GLCC 2022 项目申请书

针对时序数据的内存管理和数据结构优化

项目摘要

目前 OpenMLDB 底层存储引擎基于随机访存模式的 skiplist 数据结构，以及使用 TCMalloc 来进行内存空间的申请和管理。这种设计导致了容易产生内存碎片，以及由于随机访存带来的性能下降。在这个项目中，要求开发者从数据结构或者内存申请算法优化方向入手，设计针对时序数据的自定义缓存友好的数据结构或者定制化的内存申请管理算法，来改善内存访问效率以及内存碎片的问题，并且整合到 OpenMLDB 项目中，提升系统的内存整体使用率以及访问效率。

提案人：　　房森 wirth.fang@foxmail.com
导师：　　邓龙 denglong@4paradigm.com

项目概览

产出介绍

对OpenMLDB 底层/线上存储引擎：

- ☐ 梳理引擎结构，调研制定合理的解决方案
- ☐ 完成针对时序数据的内存管理和数据结构优化工作（二选一）：
  - ☐ 设计针对时序数据的自定义缓存友好的数据结构
  - ☐ 定制化的内存申请管理算法
- ☐ 实现相关算法并整合到 OpenMLDB 的线上存储引擎
- ☐ 产出系统化报告
- ☐ 考虑投递顶会paper

项目目标及技术实现梳理

首先，OpenMLDB 是一个开源机器学习数据库，提供线上线下一致的生产级特征平台。在人工智能工程化落地过程中，企业的数据和工程化团队 95% 的时间精力会被数据处理、数据校验等相关工作所消耗。为了解决该痛点，MLDB应运而生。

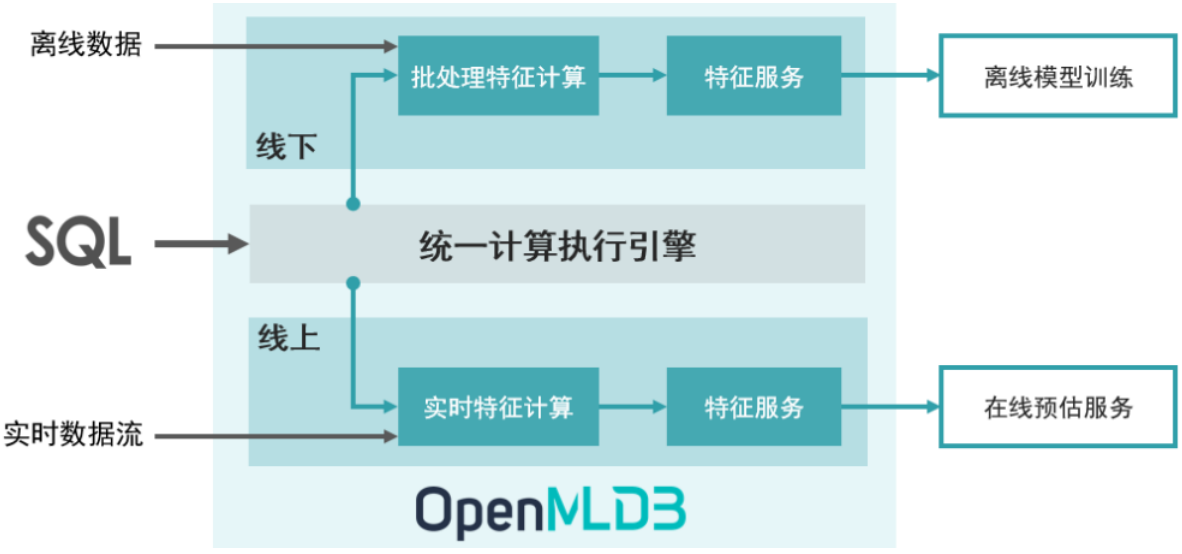
OpenMLDB 致力于解决 AI 工程化落地的数据治理难题，并且已经在上百个企业级人工智能场景中得到落地。OpenMLDB 优先开源了特征数据治理能力，依托 SQL 的开发能力，为企业级机器学习应用提供线上线下计算一致、高性能低门槛的生产级特征平台。

如果要对一个项目进行二次开发，首先要做的是读懂他的源码，我使用docker克隆了MLDB的仓库，配置好了开发环境，查阅了官网文档，对OpenMLDB的文件目录理解如下：

```
Python SDK
python
 ├─── openmldb
 |    ├─── dbapi                // dbapi接口封装
 |    ├─── native               // swig自动生成的代码
 |    ├─── sdk                  // 调用底层c++接口代码
 |    ├─── sqlalchemy_openmldb  // sqlalchemy接口封装
 |    ├─── sql_magic            // notebook magic
 |    └─── test                 // 测试相关
```

Hybridse SQL 引擎

```
hybridse/
 ├──── examples          // demo db和hybrisde集成测试
 ├──── include           // 代码的include目录，里边结构和src基本一致
 ├──── src
 │     ├──── base         // 基础库目录
 │     ├──── benchmark    // benchmark相关
 │     ├──── case         // 测试case相关
 │     ├──── cmd          // 封装的demo等
 │     ├──── codec        // 编解码相关
 │     ├──── codegen      // llvm代码生成相关
 │     ├──── llvm_ext     // llvm符号解析相关
 │     ├──── node         // 逻辑计划、物理计划中的节点定义，表达式、类型节点定义
 │     ├──── passes       // sql优化器
 │     ├──── plan         // 生成逻辑计划
 │     ├──── planv2       // zetasql语法树转化成节点
 │     ├──── proto        // protobuf定义
 │     ├──── sdk          // sdk相关
 │     ├──── testing      // 测试相关
 │     ├──── udf          // udf和udaf的注册生成等
 │     └──── vm           // sql物理计划和执行计划的生成以及sql编译和执行入口
 └──── tools     // benchmark相关
```



上图显示了基于 OpenMLDB 的 FeatureOps 的基本使用流程

在分析了MLDB代码之后，我初步制定了三个目标：

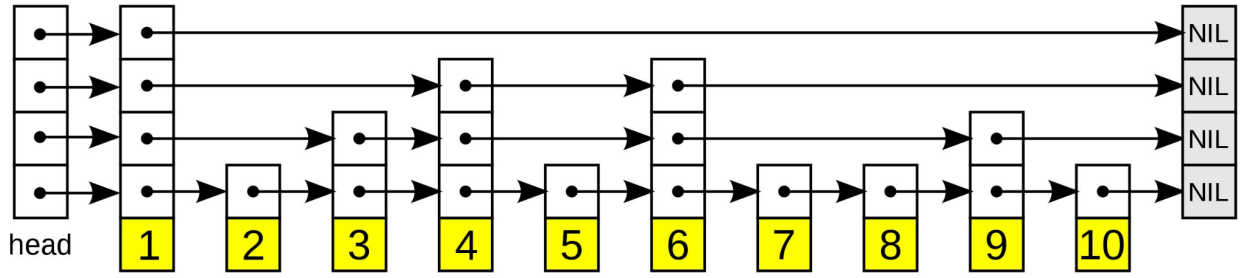
- [ ] 完成针对时序数据的内存管理和数据结构优化工作（二选一）：
    - [ ] 设计针对时序数据的自定义缓存友好的数据结构
    - [ ] 定制化的内存申请管理算法
- [ ] 实现相关算法并整合到 OpenMLDB 的线上存储引擎
- [ ] 产出系统化报告

完成针对时序数据的内存管理和数据结构优化工作

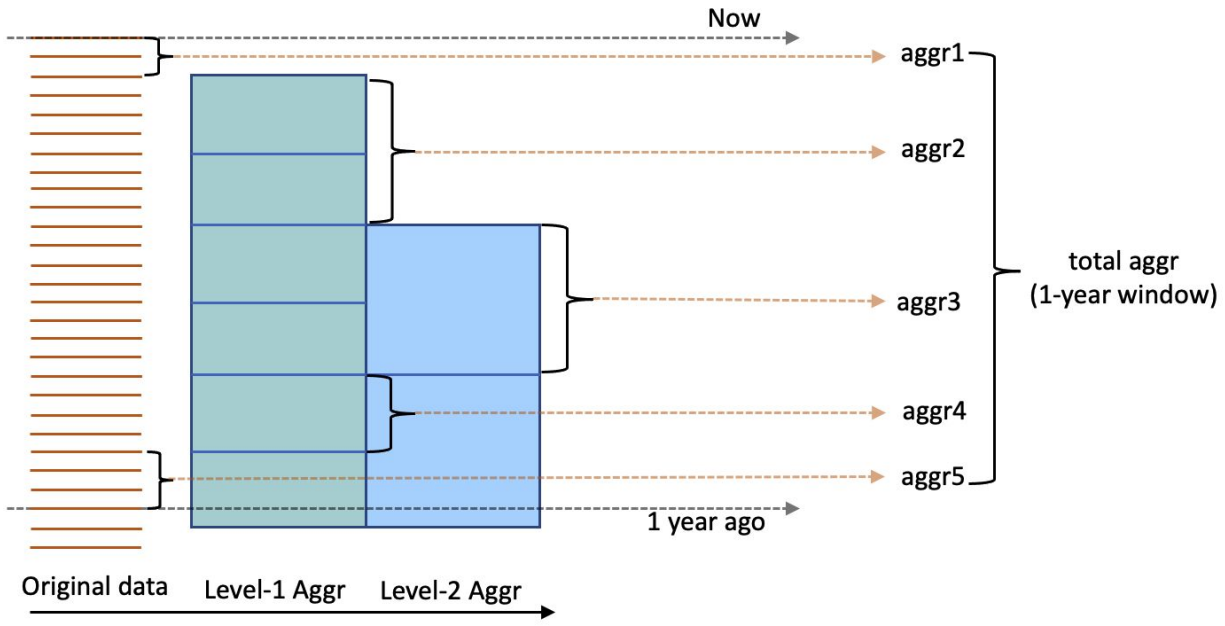
OpenMLDB 内部整合了批处理和实时两套 SQL 引擎，分别用于应对机器学习的离线开发和在线推理场景，并且保证线上线下的一致性。其中，默认基于内存的实时 SQL 引擎经过了充分的性能优化，可以达到毫秒级的计算延迟。其主要包含了两个核心优化技术：

- 双层跳表结构（double-layer skip list）：专门为时序数据访存优化设计的内存索引数据结构
- 预聚合技术（pre-aggregation）：针对窗口内数据量巨大的场景下，为减少重复计算的性能优化技术

跳表（skip list）由 William Pugh 在 1990 年提出，其论文为：Skip Lists: A Probabilistic Alternative to Balanced Trees。跳表采用的是概率均衡而非严格均衡策略，从而相对于平衡树，大大简化和加速了元素的插入和删除。跳表可以看做是在链表的基础数据结构上进行了扩展，通过添加多级索引，来达到快速定位查找的操作。其既有链表的灵活数据管理优势，同时对于数据查找和插入的时间复杂度均为 O(logn)。

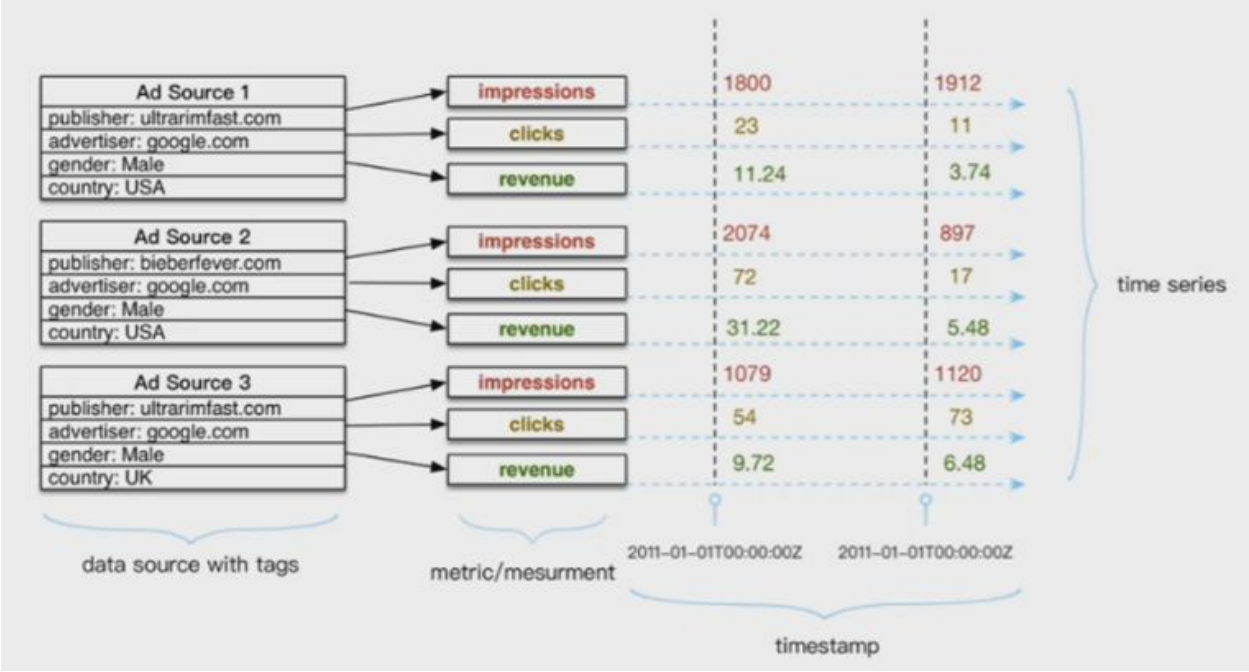


预聚合技术：在一些典型场景中（比如画像系统），时序特征的窗口内数据量可能很大（比如窗口的时间跨度横跨三年），我们把这种时序特征称之为“长窗口”特征。为了改善长窗口的性能，我们引入了预聚合技术。通过预聚合，聚合特征的部分结果，会在数据插入的时候，提前计算好；线上实时特征计算时，只需要把计算好的预聚合结果进行规约，就可以快速得到最终的聚合特征。预聚合数据相比原始数据，数据量极大地降低，可以达到毫秒级的计算延迟。



从设计针对时序数据的自定义缓存友好的数据结构入手

在讨论如何优化时序数据模型的数据结构之前，有必要简单回顾一下时序数据的几个基本概念，如下图所示：



上图是一个典型的时序数据示意图，由图中可以看出，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；和横坐标不同，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

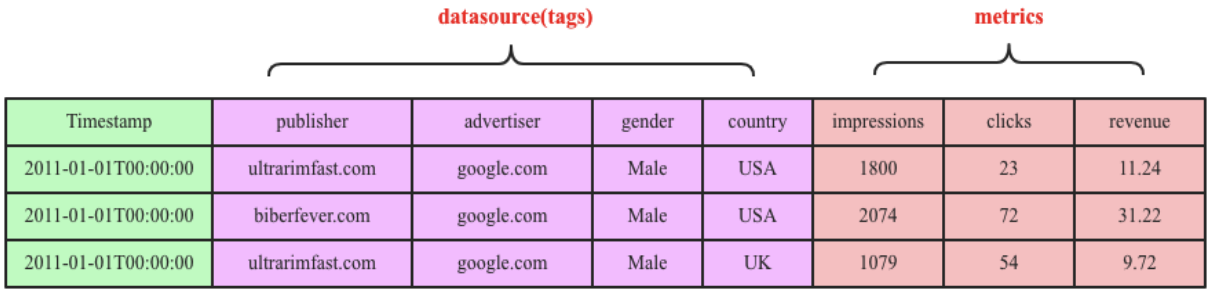
四个维度值唯一表示，metric表示待收集的数据源指标。一个数据源通常会采集很多指标（metric），上图中广告数据源就采集了impressions、clicks以及revenue这三种指标，分别表示广告浏览量、广告点击率以及广告收入。我们可以借鉴其它数据库数据结构：

OpenTSDB(HBase)时序数据存储模型

OpenTSDB基于HBase存储时序数据，在HBase层面设计RowKey规则为：metric+timestamp+datasource(tags)。HBase是一个KV数据库，一个时序数据(point)如果以KV的形式表示，那么其中的V必然是point的具体数值，而K就自然而然是唯一确定point数值的datasource+metric+timestamp。这种规律不仅适用于HBase，还适用于其他KV数据库，比如Kudu。

Druid时序数据存储模型设计

和HBase和Kudu这类KV数据库不同，Druid是另一种玩法。Druid是一个不折不扣的列式存储系统，没有HBase的主键。上述时序数据在Druid中表示是下面这个样子的：



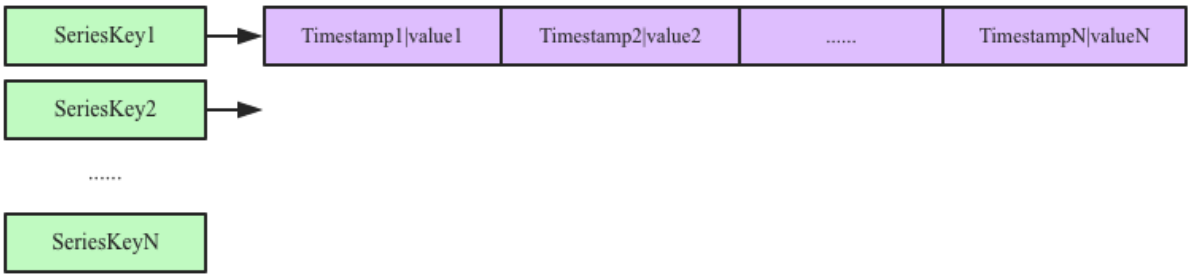
Timestamp	publisher	advertiser	gender	country	impressions	clicks	revenue
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	USA	1800	23	11.24
2011-01-01T00:00:00	biberfever.com	google.com	Male	USA	2074	72	31.22
2011-01-01T00:00:00	ultrarimfast.com	google.com	Male	UK	1079	54	9.72

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。细心的童鞋就会说了，这样存储，依然会有数据源（tags）大量冗余的问题。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。

InfluxDB时序数据存储模型设计

相比OpenTSDB以及Druid，可能很多童鞋对InfluxDB并不特别熟悉，然而在时序数据库排行榜单上InfluxDB却是遥遥领先。InfluxDB是一款专业的时序数据库，只存储时序数据，因此在数据模型的存储上可以针对时序数据做非常多的优化工作。

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datasource(tags)+metric，时序数据写入内存之后按照seriesKey进行组织：



Beringei时序数据存储模型设计

Beringei是今年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。我们知道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：

1. 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。

2. Beringei目前没有倒排索引机制，因此对于多维查询并不高效。

从定制化的内存申请管理算法入手

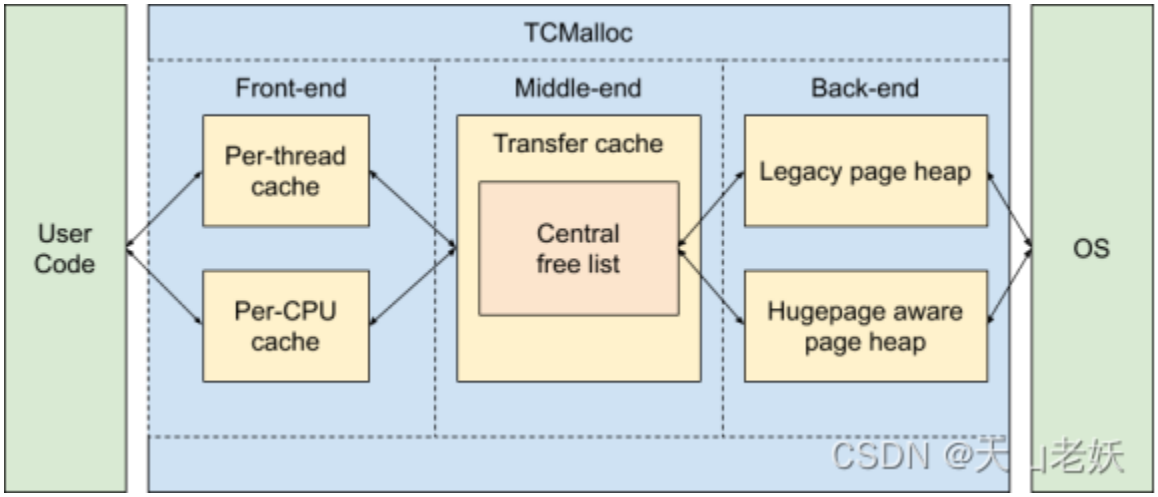
目前OpenMLDB使用TCMalloc作为内存分配方案，TCMalloc(Thread-Caching Malloc，线程缓存的malloc）是Google开发的内存分配算法库，最初作为Google性能工具库 perftools 的一部分，提供高效的多线程内存管理实现，用于替代操作系统的内存分配相关的函数（malloc、free，new，new[]等），具有减少内存碎片、适用于多核、更好的并行性支持等特性。

他的架构由三部分组成：

Front-end（前端）：负责提供快速分配和重分配内存给应用，由Per-thread cache和Per-CPU cache两部分组成。

Middle-end（中台）：负责给Front-end提供缓存。当Front-end缓存内存不够用时，从Middle-end申请内存。

Back-end（后端）：负责从操作系统获取内存，并给Middle-end提供缓存使用。



TCMalloc中每个线程都有独立的线程缓存ThreadCache，线程的内存分配请求会向ThreadCache申请，ThreadCache内存不够用会向CentralCache申请，CentralCache内存不够用时会向PageHeap申请，PageHeap不够用就会向OS操作系统申请。

TCMalloc将整个虚拟内存空间划分为n个同等大小的Page，将n个连续的page连接在一起组成一个Span；PageHeap向OS申请内存，申请的span可能只有一个page，也可能有n个page。

可借鉴的其他内存分配器：

1. Sequential Fit

是基于一个单向或双向链表管理各个blocks的基础算法，因为和blocks的个数有关，性能比较差。这一类算法包括Fast-Fit, First-Fit, Next-Fit, and Worst-Fit。

2. Segregated Free List（离散式空闲列表）

使用一个数组，每个元素是存储特定大小内存块的链表，它们所代表的大小并不是连续的，所以称为离散。经典的dlmalloc使用的就是这个算法。数据元素，参照上面的图就可以理解了。TLSF算法则是基于此进行了改进。

3. Buddy System

这是由一代大师Donald Knuth提出，后续产生许多的改进版本。最大的作用是解决外部碎片(external fragmentation)，详细的算法，参考这篇（浅析Linux内核内存管理之Buddy System）。

4. Indexed Fit

以某种数据结构为每个block建立索引，以求可以快速存取。一般以一个二叉树结构实现。比如使用Balanced Tree的Best Fit allocator, 以及基于Cartesian tree 的Stephenson Fast-Fit allocator。这类算法的性能比较高，也比较稳定。

5. Bitmap Fit

这类算法只是索引方法不同，使用以位图式字节表示存储单元的状态。它的好处是使用一小块连续的内存，响应性能更好。Half-Fit就属于这类算法。

随着技术演进，现在主流的allocators, 基本上都是综合运用了两类以上的算法。

不同内存分配器优劣势对比

ptmalloc 劣势:多线程下的性能及内存占用(线程间内存无法共享),并且内存用于存储metadata开销较大，在小内存分配上浪费比较多。优势:算是标准实现。

tcmalloc 劣势：因为算法的设计，占用的内存较大。优势:多线程下的性能。当前使用。

jemalloc 优势: 内存碎片率低，多核下性能较tcmalloc更好。参考附17。

这两个方案的目标都是为了解决目前内存碎片以及随机访存问题，最终效果是完成针对时序数据的内存管理和数据结构优化工作。

开发进度表

社区联络期 (June 25 - June 30)

这个阶段的任务是深入了解项目，并可能在此过程中解决一些可能会干扰进度的因素与预案。对所需的技术做一些研究，与社区开发人员和导师交谈，并改变一些技术步骤或计划。

编码阶段一 (July 1 - July 15)

编码阶段一有2周的时间，并在阶段结束后进行评估计划质量与风险。第一阶段的目标是

- [ ] 梳理引擎结构，调研制定合理的解决方案

编码阶段二 (July 15 - Aug 30)

编码第二阶段有6周时间，阶段结束后进行可能的中期评估，有两个初步目标，至于其具体执行细节有待商酌。

- [ ] 完成针对时序数据的内存管理和数据结构优化工作（二选一）：
    - [ ] 设计针对时序数据的自定义缓存友好的数据结构
    - [ ] 定制化的内存申请管理算法
- [ ] 实现相关算法并整合到 OpenMLDB 的线上存储引擎

编码阶段三 (Sep 1 - Sep 23)

编码第二阶段有3周时间，如果任务进行顺利，我们将产出系统化报告，和考虑将相关材料发表为会议论文。我们注意到社区中的一些issue与，我们可以考虑完成它们。

- [ ] 产出系统化报告
- [ ] 考虑投递顶会paper

一旦实现了本个阶段的目标，我们就会检查代码并修复前面代码中的任何已知bug。编写代码应该被文档化。然后我们的工作基本完成了，然后是最后的评估。

缓冲时间 (7 days)

有至少7天的缓冲时间，以防前几周发生的事情没有按计划进行。我认为一些工作可能会在两周内完成，所以我们实际上会进展得更快，并拥有更多的容错能力。

关于我

申请信息

名称:	房森 (Wirth)
Email:	wirth.fang@foxmail.com
Github:	github.com/FangSen9000
时区:	UTC+08:00 (China)
地区:	郑州, 中国
Education:	河南大学, 澳大利亚Victoria University计算机科学与技术双学位
Telephone:	+86 18143465655
CSDN blog:	Wirth‘s blog（Chinese）

Self-introduction

至于我，我是一个探路者，我热爱开源，非常享受GSoC和OSPP或者其他开源活动的氛围。我在河南大学虚拟现实实验室工作由阎朝坤教授指导，热爱技术的落地实现，身经百战，并且很高兴在OpenMLDB社区与邓龙导师一起讨论协商并且完成这个项目。我很高兴能参与这个项目，我对它的前景充满信心，即使在GLCC结束后，我也会继续为这个项目做出贡献。

参考资料

[1] OpenMLDB基于 skiplist 的内存索引结构：OpenMLDB

[2] OpenMLDB目前使用的TCMalloc算法:TCMalloc

[3] OpenMLDB目前使用的跳表算法:Skip list

[4] Optimizing in-memory database engine for AI-powered on-line decision augmentation using persistent memory

[5] 开源机器学习数据库OpenMLDB 产品介绍

[6] OpenMLDB 在线模块架构解析

[7] OpenMLDB 实时引擎核心数据结构和优化解析

[8] 官网文档 — OpenMLDB documentation

[9] C++性能优化（九）—— TCMalloc

[10] 内存分配器 (Memory Allocator)

[11] 附件：OpenMLDB提案人个人扩展简历

[12] 附件：demo-dashboard: 这是我为OpenMLDB项目存放Demo的地方(github.com)