[优化方案](http://wiki.intra.xiaojukeji.com/pages/viewpage.action?pageId=427157020)\_v3

[优化方案 1](#_Toc51148513)

[问题域 1](#_Toc51148514)

[具体表现 1](#_Toc51148515)

[优化方案 2](#_Toc51148516)

[优化方向 2](#_Toc51148517)

[具体优化项 2](#_Toc51148518)

[1.去锁 2](#_Toc51148519)

[2.重新设计更新实体操作。 3](#_Toc51148520)

[3.projectL2Cache重构 4](#_Toc51148521)

[核心调用流程优化 6](#_Toc51148522)

[1.查询 6](#_Toc51148523)

[2.load表 6](#_Toc51148524)

[3.unLoadHiveTables 7](#_Toc51148525)

[4.saveModelDesc 8](#_Toc51148526)

[5.deleteModel 9](#_Toc51148527)

[6.saveCubeDesc 9](#_Toc51148528)

[7.deleteCube 10](#_Toc51148529)

[8.更新segment 11](#_Toc51148530)

[9.其他 11](#_Toc51148531)

# 问题域

模型较多时，建模慢，建模影响查询。

## 具体表现

1.查询响应时间受建模影响较大。

2.单服务进程上连续串行建模慢。

3.在多个服务进程上连续建模会有失败。

# 优化方案

## 优化方向

1.查询尽量不要被不相干的元数据更新操作影响。

2.更新操作直接刷新本地缓存，不需要通过web广播来通知自己刷新缓存，会跟后续建模操作产生锁冲突。

3.优化读写锁，尽量不对逻辑块加锁。写应该用cas语义失败重试或使用并发性能更好的容器避免加锁，读的场景评估是否允许脏读，不允许也使用更细粒度的锁。

4.精细化刷新缓存，只更新有变化的缓存。

5.减少业务链路里重复的缓存刷新。

6.并发建模请求负载均衡后的多个服务进程，容易保存失败，应该有基于cas的内部重试。

7.广播元数据变化大多会清空LRU2Cache，这个是给查询选cube用的，除了cube变更状态/删除，修改segment状态为Ready/删除等有限几种变化，其他情况都不用改，因为不影响选cube。

## 具体优化项

### 1.去锁

去掉manager和个别service上的读写锁，CaseInsensitiveStringCache 用ConcurrentSkipListMap,已能保证大多数操作的并发一致性。

CachedCrudAssist引用的CaseInsensitiveStringCache也是能保证并发性的。

projectinstance的锁到具体内部实体list，set上。

### 2.重新设计更新实体操作。

针对一般实体的更新，先更新存储再让缓存可读。  
单个服务进程，先更新存储，成功了再更新缓存。潜在的风险，是后更新缓存的被前置更新的覆盖。对表，model，cube来说，更新存储和更新缓存的操作必须加锁。  
多个服务进程，收到实体更新通知，为避免潜在的多个并发“读存储-写map”的顺序和服务自身的“写存储-写map”的不一致，必须加锁，加锁的最小单位是资源本身。

思考“读存储-写map”是否要加锁？如果加读写锁，每个res一个锁锁有点多。如果是同步锁，读同一个资源的存储就无法并发了，现在代码里reload逻辑还是挺多的。不锁潜在风险是并发修改或一个修改多个reload读时，可能把旧的实例后放入缓存了，导致和存储的不一致。决定采用都加同步锁，同时减少不必要的reload，并且将reload逻辑细分为reload并更新缓存和不需要更新缓存，单纯reload不加锁  
instance对象当reload会new新的，所以更新存储和缓存时，应该跳出实例，对资源的唯一路径加锁。  
对于new实体操作，并发只允许一个保存成功。  
udpate实体操作，单一操作要么全部成功要么全部失败。如果直接在旧instace上更新，可能会保存另一个更新操作的一半变更。要么更新实例属性时就加资源路径锁，要么clone实例更新，更新成功后再更新map映射。  
前者的风险是更新属性后可能还有一堆其他逻辑，当后续逻辑要更新其他实体时有死锁风险。后者在同一个服务内更容易更新失败，如果要提高操作成功率需要reload重新更新cas更多次，并且对于实体整体的更新无法reload。并发操作是少见的，个人倾向cas。  
再加上通过haproxy请求多个服务进程的场景，有2点需要注意。1.可能并发有冲突cas失败，进程内的锁不管用。全局锁，或者cas失败操作失败，或者cas失败了reload重试cas。2.连续操作时前置操作返回成功了，新操作请求的服务进程还没更新对应的缓存，可能找不到元数据。  
考虑实际场景，单个表，model，cube自身的并发创建/更新，一般只出现于同一个请求连续发了多次，并发创建/更新只成功一个影响并不大。所以实现可以clone/reload/new的基础上修改，再cas，失败就直接操作失败。  
对于prj更新table/model/cube/其他属性，如果要允许并发，需要拆project元数据为多个。如果保持现在的prj整体为最小粒度，在并发时必然有冲突。对更新其他属性，直接在实体上修改，cas失败就返回异常。但更新prj的子元素列表失败，因为已经保存子元素了，直接操作失败会留有子元素的脏数据。这本质是同时更新子元素和prj的操作原子性问题。没有事务的存储引擎是无法保证多个更新的原子性的，取舍为reload prj并更新子元素cas，重试几次后还是失败则操作失败。如果真的由于存储引擎问题或并发太高问题产生子元素的脏数据，概率很低，用清理工具删除元数据即可。再就是更新segment，这个可以采用prj更新子元素相同逻辑来处理，目前应该就是这么做的。

对于连续操作在多个服务进程上的串行，如果发现依赖实体还无缓存，可以尝试主动reload一次依赖实体，依然不存在再报错。如果更新依赖实体再来新建实体，由于缓存还未刷新导致新建实体检查正确性失败，非常规操作，直接失败让客户端重试即可。  
对lru2cache，资源缓存换了对象实例，可能会影响lru2cache映射。但考虑还没被引用的才可以直接修改信息，所以收口lru2cache的变化，只有在会影响查询路由的cube变化时再统一更新，其他时候直接读即可。

对于其他读缓存还未更新的场景，直接按读缓存处理，潜在风险目前只想到当依赖实体被删除时基于依赖并发创建实体(例如删除model的同时基于该model并发创建cube)可能出现问题。考虑这是非常规操作，依赖都要删除了还要基于依赖去创建本身也是有问题的，后果是新创建的实体不可用，不影响已有查询和其他实体创建，影响可控，用清理工具删除元数据即可。

CachedCrudAssist的save方法存储，put缓存并更新，reload，putlocal。目的是cache不使用传进来的实体，以允许调用者继续更新传进来的实体。

### 3.projectL2Cache重构

#### 核心原则

最重要的方法是getRealizationsByTable，每次查询都用到。其他listDefinedTables，listAllRealizations，listExposedColumns，listExposedTables方法也会被用到。  
核心原则是读缓存的性能优先，写缓存不堵塞读，多进程场景下允许短时间内读到脏缓存。  
写缓存操作最小粒度加锁，保证高并发场景下的最终一致性。  
读缓存尽量不加锁，读访问的容器对象要么是支持并发的，要么只通过new重置引用来更新。projectcache的tableCache每次更新new一个重新put，其他的容器字段改并发容器。  
projectCaches改为ConcurrentHashMap提高读性能。prjCache的realizations由Set<realization>改ConcurrentHashMap，tables由Map<String, TableCache>改ConcurrentHashMap，exposedTables改为copyonwriteset

#### 更新

更新操作拆分为PrjCache整体load和局部更新。  
**loadPrjCache**  
只有当系统第一次加载某个prj的projectcache时，要double check加锁等缓存加载完再读，其他时候读操作都不受锁影响。  
reloadprojectcache会加一个prj的锁，先将旧projectcache的reload标签设置为true，再重新load一次prjCache对象，put回map。

**局部更新**  
**最终一致性保证**  
更新/删除 table或realization时局部更新prjCache。先get prjcache并判断prjcache的reload是否为true，true则加prj的锁等待reload完成后再重新get prjCache，在新的prjcache上做更新操作。  
假设reloadprjcache前已经有更新table/realization操作正在执行了，则更新table/realization会在旧prjCache上操作。由于触发二级缓存更新realization的前提是一级缓存已经更新了，所以reload新的prjCache是可以加载到table/realization的更新，不会有正确性问题。

**table操作**

更新table操作，当save table成功/收到同步table广播reloadLocal后触发。能更新说明没有依赖自己的model更没有realization，所以exposedTables不需要更新。对table name加锁，{tables putIfAbsent，get tableCache，从1级缓存中获取最新的tableDesc，设置tableDesc}。  
删除table操作，当删除table成功/收到删除table广播removeLocal后触发。exposedTables遍历一遍，找到同名的remove掉。tables也remove。

**realization操作**  
更新realization操作，当save realization/收到同步广播后触发。先对canonicalName加锁，{重新从1级缓存中获取最新的realization对象，如果prjCache中已存在相同更新时间的realization，说明最新的对象更新过了，不需要更新直接return。否则遍历tablesCache，把存在realization的tablesCache找出来，再计算new realization有哪些tablesCache，这些是要改变的tablesCache。遍历每一个要改变的tablesCache，对table name加锁，{new tableCache并init，put回tables。}}  
删除realization操作，当删除realization成功/收到删除realization广播removeLocal后触发。realizations中删除。遍历tablesCache，把存在realization的tablesCache找出来，加table名的本地锁。{new tableCache并init，put回tables}。

## 核心调用流程优化

### 1.查询

KylinUserManager.get去掉锁

ProjectManager.getRealizationsByTable, 即ProjectL2Cache.getRealizationsByTable容易由于元数据变化而被清空重新加载，后续逻辑链路不轻易清空ProjectL2Cache。

### 2.load表

ProjectManager.getProject去掉锁，

TableMetadataManager.getTableDesc去掉锁，getProjectSpecificTableDesc去掉锁，去掉从无prj属性的table中捞表的逻辑，现在必须要选prj的。

DataModelManager.getModelsUsingTable去掉锁,cubeManager.listAllCubes去掉锁

CubeDescManager.getCubeDesc,TableMetadataManager.saveSourceTable去锁，

TableMetadataManager.getTableExt，saveTableExt去掉锁，

ProjectManager.addTableDescToProject去掉锁，校验表名存在的逻辑仅当刚保存还没返回就被删除请求了时有用，去掉。内部projectInstance.addTable可能存在并发，用法问题概率极低，tables对象TreeSet，

调用addTable方法的for循环外围加synchronized（tables）。不需要clearL2Cache，添加个新的save方法调用。//TODO,save失败了reload重试

#### 广播

同步表

由于如果有依赖该表的model则无法重新load表，

所以对本服务来说，重新reload表，通知notifyProjectSchemaUpdate更新，CacheService清理读缓存，

DataModelSyncListener重新reload表,model不需要，CubeSyncListener，CubeDescSyncListener都不需要，可以不广播自己。

对其他服务来说，只需要reload表，其他不需要。SrcTableSyncListener去掉锁和notifyProjectSchemaUpdate

同步表ext

对本服务，reload，直接可以去掉，对其他服务，SrcTableExtSyncListener去掉锁

同步prj

对本服务，可以不回调。对其他服务，只reload，可以不回调project变化，所以prj变化的回调可以添加更精细化调用的方法

### 3.unLoadHiveTables

TableACLManager.deleteTableACLByTbl去掉锁 //TODO,判断有acl变化再save

ProjectManager.removeTableDescFromProject去掉锁，不再调用TableMetadataManager.getTableDesc判断存在（外围已经做了），projectInstance.removeTable加锁。//TODO,save失败了reload重试

DataModelManager.getModelsUsingTable去掉锁

TableMetadataManager.removeTableExt, removeSourceTable去掉锁，

SourceManager.getCachedSource,增加fordelete的方法，找不到就返回null，不做后续的unloadtable。//只为kafaka有效

#### ****广播****

同添加表。

### 4.saveModelDesc

DataModelManager.getDataModelDesc，createDataModelDesc 去锁，

TableMetadataManager.getAllTablesMap去锁，

ProjectManager.addModelToProject去锁，不调用removeModelFromProjects，projectInstance.addModel内部加锁synchronized（tables），也不需要clearL2Cache，复用新save。

#### 广播

同步model

本进程不需要reload，也不需要通知projectschema更新。其他进程只reload。不需要清理读缓存，reload table，reload all models，reload all cubes

同步prj

对本服务，可以不回调。对其他服务，只reload，可以不回调project变化，所以prj变化的回调可以添加更精细化调用的方法

### 5.deleteModel

CubeDescManager.listAllDesc去锁

DataModelManager.dropModel去锁，

ProjectManager.removeModelFromProjects,findProjectsByModel去锁,projectInstance.removeModel加锁，也不需要clearL2Cache，复用新save。

#### 广播

同上

### 6.saveCubeDesc

CubeManager.getCube,CubeDescManager.getCubeDesc去锁，

CubeDescManager.createCubeDesc去锁，save失败跟现在一致直接异常。

CubeManager.createCube去锁， 调动ProjectManager.moveRealizationToProject的逻辑换个方法，无锁，只addRealizationToProject不removeRealizationsFromProjects，

调用addRealizationEntry处加锁synchronized(realizationEntries).也不需要clearL2Cache，复用新save。

#### 广播

CubeDescSyncListener，CubeSyncListener

同进程不需要通知，其他进程reload cubedesc.不需要清理读缓存，reset表，reload cube，cube desc，l2cache。可以不回调project变化，所以prj变化的回调可以添加更精细化调用的方法

### 7.deleteCube

HybridManager.getHybridInstance去锁，

线上配置kylin.storage.clean-after-delete-operation设置为true

HybridCubeCLI.update 传参数，允许不调用ProjectManager.moveRealizationToProject方法

CubeManager.getCubesByDesc去锁

CubeManager.dropCube去锁，

removeRealizationsFromProjects，findProjects去锁，调用projectInstance.removeRealization处加锁synchronized(realizationEntries)

~~aclService.deleteAcl 去锁，aclMap改成允许并发的数据结构。~~

#### 广播

CubeDescSyncListener

本进程不需要通知，其他进程removeLocalCubeDesc去锁，不需要notifyProjectSchemaUpdate。

CubeSyncListener

本进程不需要通知,。其他进程removeCubeLocal，

HybridSyncListener

本进程不需要通知，其他进程cube分支去锁。

ProjectSyncListener

本进程不需要通知reloadproject， ~~但需要clearL2Cache~~。其他进程只要reloadProjectQuietly去锁~~和clearL2Cache~~。//TODO 精细化clearL2Cache，，先new一个构建好了再更新

### 8.更新segment

cubeManager.promoteNewlyBuiltSegments 去锁，updateCube去锁，//TODO 精细化clearL2Cachey

#### 广播

CubeSyncListener,HybridSyncListener

本进程不需要通知。其他进程只要reloadProjectQuietly去锁~~和clearL2Cache~~，不需要通知HybridSyncListener//TODO 精细化clearL2Cache

### 9.其他

将各manager其他方法的读写锁去掉，按需补充更细粒度的锁和保存重试。

**ProjectManager**

reloadProjectQuietly，

reloadAll，

listAllProjects，

getPrjByUuid，

createProject,

dropProject,

updateProject,

removeProjectLocal,

removeRealizationsFromProjects,

addExtFilterToProject,

removeExtFilterFromProject,

removeExtFilterFromProject,

touchProject,

getProjectOfModel,

findProjects,

findProjectsByTable

**TableMetadataManager**

reloadSourceTableQuietly

listAllTables

resetProjectSpecificTableDesc

reloadTableExtQuietly

listAllExternalFilters

getExtFilterDesc

saveExternalFilter

removeExternalFilter

**DataModelManager**

listDataModels

getModels

isTableInAnyModel

reloadDataModel

updateDataModelDesc

**CubeManager**

reloadAndListAllCubes

getCubeByUuid

clearSegments

updateCubeStatus

updateCubeDropSegments

dropOptmizingSegments

updateCubeSegStatus

updateCubeLookupSnapshot

reloadCube

reloadCubeQuietly

removeCubeLocal

appendSegment,refreshSegment,optimizeSegments,mergeSegments

**CubeDescManager**

reloadCubeDescQuietly

reloadCubeDescLocal

updateCubeDesc

removeCubeDesc

HybridManager

lockForRead

listHybridInstances

reloadHybridInstance

reloadAllHybridInstance

KylinUserManager若干

TableACLManager若干

ExecutableDao若干