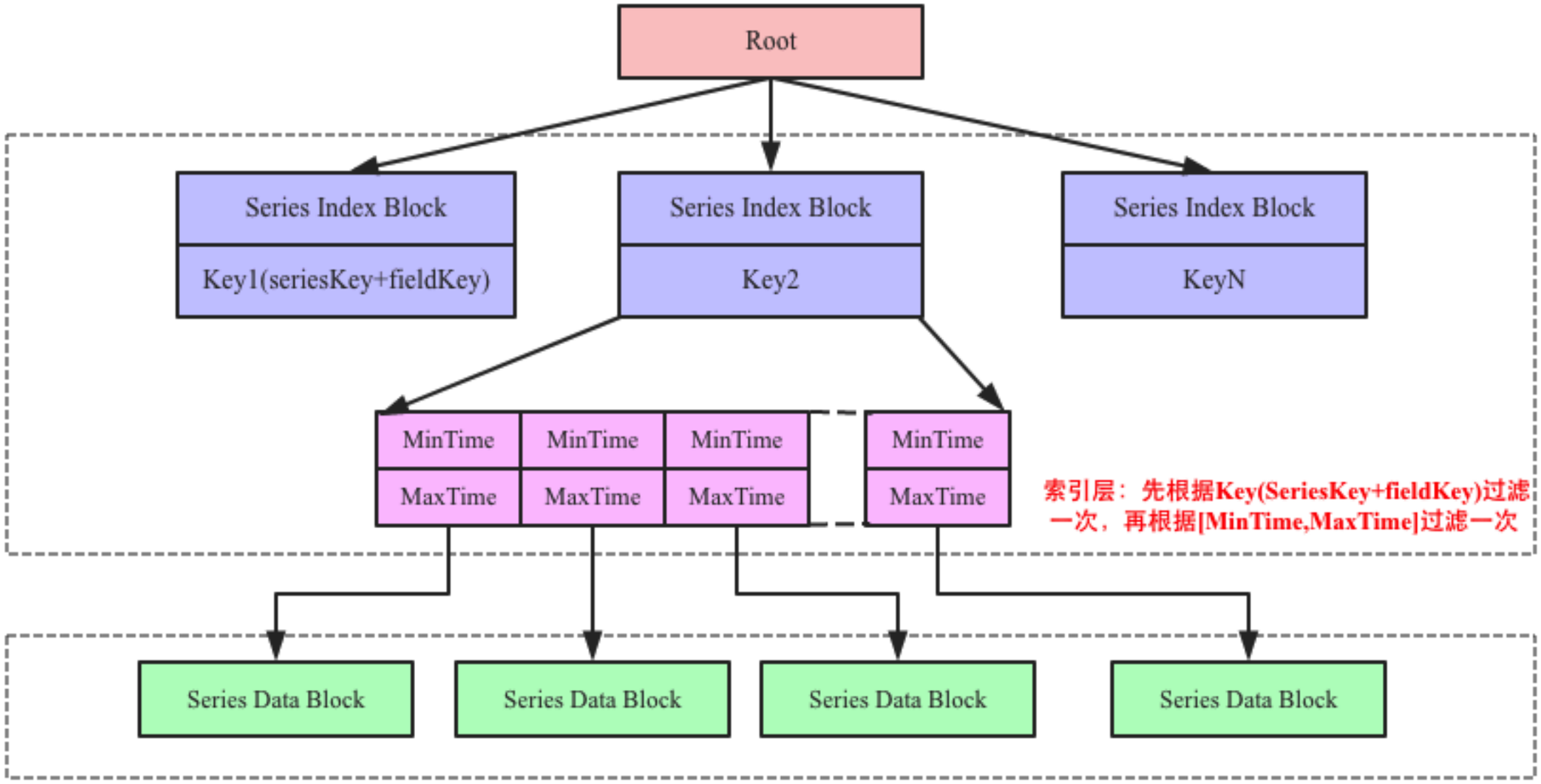
1. Index Block Meta最核心的字段是Key，表示这个索引Block内所有IndexEntry所索引的时序数据块都是该Key对应的时序数据。

2. Index Entry表示一个索引字段，指向对应的Series Data Block。指向的Data Block由Offset唯一确定，Offset表示该Data Block在文件中的偏移量，Size表示指向的Data Block大小。Min Time和Max Time表示指向的Data Block中时序数据集合的最小时间以及最大时间，用户在根据时间范围查找时可以根据这两个字段进行过滤。

技术体系参考解读（8）：InfluxDB TSM文件结构



技术体系参考解读（9）：TSM引擎工作原理－时序数据读取



TSM在启动之后就会将TSM文件的索引部分(上图中间部分)加载到内存，数据部分因为太大并不会直接加载到内存。用户查询可以分为三步：

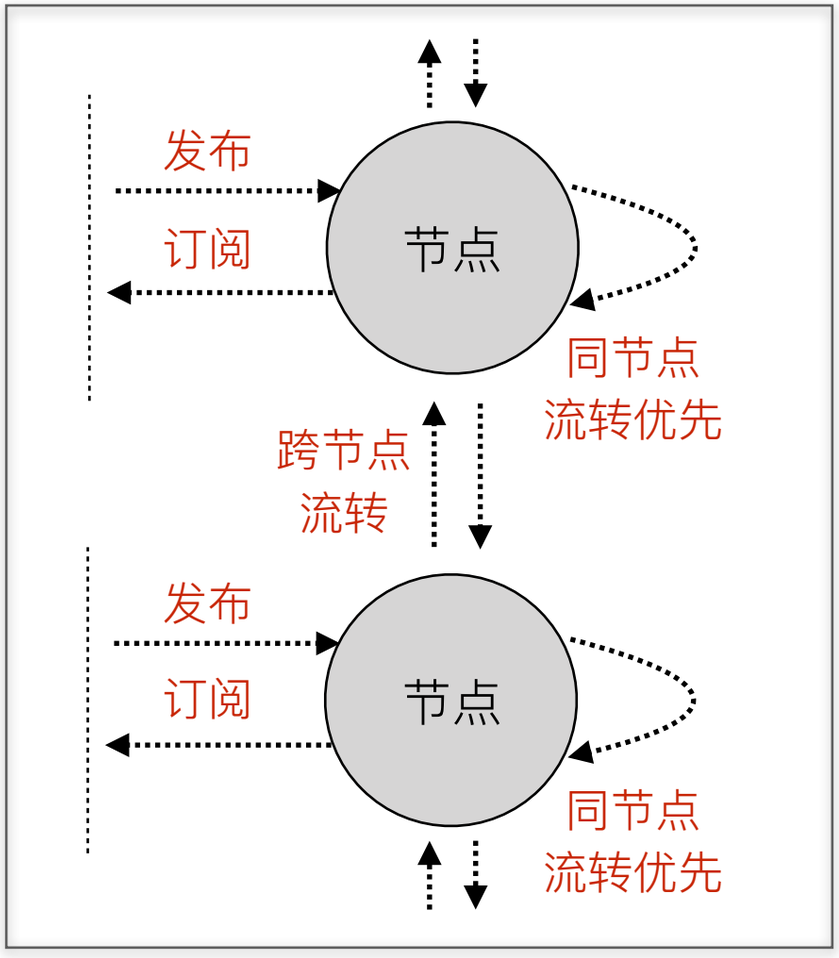
1. 首先根据Key找到对应的Series Index Block，因为Key是有序的，所以可以使用二分查找来具体实现

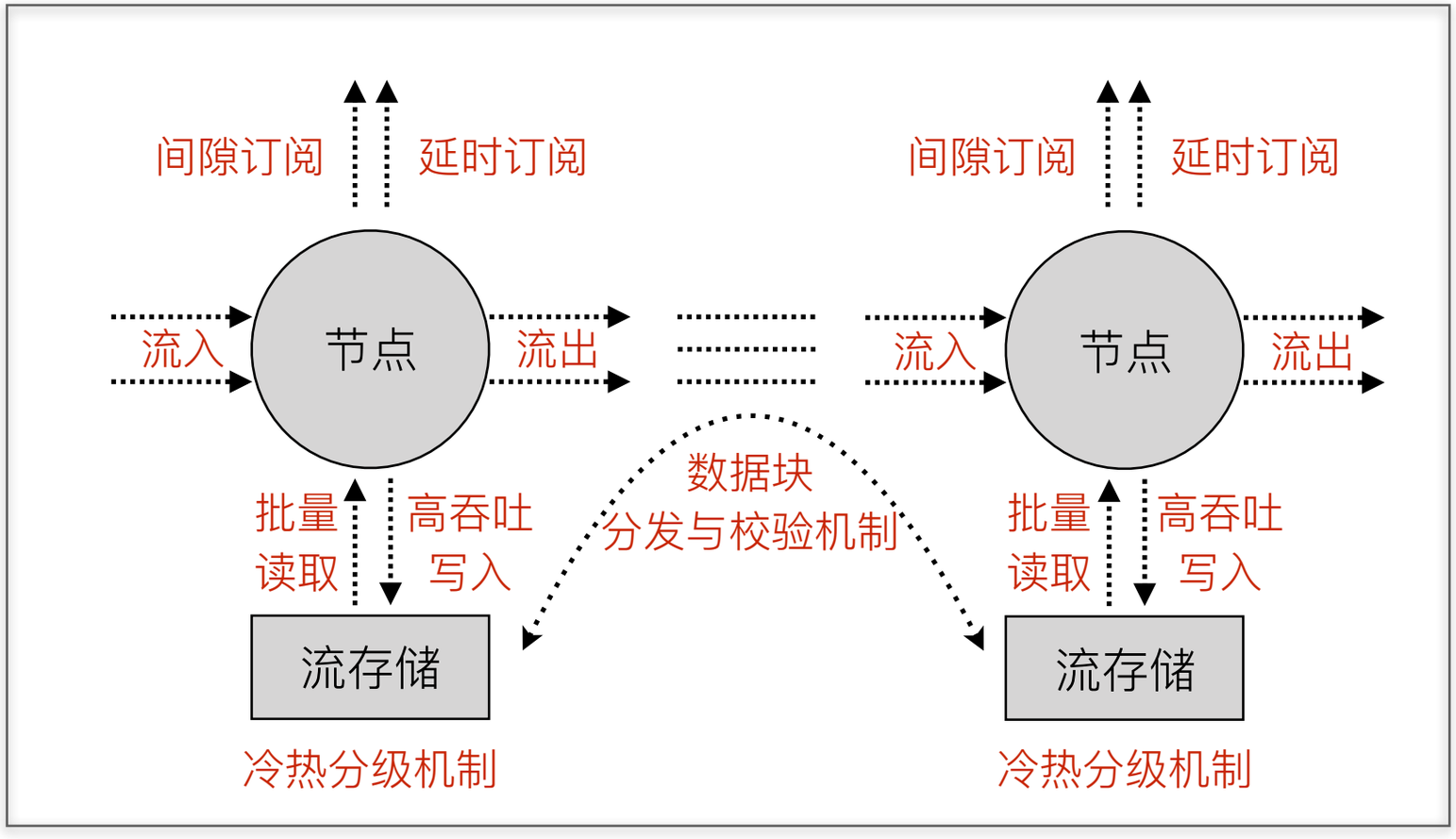
2. 找到Series Index Block之后再根据查找的时间范围，使用[MinTime, MaxTime]索引定位到可能的Series Data Block列表

3. 将满足条件的Series Data Block加载到内存中解压进一步使用二分查找算法查找即可找到

Beacon 存储模型设计（1）：时序数据流机制概述

- Beacon以数据(包含消息与控制逻辑)流转为主，即将发布者(连接节点)的数据源源不断流至订阅者(连接节点)，同时同节点(发布与订阅)优先处理。
- Beacon在实时在线流转数据时，附带持久化操作，即在该节点进行时序数据流存储。
- Beacon在只有发布没有订阅的非实时离线场景中，时序数据流存储相当于间隙/延时(通过特殊标示Topic)订阅机制。订阅者通过设置时间范围进行批量下载/分发数据。
- Beacon依据不同渠道，默认设置不同的失效时间。超过失效时间，节点上的数据块将会被清理。失效时间过短，将不会进行基于数据块的分发。
- Beacon存储模型，以写入为主，读取为次。写入机制类似于时序数据库，读取机制则非常简单，由于没有指标这类数据主体，从而不需要时序数据库的多维/聚合查询这类的功能设计。



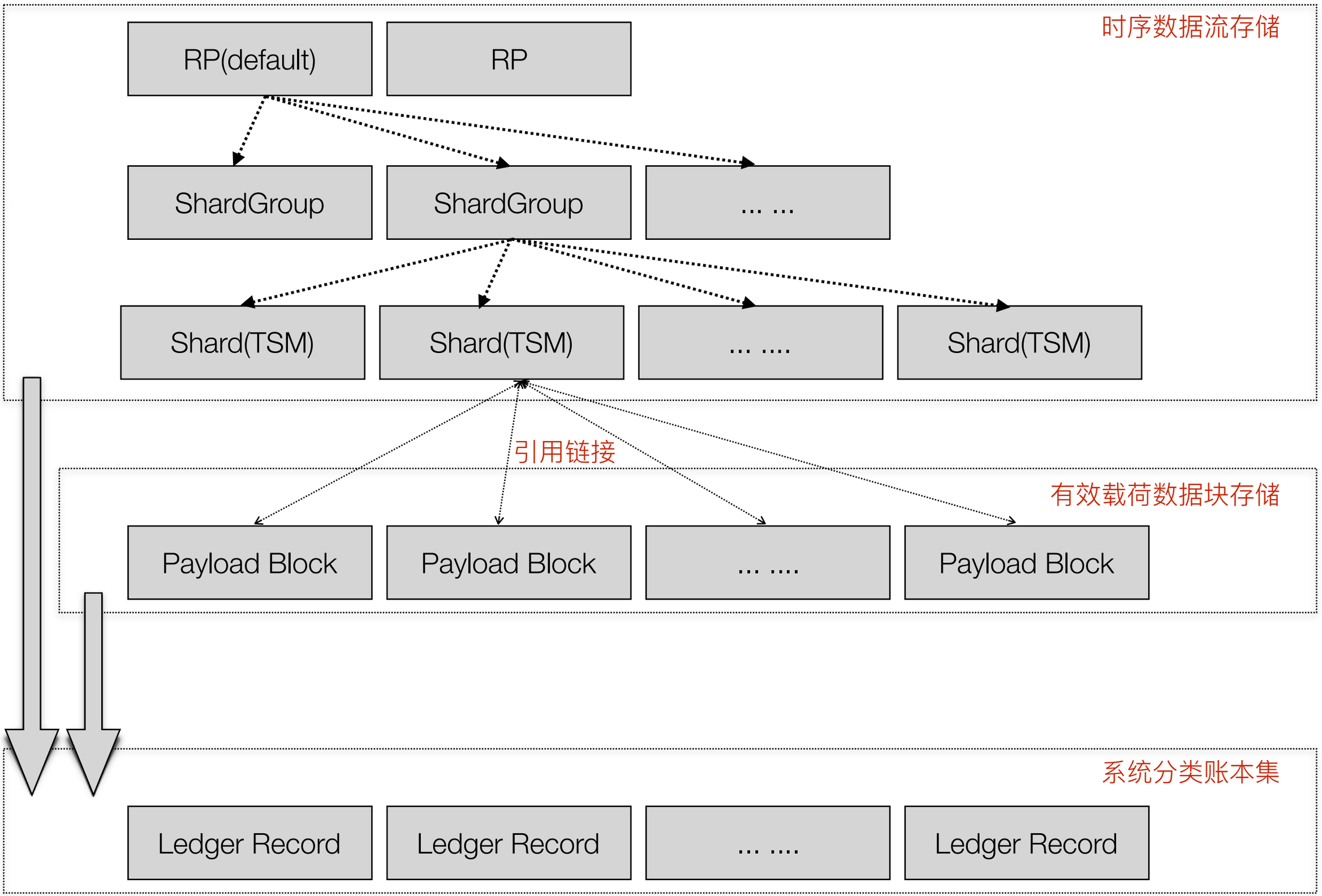


Beacon 存储模型设计（2）：时序数据流机制概述

- Beacon中的时间线：(1)发布者应当在Payload中，设置数据产生时间戳；(2)接入节点，会以接入时间戳作为数据流传递的起始；(3)中转节点，会陆续添加对应的时间戳；(4)订阅节点，也会附加对应的时间戳；(5)多订阅节点中的最后一个，作为数据流传递的终止；(6)存储机制中，也会记录写入时间戳，并在数据块同步时，附带同步/校验其流转的时间戳序列；
- Beacon中的数据单元标识码：(1)发布者的Hash(Payload),将是全局唯一的统一标识码；(2)发布者不能提供Hash(Payload)情况下，接入节点将做Hash计算；(3)中转与订阅节点，将会校验该标识码；(4)Payload将会在多个节点存在，若它大于某个阈值，则直接作为Payload Blob数据块进行分发(超大的，接入节点会进行分割)，若它小于某个阈值，则会依据时间线，合并成Payload数据块后分发；(5)Payload 数据块与对应InfluxDB中的Series Data Block，不是一回事。它是指Payload内容本身形成的类似于IPFS的数据块；
- Beacon中的数据块：(1)有阈值设定，控制数据块大小；(2)类似于IPFS，也会做Hash计算；(3)类似于IPFS中的BitSwap，进行数据块交换；(4)不同于IPFS，没有文件名或对应的Hash码，它由时间区间驱动的，将符合条件的时间戳串接而成；(5)不同于热数据主要依赖节点数据流转发，冷数据是通过对等节点的数据块交换完成数据分发与流转；(6)可从多个节点，同时批量下载数据；
- Beacon存储模型：(1)对应InfluxDB中的measurement，则采用Topic的Hash码；(2)对应InfluxDB中的datasource(tags)，则有Publisher<ClientId>，Forwarders<NodeIds,如前节点、当前节点、后节点>，Subscriber<ClientId>等；(3)对应InfluxDB中的Fields，则有数据主体Hash(Payload)、Size(Payload)，Offset(转发/写入的时间戳偏差等，用于Beacon性能优化)；(4)列存储，用户数据下载只需要数据主体这一Field，网络性能优化，只需要Offset这一Field；

Beacon 存储模型设计（3）：时序数据流机制概述

- Beacon 存储模型涉及TSM引擎、有效载荷数据块、分类账本记录等部分；



Beacon 存储模型设计（4）：分类账本集概述

- Beacon的时序数据流在节点之间进行分发与传递活动记录，会保存到相关分类账本记录集中。
- Beacon的有效载荷数据块，在节点间进行交换的记录，也会保存到相关分类账本记录集中。
- Beacon的节点，依据分类账本记录集，可追踪时序数据流活动和数据块交换的整个历史记录，避免相关内容被篡改。

- 无论是数据流分发与传递，还是数据块交换，都能基于全局分类账本所提供的交易系统，使得一个交换或计量的激励机制成立。
- 它利用类似BitSwap信用和负债的策略机制，进行动态的调节。
- 交换的负债比例公式：R = bytes_sent/bytes_recv + 1
- 转发的负载比例公式：R = forward_call/forward_request + 1
- 其中负债比是信任的衡量标准，之前成功的互换/转发的节点会宽容债务，而对不信任不了解的节点会严格很多。(1)给创造很多节点的攻击者(sybill攻击)提供一个障碍；(2)保护之前成功合作节点的关系；(3)最终阻塞那些关系恶化的节点之间的通信，直到它们被再次证明；