背景和动机

在数据库中，索引是一个能支持快速查询的组件，一般的索引是建立在已有数据上，通过按照某种规则将数据组织起来，形成的存储在内存或者磁盘上的结构。查询是按照之前组织数据的规则在索引组件中进行查找，可以快速读取到对应的数据，同时减少数据访问量。早期的数据读写负载较为均衡，一般的B-Tree,HashTable等结构足以应，但随着科技的发展，尤其是在互联网领域，超大规模数据流输出越来越常见，早期针对磁盘设计的存储结构无法满足当前高吞吐量写入场景，例如B-Tree在更新的时候需要进行大量的随机写操作，严重降低了写入效率。LSM-Tree是90年代末被提出的一种键值存储的持久性数据结构，Google在2010年提出使用LSM-Tree思想的BigTable，一种在buffer中缓存所有写入请求后将其flush到磁盘文件，并对新旧文件进行merge的组织结构，并开源了Key-Value存储引擎LevelDB，随后LSM-Tree结构的数据库广泛应用于工业，包括RocksDB和其他许多数据库系统。

LSM-Tree的设计能达到高速写目的，但同时它降低了读的效率，因为LSM-Tree并不是原地更新数据，而是将数据顺序写入磁盘文件，在查找数据时，例如LevelDB中，需要从新文件一层一层查到旧文件，采用类似二分的方法，直到找到符合条件的数据。这样的过程涉及到多次磁盘I/O，尤其是在range query时，对读的影响特别明显。为了解决读性能差的问题，levelDB会在一定条件下对磁盘文件SSTable进行compact，将具有重合key的文件合并成新的文件。但这仍需要多次磁盘I/O读取每个SSTable文件的key范围，来确定querykey可能存在的SSTable文件。我们尝试在LSM-Tree结构上建立辅助结构，提高读的性能。

Learned Index是Kraska等人在2018年提出的一种结合机器学习技术的索引架构，并展示了相较于传统索引的适应性。已有的数据库索引技术，B-Tree，Hash等虽然已经发展的非常成熟，但他们都没有利用已有数据集的分布特性。而且随着数据集的越来越大，B-Tree等结构构建的索引占用空间越来越大，更新效率也会降低。现有的索引技术可以看做是Key映射到Value存储位置的函数，累积分布函数（CDF），表示为F(Key)->pos, pos表示Value在数据库的存储位置。学习索引技术就是通过学习数据的分布特征，建立一个可以替代F的机器学习模型, 一般来说模型会存在误差，这可以通过人为设定误差阈值锁定预测pos的范围，在这个范围内进行二分查找即可找到目标Value。使用学习索引技术可以通过模型以O(1)的时间复杂度快速找到Key所在的近似位置，同时占用的空间为常数大小，因为模型只需要保存学习后的参数。

学习索引不适用于动态数据，因为学习后建立的学习模型仅支持该数据集的查询，新插入的key可能会使学习模型的误差范围超过期望值。现有针对动态数据的学习索引技术，建立动态的学习模型，即当新key插入后，对模型进行切割或者合并来动态调整学习索引。LSM-Tree是一种针对写进行优化的数据存储结构，它的工作负载写入大于读取，从直观上看，它不适合建立学习索引，因为它不间断写入的数据会对学习模型进行干扰。但是，从另一方面来看，LSM-Tree的存储文件，如LevelDB中的SSTable，在一定时间内（与其他SSTable发生compact之前）会一直保持不变，所以我们得出结论，在固定时间内，在SSTable上建立的学习模型是有效的。另外，很重要的一点，一个SSTable文件内的数据是排序后的，我们可以利用排序后的数据分布特点，很轻松的建立学习模型。

学习模型

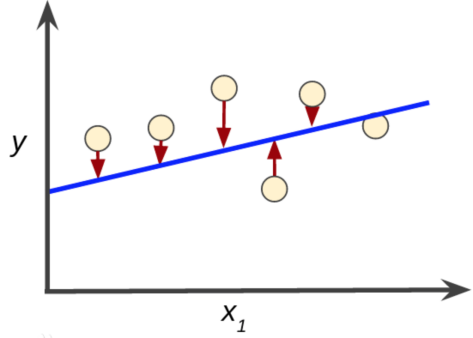
在理想条件下，任何非线性的物体都可以被拟合成为线性的物体。我们可以对世界上绝大多数的现象建立线性模型，并进行均值预测。学习索引就是要建立Key与Position之间的线性模型，通过对已有数据集的学习，预测query Key的近似Position。

线性模型的基本表示方法为：

给定数据集：R. 线性回归模型就是要找到一个函数F使得：

≈

在上式中 约等于 表示预测到的Pos在Key真实Pos的误差允许范围内，因为在大多数情况下，我们无法建立一个能准确预测到Pos的模型。如下图所示，我们不能找到一条完美经过所有点的线，但可以找到一条误差值可接受的线，它到数据集中任何一点的距离均在误差内。



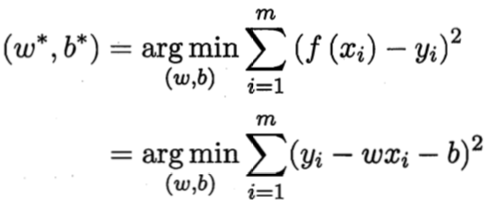
高斯提出了最小二乘法，求距离数据点最小的解向量来计算线性回归方程。该方法就是在一定条件约束下求解目标函数的极值。

损失函数是一个关于真实参数以及对该参数估计的函数：IMG_256，在概率论中，损失函数可以帮助我们衡量估计的好坏，当损失越小，估计的结果就越好；损失越大，估计的结果就越差。一种典型的估计方法就是平方误差损失函数，在线性回归模型中较为常见。

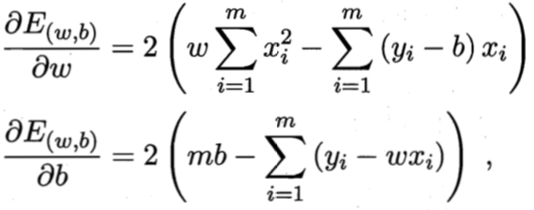
最小二乘法参数解求解方法

假设噪声符合高斯分布，求解线性回归模型参数的问题即为求解最小均方误差的问题。令偏导数为0求均方误差极值。

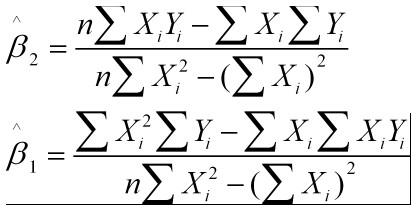
由≈得到



这里使用的均方误差在几何意义上表示了欧式距离，即数据集上的点到拟合曲线的距离。我们设立的方程为一元一次方程，对均方误差方程中w，b同时求偏导得到：



令上式=0就能得到w，b的值，同时均方误差表示的损失函数达到最小。



对于更高维的数据集，如何找到损失函数最优解有待学习和研究。

TODO: 交替最小二乘法 矩阵运算

算法描述

input: X, Y

output: w,b

double t1,t2,t3,t4

for i<-0 to X.size do:

t1 += X[i]\*X[i]

t2 += X[i]

t3 += X[i]\*Y[i]

t4 += Y[i]

w= (t3\*X.size() - t2\*t4) / (t1\*X.size() - t2\*t2)

b= (t1\*t4 - t2\*t3) / (t1\*X.size() - t2\*t2)

为了能组织起我们建立的模型，我们设计了模型结构。首先在模型中有以下几个成员变量，最大Key，最小Key，模型参数weights，

模型结构

class Model{

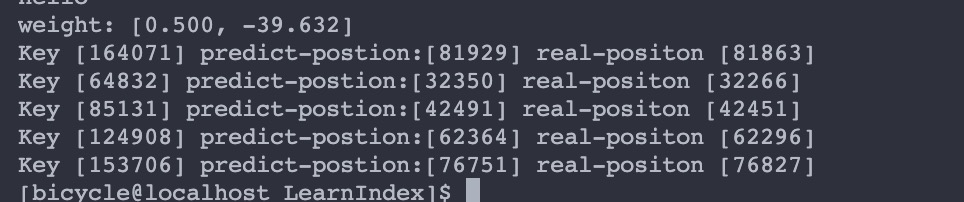
min\_key

max\_key

weights[]

}

随机生成N个数，对他们排序，保存到数组中，他们的位置表示pos就是数组Index，将数据输入最小二乘算法中求出w和b，即可建立学习模型。使用学习模型进行pos预测，得到实验结果：



从实验结果来看，预测效果在一定程度上是良好的。这是因为随机生成的key符合均匀分布，对他们进行排序后的数据分布能够很好地拟合成一条直线模型。但是，预测到的pos并不和真实pos一样，存在一定范围的误差，这是因为拟合的直线不是完全地经过数据集中每个点，而是一条到所有点欧式距离最短的曲线，这个欧氏距离最大值就是预测结果的最大误差。所以，我们可以通过设定一个误差阈值，来尽可能拟合出具有良好预测效果的线性模型。假设设定误差阈值e，通过上述的学习方法学习的模型为：，那么通过模型进行预测得到的pos。数据真实pos必然落在得到的区间内，我们可以在区间内进行二分查找，因为数据是有序的，直到找到该Key。总体来说，学习索引是对数据分布有规律的数据集有效的，当数据分布越无序，学习索引模型预测效果越差，所以我们应该设计一个可靠的方法，找到数据集的分布规律，使用对应的学习模型进行训练。

对于无序数据来说，一个常见的方法就是使用聚类算法将数据集划分为K个具有关联性的子集，针对子集进行学习。但是SST文件天然具有有序性，数据集的全体数据服从单调递增分布，我们设计的模型就是要利用其有序性建立一个学习索引。然而，拟合出一条整体数据集的单调递增曲线是非常困难的，所以我们设计出分段拟合方法，将整体数据集切割为M个区域，在每个区域建立线性回归模型进行学习。当区域长度足够小，拟合到的M条线段无限接近全部数据服从的单调递增曲线。但是，太小的区域会使M变得很大，增加了索引的查询效率和维护成本，因为M比较大时，我们要从M个线段中找到目标Key可能存在的线段，查询的时间复杂度至少为log(M), 查询效率与M呈负相关；当插入新的Key时，由于数据区域容纳的数据很小，会频繁创建新的线段并进行学习，造成恶性循环。

我们设计了一种递归分割算法，人为给定误差阈值E，对S区域内进行分割。首先选择首尾两个端点left和right，计算S区域内距离首尾两个端点所在直线最大的点作为第一个切割点，记该切割点为p1，到直线距离为d1，若d1小于E，说明线性模型可接受；若d1大于E，分别对p1左右两个区域进行递归操作，直到所有区域的线性模型可接受。

算法 split\_models为：

Input：待切割数据集X，数据集位置Y，误差阈值E, 数据集起始位置low, 数据集结束位置high

Output: 学习模型集合S

max\_distance=0;

for(x :low-> high){

distance = caculate\_distance(x, y, s.begin, s.end);

if(distance>max\_distance){

break\_point=(x,y);

max\_distance =distance;

}

}

if(max\_distance>E){

split\_models(X, Y, E, low, break\_point.x);

split\_models(X, Y, E, break\_point.x+1, high);

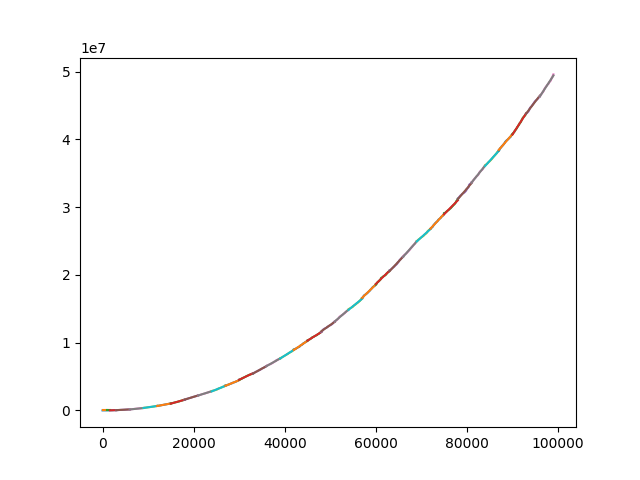
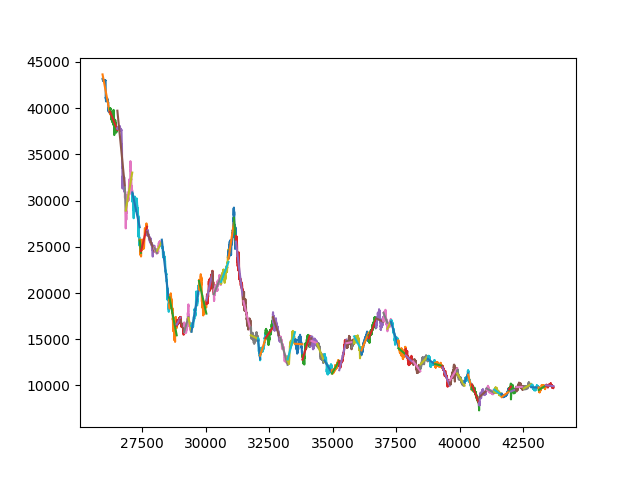
}else{

model = new LModel(X, Y, low, high);

S.append(model);

}

在一个数据集X上, 我们学习到了s个线性模型，并保存在模型集合S中。为了直观的显示我们的线性模型结构，下图表示上述算法的执行结果。

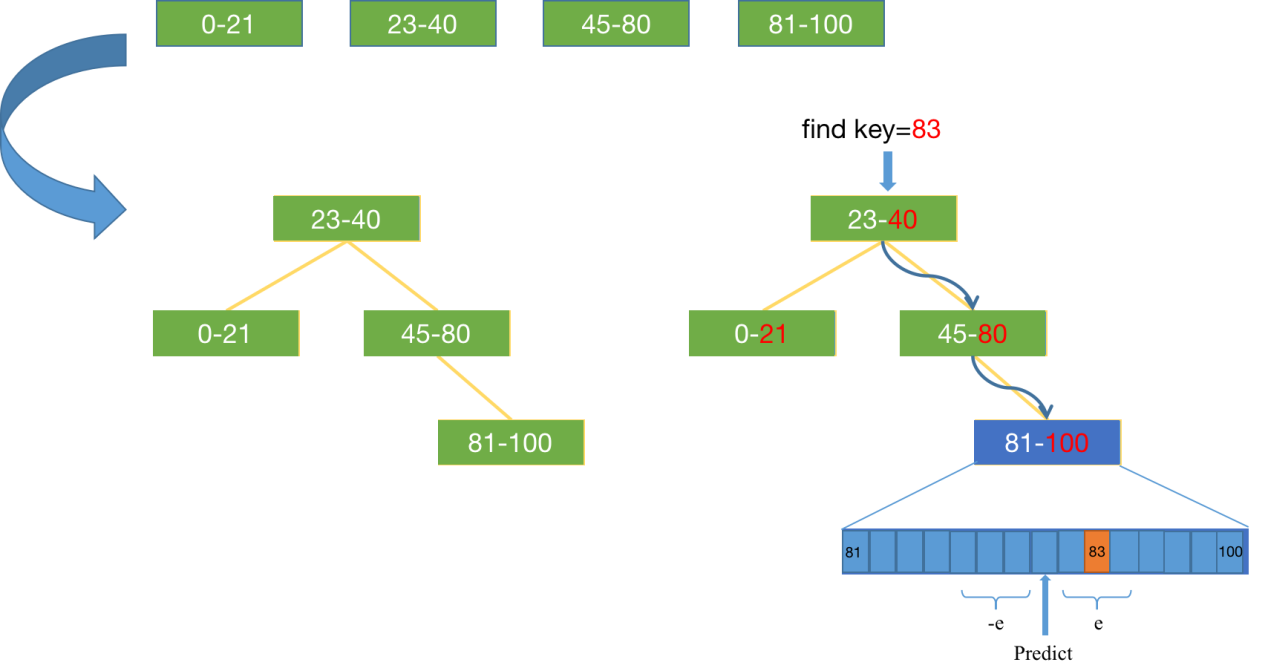


这是一个典型的具有一定分布规律的数据集，key是时间，value是数据存储位置，我们使用上述算法对该数据集进行切分，得到s个大小不同的数据区域，该结果在理论上保证是最优的，即保证在每个区域内能够建立可接受误差的线性模型。

对S区域切分得到的所有子集进行训练，得到s个线性模型lmodel。由于S区域是有序的，我们设计了Binseca容器存储所有训练得到的线性模型，该容器使用模型的min\_key作为排序关键字，按照从小到大的顺序插入子集训练得到的线性模型。同样的，我们在查找某个key时，使用二分查找的方式，找到Binseca容器中第一个min\_key不小于key的前一个模型作为目标模型，然后在使用该模型进行position预测，即调用模型的predict方法进行预测，预测到的结果是一个包含误差的区间[pos-e, pos+e]，e表示该模型的误差，在该区间内进行二分查找，直到找到target Key。

假设每个sst文件已经经过区域切分且建立好学习模型，同时现有的所有sst文件已经按照顺序排列。然后建立空的Binseca，对所有模型，按照模型中min\_key从小到大的规则插入到容器中。

例如：



我们针对一个有序的SSTable文件进行区域分割，然后使用线性回归模型学习分割后的每个区域，并生成了学习后的模型集合，最后创建一个Binsera容器，将集合内所有模型按照顺序组织起来。使用建立的该结构进行查询，时间复杂度跟模型数量M呈log(M)关系，空间复杂度为M

插入数据

在LSM-Tree中，写入数据并非原地更新，而是顺序写入磁盘文件log中，当内存中数据达到阈值后，才会将数据dump到SSTable中。所以频繁写入数据，并不会对我们建立的学习索引进行频繁更新。然而为了降低读放大的缺点，一定条件后会对SSTable文件进行merge。merge生成新的SStable文件，并删除旧的SSTable文件。如果在LSM-Tree中发生merge，我们就要更新索引，来支持新数据文件的查询功能。

当LSM-Tree发生merge时，我们对新生成的文件建立学习模型，并删除旧文件对应的学习模型

算法描述：

input: 进行merge的source\_sstables S，

output:

new\_sst = multi\_merge(S)

for each sst in S do

remove sst.mTree // 删除sst文件的模型

remove sst // 删除sst文件

mTree = new MTree()

models = split\_models(sst.data)// 新的sst文件进行学习

mTree.load(models) // 组织sst的学习模型到Mtree中

删除数据

删除数据并不是在数据库存储文件中擦除target Key，而是将删除命令顺序写入log，当SSTable文件进行合并时清理无效数据。在我们建立的学习索引中，删除数据操作跟插入Key一样，当merge触发后重新对新SSTable文件建立学习模型。

在levelDB中，数据存储结构按照level组织，每个level包含多个SSTable文件。level0比较特殊，因为level0的SSTable文件之间可能会有重复Key，而在其他更高的level中，文件之间不会出现重复Key。SSTable文件中数据是有序的，文件的IndexBlock区域存储着索引数据，IndexBlock保存一段Block的最大Key以及它的offset。查询target key时，现有的技术是遍历level中所有的sst文件，找出Key可能存在的一个，然后在IndexBlock区域通过二分查找找到targetKey可能存在的Block，具体就是读取offset位置的Block，最后在该Block中查询出对应的Value。

现有的做法是在MANIFEST文件，一个类似于记录所有sst文件的元信息的文件，元信息包括sst文件存储键的范围。但是查询的target\_key可能不在该文件中，但属于该文件表示的键范围，最坏情况下会遍历一层所有的sst文件，磁盘I/O次数跟文件个数T有关。现有的技术是在sst文件中增加一个额外的索引，例如布隆过滤器，来返回文件是否包含查询键。但是它只适合点查询，不适合范围查询，而且需要占用额外的磁盘空间。

我们设计了STree，按照sst文件中key的范围将level中所有文件组织起来，将避免无效的磁盘I/O。另外一方面，节省了将sst文件IndexBlock区域建立的时间消耗，一定程度上节省了磁盘空间。

建立好的STree，是按照sst文件内max\_key排序的，在STree中进行查询操作时，从根节点开始比较该sst文件的max\_key，若key大于max\_key, 继续向右子树查询；若key小于max\_key，则判断key是否在当前sst文件的范围内：是则调用该sst的MTree进行模型查询，否则从当点sst文件节点左子树执行操作。

search():

input: 查询key，STree根节点root

if root==null then

return null

endif

if key>root.max\_key then

search in root->rightNode

else

if key in range(root) then

return search in root.Mtree

else

search in root->leftNode

endif

更新STree

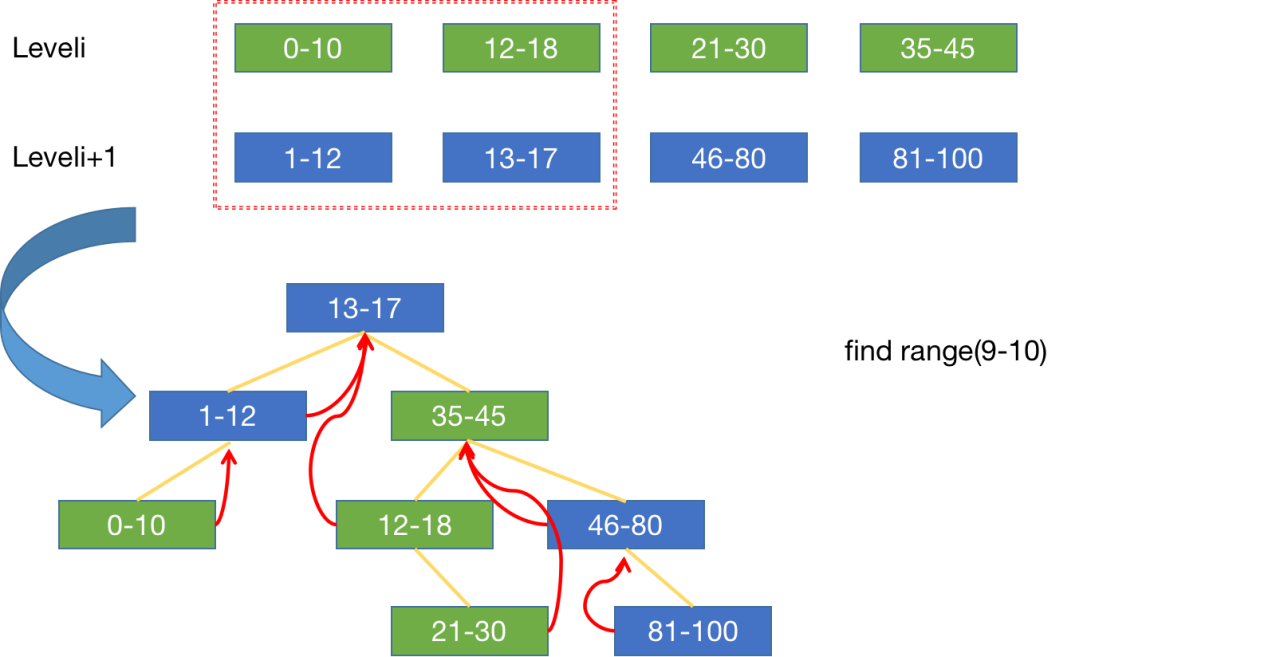
当sst文件发生merge后，会对新生成的sst文件建立学习模型Mtree，并删除旧的sst文件以及对应的Mtree。在我们设计的学习索引中，更新Stree要做的就是在merge过程中，将旧的sst文件在Stree中的节点删除，再将新的sst文件Mtree插入到Stree中。插入过程仍旧是按照STree的排序规则执行。

范围查询

范围查询和单点查询不同，除了在level0中可能存在重复片段，其他level不会出现重复片段，单点查询只需要在我们构建的Binseca容器中进行查找即可。然而对范围查询操作，由于各个level之间可能存在重复的key，而且可能存在查询区域分布在不同level。所以需要读取所有level中所有的SSTable文件，在他们中找到要查询的区间并进行组合（因为不同层次的不同文件创建时间不同，需要组合成一个同一的版本返回给用户）

我们设计了带有线索的树结构，提供高效的范围查询操作。与STree的组织结构不同的是，我们对系统中所有的sst文件按照max\_key大小进行排序建成一个排序树，树中每个节点除了包含key的范围之外，还有一个指向前驱节点的指针prev和指向后继节点的指针next。我们通过建立线索，保存了所有文件的相对顺序，当进行查询操作时，可以通过访问前驱指针或者后继指针来快速读取到查询范围内的数据。

例如：



范围查询时假设查询（p,q）, 首先在树中查找到max\_key不小于p的第一个节点s，若p处于该节点存储的key的范围之内，那么读取到p的value，并向后继续查找直到q，若在此过程中读取到后继节点，则读取后继节点的数据；若p小于min\_key, 则通过节点的前驱节点，找到min\_key不大于p的第一个节点，执行查询操作。因为查询树是按照max\_key排序的，且系统中所有的sst文件都会被组织到查询树，所以左子树可能存在和当前节点重复的区间。例如图中0-10和1-12都包含9-10的区间，这时我们应该继续向左子树查询，直到左子树根节点的max\_key大于p.

最近有效性

由于LSM-Tree的设计思想为，顺序写入数据，定期合并重复文件，所以层次之间会存在重复的Key。我们假设只查询最新的数据内容，那么在使用上面的查询树时，应舍弃掉时间更早的数据。在范围查询过程中，我们设计了合并堆，来确定最近有效的数据内容。

首先在查询树中找到所有可能出现在查询范围内的sst文件，这些文件之间会出现重复的key，而且暂时无法得知新旧信息。然后为每个待合并的sst文件创建一个selector，selector是一个指针，指向sst文件中第一个不小于p的Key，这可以通过模型树快速定位到。接着我们按照selector指向的Key排序顺序构建一个二叉树，从底向上出现的第一个非叶节点开始进行调整，直到构建成的树满足小顶堆定义，当出现重复的Key时，较新的selector数据内容被读取，重复key的新旧selector指针向后移动一位，此时堆的内容发生变化，从最后一个旧指针开始从底向上对堆进行调整。当selector指针指向的sst文件数据超出q或者读取结束，将其从堆中删除，调整堆。重复操作直到堆中所有数据输出。