DBTrain Lab 2 Report

刘泓尊 2018011446 计84

CI job ID: #112036

passed #112036 master 51a23164 #60120 by

基于B+Tree的索引结构

基于 `lab2-clean` 分支开发，实现了基于B+Tree的索引，支持节点分裂和合并、支持one-to-many的key-value插入。树节点页面在 `page/bptree_page/` 文件夹下。

1.节点页面设计

B+Tree 的阶数根据 `KeySize` 动态确定。节点上的 某个key值 是 其子树**最小key值**。

B+Tree的叶节点和内部节点使用同一个数据结构 `BPTreeNode` 维护，通过内部维护的 `NodeType` 区分是叶子还是内部节点。

对于内部节点，序列化的页面结构如下：

4096 Bytes

NodeType	KeyType	KeySize	CurSize	UNUSED	ParentPageID	Key Field	ChildPageID	BLANK
4	4	4	4	4	4	4 / 8	4	

Repeat `CurSize` Times

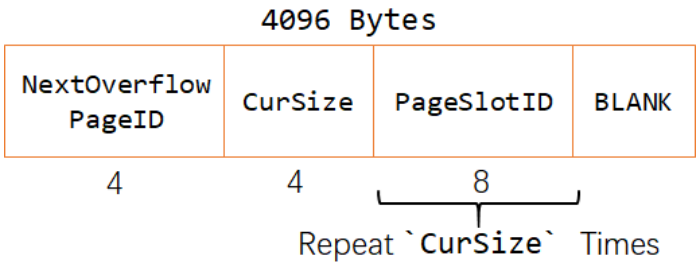
对于叶子节点，序列化的页面结构如下：

4096 Bytes

NodeType	KeyType	KeySize	CurSize	NextLeaf	ParentPageID	Key Field	PageSlotID	OverflowPage ID	BLANK
4	4	4	4	4	4	4 / 8	8	4	

Repeat `CurSize` Times

为了支持one-to-many的key-value pair, 叶子节点的每个key对应一个value页面链表，这些value页面成为**溢出节点**。每个叶子节点都保存了指向value链表的指针 `OverflowPageID` 。 溢出节点的接口在 `page/bptree_page/bptree_overflow_page.h` . 每个溢出节点页面都是链表上的一个节点。溢出节点页面的序列化如下：



对于`IntField`,每个内部节点的key数量为 508 个,每个叶子节点的key数量为 254 个,每个溢出节点能存储value 510个。

对于`FloatField`,每个内部节点的key数量为 339 个,每个叶子节点的key数量为 203 个,每个溢出节点能存储value 510个。

所以这种情况下，B+Tree仅2层就可以存储数据十万多个。极大提升了检索效率。

2.增删改查

Range Query：从根节点开始，对每个内部key向量进行二分查找`pLow`（借鉴了`lab2-simple`分支的`LowerBound`等二分查找函数），得到下一层节点的`PageID`.直到查到叶节点的某个key。从这个key对应的value开始，在叶节点链表遍历，直到遇到`key >= pHigh`停止。得益于B+Tree的叶节点链表，区间查询相比于BTree,可以节省很多次磁盘IO。

Update：同样从根查找到叶子，判断叶子是否存在`pKey`,不存在返回false,否则遍历key对应的value以及溢出节点链表，查找有没有对应的`value`,之后更新。在随机数据下，很多叶子节点的key对应的value只有1个，这时候也就没有创建溢出节点页面，所以遍历溢出节点链表的情况极少出现。

Insert（难点）:从根节点找到叶子节点，找到合适的插入位置。如果出现**上溢**，执行节点分裂。节点分裂实现难度比较大，我通过一个递归函数解决，传入当前节点指针，处理完本层分裂之后，再处理父节点的上溢情况，直到不需要再分裂。每个节点分裂都会将其 后`m/2`的key-value pair分给右兄弟，然后在父节点的向量处插入一个新索引，之后递归处理父节点可能存在的上溢。对于叶节点，需要注意**维护叶节点链表**。

Delete（难点）:同样从根查找到叶子，判断叶子是否存在`pKey`,不存在返回false。如果要删除特定值， 可能还要遍历溢出节点页面，判断value是否存在，同时注意维护溢出页面为空时回收页面。如果key对应的所有value都被删除了，则删除该key。如果出现**下溢**，执行节点合并或向左右兄弟借用。处理下溢是最难的点了，同样通过递归函数解决，处理完本层再处理上一层。下溢要处理的情况很多，我概括为以下5点：

- 本节点没有父节点，说明为根，不需要满足`> m/2`的规定，只有当`key`数量为1时，才删除该根，将root的唯一孩子作为新的根。返回。
- 本节点存在左邻居，如果左邻居key数量 > `m/2`(可借)， 则从左兄弟借一对，插入本节点头部，然后更新父节点的`key`为借来的`key`,因为这个`key`一定更小。返回。
- 本节点不存在左邻居，则一定存在右邻居。类似的，如果右兄弟可借，则向右兄弟借第一个值，同时把右兄弟的父节点的`key`更新，因为这个`key`一定会变大。返回。
- 本节点左右邻居都不够借。执行节点合并：如果存在左邻居，则和左邻居合并，删除父节点中针对本节点的key-value pair，递归处理父节点可能的下溢。对于叶节点，需要注意维护叶节点链表。
- 如果不存在左邻居，则一定存在右邻居，和右邻居合并，删除右邻居的父节点中 针对右邻居的key-value pair,递归处理父节点可能的下溢。对于叶节点，需要注意维护叶节点链表。

处理上溢/下溢的过程比较复杂，复杂度$O(logn)$但是常数比较大，尤其是配合flush到磁盘的序列化操作。所幸这种情况在随机数据下很少出现，也很少要递归处理到根节点，所以B+Tree能保证高效的插入和删除。

3.对框架的一处小修改

在执行 `Clear` 操作的时候，框架在 `IndexManager::DropIndex` 中，调用 `Clear` 之后又删除了根节点，但我的实现会在调用 `Index::Clear` 的时候也删除根节点。所以我把 `DropIndex` 的重复删除去掉了：

```
Index *pIndex = GetIndex(sTableName, sColName);
pIndex->Clear();
delete pIndex;
PageID nRoot = _iIndexIDMap[sIndexName];
assert(!MiniOS::GetOS()->Used(nRoot)); // add by me
// MiniOS::GetOS()->DeletePage(nRoot); // deleted by me
_iIndexIDMap.erase(sIndexName);
_iIndexMap.erase(sIndexName);
```

即便我不删除根节点，框架对根节点 `nRoot` 的维护也有一点问题。因为框架获得 `nRoot` 是在 `IndexManager::AddIndex` 处，但是此时直到 `IndexManager::DropIndex` 过程中， `nRootID` 是会随着树高的变化而动态变化的，但是 `IndexManager` 里维护的 `nRoot` 没有变，所以可能会出现重复删除的情况。所以为了能适配我的实现，我就把这里回收根节点的操作去掉了。

4.对Index的更强测试

`RandomSqlTest` 只有1000次插入和1000次查询，数据量比较小，而且也只有插入和查询，不能全面测试B+Tree的性能与正确性。我在 `IndexTest` 模块添加了一些操作，用来测试增删改查的正确性. 插入了10w条数据，之后更新5k条，删除7k条，以及若干查询操作， 总共12w左右次操作。同时也加入了对one-to-many的key-value pair的插入和查询，测试溢出节点的实现正确性。树高约为2。

```
TEST(Lab2, IndexTest) {
  Index* pIndex = new Index(FieldType::INT_TYPE);
  // test insert 100000
  for (int i = 2; i < 100000; ++i) {
    pIndex->Insert(new IntField(i), {0, i});
    if (i % 1000 == 0) pIndex->Insert(new IntField(i), {i, 0});
  }
  // test search
  Search(pIndex, 0, 5435);
  // test update 5000
  for (int i = 2000; i < 7000; ++i) {
    EXPECT_EQ(pIndex->Update(new IntField(i), {0, i}, {i, i}), true);
    if (i % 1000 == 0) EXPECT_EQ(pIndex->Update(new IntField(i), {i, 0}, {i, i}),
true);
  }
  // test delete 5000
  for (int i = 2000; i < 7000; ++i) {
    EXPECT_EQ(pIndex->Delete(new IntField(i), {i, i}), true);
    if (i % 1000 == 0) EXPECT_EQ(pIndex->Delete(new IntField(i), {i, 0}), false);
  }
  // test delete all 2000
  for (int i = 8000; i < 10000; ++i) {
    if (i % 1000) EXPECT_EQ(pIndex->Delete(new IntField(i)), 1);
    else EXPECT_EQ(pIndex->Delete(new IntField(i)), 2);
```

```
    }
    // test search
    Search(pIndex, 0, 10000);
    // test clear
    pIndex->Clear();
    delete pIndex;
}
```

下面是测试运行结果。可以看到对这12w次操作，执行耗时`6110ms`，**10w级数据平均每个操作执行时间约`0.05ms`**，性能是很高的。 对于`RandomSqlTest`，1000次插入+1000次查询操作和lab1的1000次插入+10次查询操作耗时接近。

```
Database Init.
Build Finish.
[==========] Running 7 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 7 tests from Lab2
[ RUN      ] Lab2.IndexTest
Range[0,5435): 5438 results
Range[0,10000): 3005 results
[       OK ] Lab2.IndexTest (6110 ms)
[ RUN      ] Lab2.RandomSqlTest
[       OK ] Lab2.RandomSqlTest (176 ms)
[ RUN      ] Lab2.InsertSelectTest
[       OK ] Lab2.InsertSelectTest (1 ms)
[ RUN      ] Lab2.UpdateTest
[       OK ] Lab2.UpdateTest (0 ms)
[ RUN      ] Lab2.DeleteTest
[       OK ] Lab2.DeleteTest (0 ms)
[ RUN      ] Lab2.UpdateDeleteTest
[       OK ] Lab2.UpdateDeleteTest (1 ms)
[ RUN      ] Lab2.DeleteInsertTest
[       OK ] Lab2.DeleteInsertTest (0 ms)
[----------] 7 tests from Lab2 (6288 ms total)

[----------] Global test environment tear-down
[==========] 7 tests from 1 test suite ran. (6288 ms total)
[  PASSED  ] 7 tests.
```