简易key-value型

Database 设计文档

515030910029
俞昕宜

2016.07

目录

一、基本信息

- 项目名称：简易key-value型数据库
- 主要数据结构：B+树
- 开发环境：Xcode Version 7.2
- 测试环境：MacBook

  处理器：1.2 GHz Intel Core M

  内存：8 GB 1600 MHz DDR3
- 概述：实现<key,value>型数据的增添、删除、修改、查找功能

二、提供接口

提供 `<int,int>` 与 `<string,string>`两种类型<key,value>存储，其中若key类型为`string`，则其最大长度可由用户设定；若value类型为`string`，则其为不定长。

- 创建

  ```
  DataBase(const string& pathname);
  ```

  - pathname：文件路径名
  - 创建一个数据库对象
- 打开

  ```
  void open();
  ```

  - 创建索引与数据文件（若原先不存在）、读取索引文件建立B+树、创建读与写缓存、创建空间重利用容器
- 关闭

  ```
  void close();
  ```

  - 强制刷新数据文件写缓存
- 存储

  ```
  bool store(const T& key, const V& value);
  void storeUsr(const T& key, const V& value);
  ```

  - 提供两种接口：测试用接口与用户用接口，用户用接口会直接输出提示信息如“STORE SUCCESS!”或提示数据库中已有该词条等
  - key：待存储的关键词；value：待存储的值
  - 存储该<key,value>词条
  - 不支持重复存储，即若数据库中已保存此key相关数据，此次存储操作将不对数据库进行任何修改（若使用用户接口，则会输出提示信息）
- 删除

```
bool remove(const T& key);
void removeUsr(const T& key);
```

- 提供两种接口：测试用接口与用户用接口，用户用接口会直接输出提示信息如“REMOVE SUCCESS!”或提示数据库中无该词条等
- key：待删除的关键词
- 删除该key存储的value
- 若原数据库中不存在该关键词数据，此次删除操作将不对数据库进行任何修改（若使用用户接口，则会输出提示信息）

- **修改**

```
bool modify(const T& key, const V& value);
void modifyUsr(const T& key, const V& value);
```

- 提供两种接口：测试用接口与用户用接口，用户用接口会直接输出提示信息如“MODIFY SUCCESS!”或提示数据库中无该词条等
- key：待修改的关键词；value：该关键词对应的新值
- 若原数据中不存在该关键词数据，此次修改操作将不对数据库进行任何修改（若使用用户接口，则会输出提示信息）

- **查找**

```
std::shared_ptr<V> fetch(const T& key);
bool fetchTst(const T& key, const V& correctValue);
void fetchUsr(const T& key);
```

- 提供三种接口：原始接口、测试用接口与用户用接口。测试接口封装了与正确值的对比，返回一个`bool`类型；用户用接口会直接输出<key,value>或提示数据库中无该词条等
- key：待查找的关键词
- 查找该关键词的值

三、模块实现

- **interface_test_version.cpp**

接口实现，提供了上述接口

- **btnode.cpp**

B+树的节点类，对节点包含的信息进行封装

- **btree.cpp**

数据库的主体部分，B+树索引文件类，实现对索引文件的操作和B+树的操作封装（对B+树的操作即对索引文件进行相应操作）

- **data.cpp**

数据文件类，实现对数据文件操作的封装

- **cache.cpp**

数据读缓存类，存储最近使用到的<key,value>对，实现对该缓存的管理封装

- **buffer.cpp**

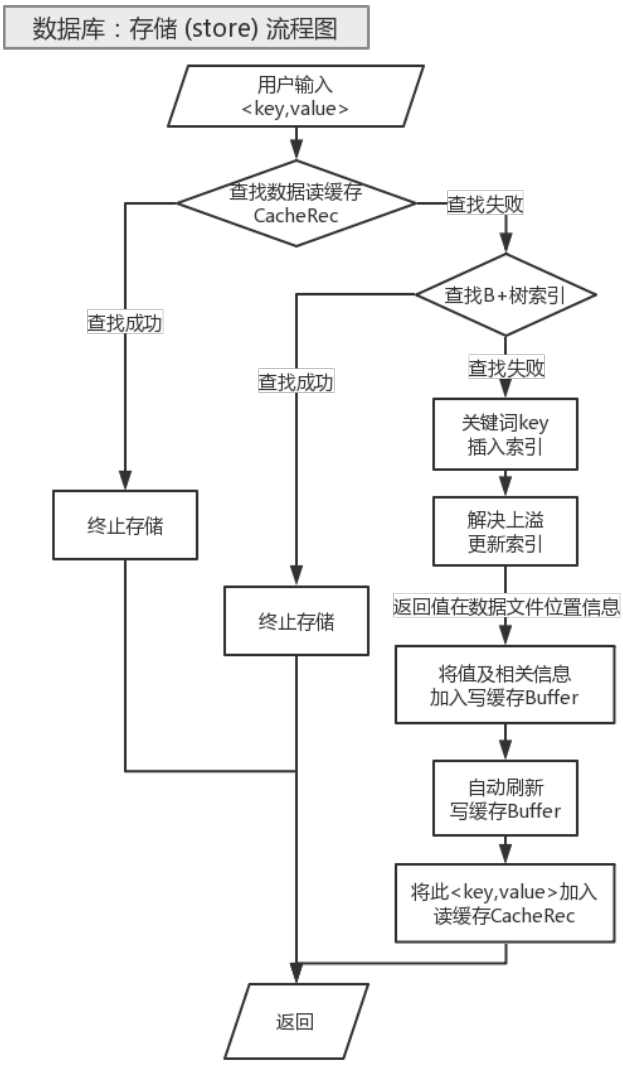
数据写缓存类，存储一定数量对数据文件的操作，达到某一设定值后一并对数据文件操作，实现对该缓存的管理封装

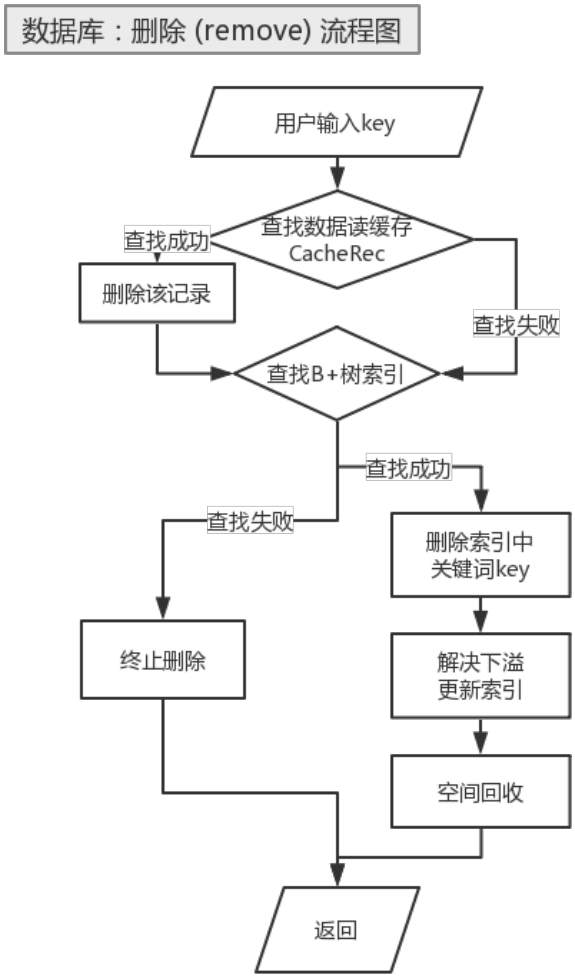
- **freebin.h & freebin.cpp**

空间重利用类，存储可重新利用的索引及数据文件空间信息，实现对该结构的管理封装

四、功能实现及流程图（均指原始接口）

- 打开
  - 尝试打开索引与数据文件，若本不存在，则创建文件
  - 读取索引文件，获得相关信息（如根节点、索引与数据文件长度等），建立B+树
  - 初始化数据读写缓存
- 关闭
  - 强制刷新数据写缓存，关闭文件



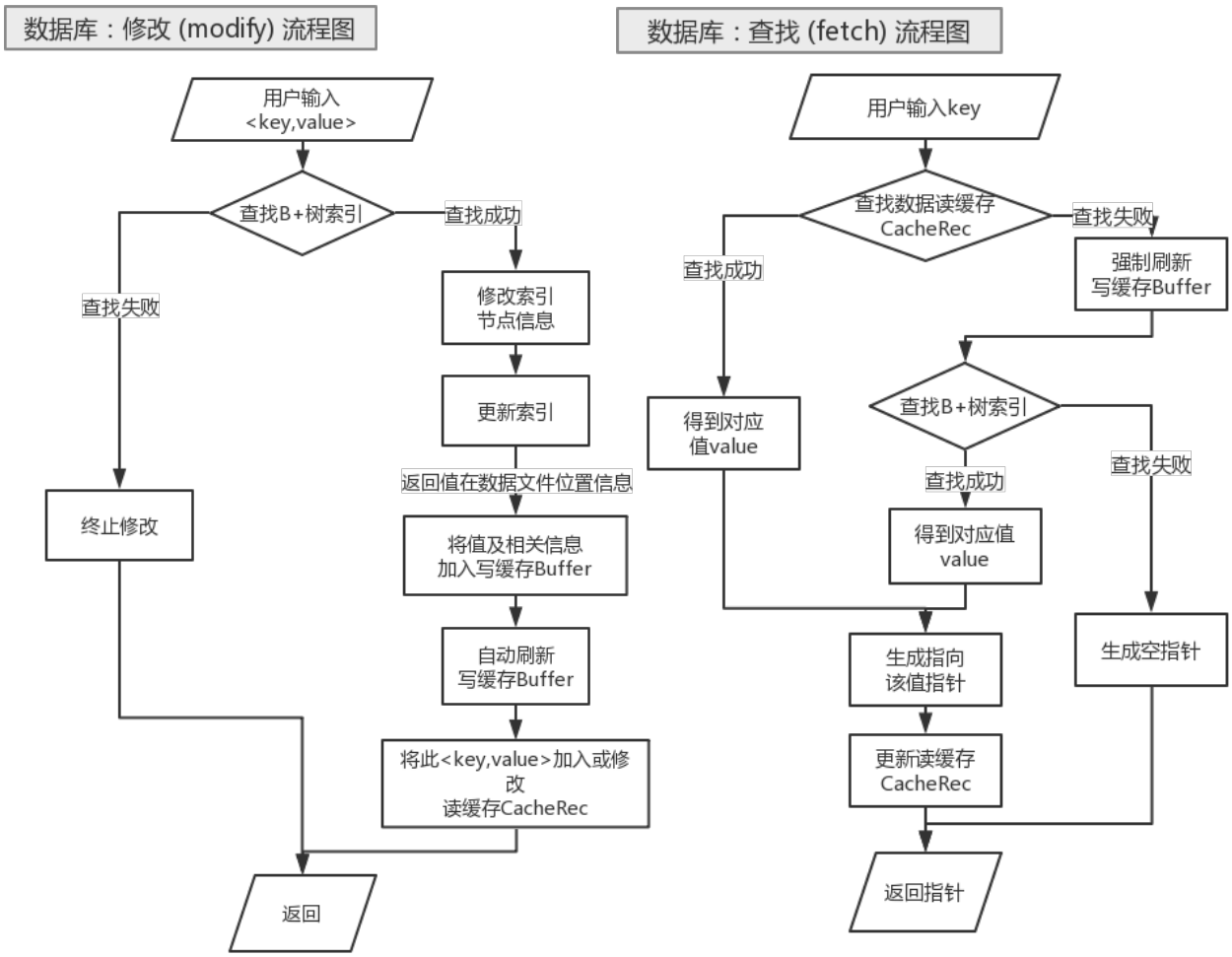


- 存储
  - 查找数据读缓存CacheRec，若存在该关键词，终止存储，直接返回
  - 否则，查找B+树，若存在该关键词，终止存储，直接返回

- 否则，利用查找结果插入该关键词，解决B+树上溢等问题后将修改过的节点写入索引文件
- 将该关键词对应值及其在数据文件中所在位置、长度等信息（重利用空间）放入数据写缓存`Buffer`，自动刷新`Buffer`
- 将此<key,value>对加入读缓存`CacheRec`

- **删除**

  - 查找数据读缓存`CacheRec`，若存在该关键词，删除
  - 查找B+树，若不存在该关键词，终止删除，直接返回
  - 删除该关键词，空间回收



- **修改**

  - 查找B+树，若不存在该关键词，终止修改，直接返回

- 否则，利用查找结果修改索引文件中的该关键词相关信息
- 将该关键词对应值在数据文件中所在位置、长度（重利用空间）等信息放入数据写缓存Buffer，自动刷新Buffer
- 在数据读缓存CacheRec中修改或添加此<key,value>对

- **查找**

- 查找数据读缓存CacheRec，若存在该关键词，返回其对应的值
- 否则，强制刷新数据写缓存Buffer
- 查找B+树，若存在该关键词，得到其在数据文件中的值，返回指向该值的指针
- 将此<key,value>对加入数据读缓存CacheRec
- 否则，返回空指针

五、特点

1. 不定长value

在该数据设计的两种value类型中，string类型可以不定长度

实现方法：在索引文件中保存value的长度

2. 空间重利用DataBin与NodeBin类

- **主要数据结构**：

  ```
  list<DataBinToken> binList;
  list<int> stPosList;
  ```

  其中DataBinToken是自行设计的结构体，保存数据文件可利用空间的位置与长度
- **实现功能**：
  - DataBin管理数据文件中可用空间信息
  - NodeBin管理索引文件中可用空间信息，因本数据库设计中一个节点在索引文件中的长度固定，因此该结构只需保存可用空间起始位置
- **被调用环境及作用**：
  - 在存储(store)、修改(modify)关键词时，获取值在数据文件的位置时，可先考察DataBin中是否有可重利用空间
  - 在删除(remove)关键词时，可将对应值在数据文件中的空间回收，放入DataBin结构
  - 在解决上溢问题时，获取新节点在索引文件的位置时，可先考察NodeBin中是否有可用空间
  - 在解决下溢问题时，可回收废弃节点在索引文件中的空间，放入NodeBin结构
  - 此结构很好的提高了空间利用率，见性能分析

3. 缓存设计

3.1 数据读缓存CacheRec类

- **主要数据结构**:

  `vector<pair<key,value>> memory;`

- **实现功能**:
  - 利用数据使用的局部性(locality)，将近期使用过的<key,value>保存，且始终保证最近使用的<key,value>位于memory的第一位
  - 维护缓存容量，一旦缓存超过容量，删除最后一条<key,value>对，即删除了最久没有使用的数据
- **被调用环境及作用**:
  - 在进行关键词的查找时，若该<key,value>刚刚被访问过，可直接读取该缓存返回对应值，免去对B+树的查找，提高效率
  - 在进行关键词的插入时，若该<key,value>存在于该缓存中，可提早终止插入操作（不支持相同key不同value的插入），免去对B+树查找

3.2 数据写缓存Buffer类（一致性刷新）

- **主要数据结构**:

  `queue<BufferToken> buffer;`

  其中`BufferToken`是自行设计的结构体，存储了对数据文件修改操作的信息（如数据值、位置等）
- **实现功能**:
  - 为避免每次增添修改数据后即打开数据文件进行操作的时间消耗，将多次操作集中在一起一并进行
  - 提供自动(`AUTO`)与强制(`COERCIVE`)两种刷新模式：前者是在写缓存容量满后自动刷新，打开数据文件进行操作；后者是强制刷新，将目前存在于缓存的操作全部完成。

    后者主要保证数据库的正确性，试想：在插入<key,value>后，仅仅将其放入写缓存

自动刷新，若在未刷新时用户需要查找此关键词，而其值还未存入数据文件，就会发生错误。因此在每次查找(fetch)操作前数据库都会执行强制刷新。

- **被调用环境及作用**：
  - 在存储(store)或修改(modify)数据时，都会将相关操作放入数据写缓存，避免多次打开文件的时间耗费
- **参数分析**：

  缓存大小：见性能测试分析

3.3 索引文件写缓存（一致性刷新）

- **主要数据结构**：

  ```
  vector<shared_ptr<BTNode>> writeBuffer;
  ```
- **实现功能**：

  在插入或删除关键词时，不仅是对叶节点的修改，也会由于下溢或上溢涉及到非叶节点。该索引文件写缓存保存了一次插入或删除操作涉及到的需要修改的节点指针，在操作完成后一并对索引文件进行修改
- **被调用环境及作用**：

  在存储(store)或删除(remove)关键词时，将所有修改过的节点指针放入该缓存，一次存储或删除操作结束后，进行修改，避免多次写入

六、测试及分析

1. 正确性测试

1.1 大量数据测试(`correctness_test.cpp`):

- 建立数据库，分别执行下列操作:
  - 随机存储TIMES条数据
  - 随机删除部分数据
  - 读取全部数据（包括删除后的数据），比对是否与插入相同
  - 随机修改数据
  - 读取全部数据，进行比对
  - 随机存储TIMES条数据
  - 读取全部数据，进行比对
- 该测试主要涉及大数据量下数据库的正确性
- 在TIMES分别取100、1000、10000、100000、1000000时，三次读取操作，全部正确

1.2 非寻常测试(`abnormal.cpp`):

- 建立数据库，分别执行下列操作:
  - 随机存储10000条数据
  - 循环以下步骤多次:
    - 存储一特定关键字及值<key,value>，读取该关键字
    - 删除该关键字，读取该关键字
    - 删除该关键字，读取该关键字
    - 修改该关键字，读取该关键字
    - 存储该关键字及一个不同的值，读取该关键字
- 该测试涉及读取、删除、修改不存在于数据库中的可能造成异常的操作
- 每次循环中，都按照"原value-空指针-空指针-空指针-新value"的顺序返回读取内容，表明测试正确

1.3 综合测试(round_test.cpp):

- 建立数据库，执行老师提供ppt上部分操作，先插入nrec数据，后循环以下步骤多次：
  - 随机读一条记录。
  - 每循环37次，随机删除一条记录。
  - 每循环11次，随机添加一条记录并读取这条记录。
  - 每循环17次，随机替换一条记录为新记录。
- 该测试涉及综合存储、删除、修改、读取操作
- 在每次循环读取中，都得到与插入时相同的值，或者该关键字被删除且读取返回空指针，表明测试正确。

2. 性能测试

为了测试速度，以下测试结果基本来源于<int,int>型数据库

2.1 空间性能

空间重利用对文件大小影响(space_test.cpp)

空间利用 20万数据量文件大小比对（单位：M）

	无空间重利用	空间重利用	差占重利用百分比
数据文件	1.6	1.5	6.3%
索引文件	7.0	6.4	8.6%

- 本次测试使用<int,int>型数据库，节点大小为128，数据量为20万，执行操作为：插入20万数据，删除7万数据，再插入20万数据
- 结果分析：
  - 不使用空间重利用后，再插入的数据只可增加到数据文件末尾。一个int大小为4byte，可估算数据文件大小为：

    (200000 + 200000) * 4 / 1024 / 1024 = 1.5 M
  - 而使用空间重利用后，再插入的数据可使用之前的空间。可估算数据文件大小为：

(200000 - 70000 + 200000) * 4 / 1024 / 1024 = 1.3 M

可近似认为符合预期。

- 在节点大小越大、删除数据量越多、每个数据长度越长的情况下，空间重利用对文件大小影响越大，这不难理解。

2.2 时间性能

2.2.1 读缓存(模仿用户行为)对操作时间影响(read_buffer_test.cpp)

读缓存 时间对比/s

	无读缓存	读缓存	差占读缓存百分比
1000次操作	0.906	0.813	10.26%
10000次操作	10.186	9.319	8.51%
100000次操作	143.547	124.534	13.25%

- 本次测试操作模仿用户操作，执行以下循环N次：

  随机插入20条记录；随机查找刚插入的这条记录；有时（随机）修改这条记录；有时（随机）删除这条记录

  该操作模仿了用户插入数据后，查看某条数据是否插入正确，进行修改或删除的操作

- 结果分析：
  - 不使用读缓存时，在查找、删除、修改记录时，都需要重新搜索树，到数据文件中抓取相应信息，对照节点大小为32的单次操作时间，可估计需要时间大约为：

1000次循环： $1000 \times (35 \times 20 + 101 + 39 \times 0.5 + 19 \times 0.5) \times 10^{-6} = 0.83\ s$

10000次循环： $10000 \times (58 \times 20 + 96 + 65 \times 0.5 + 36 \times 0.5) \times 10^{-6} = 10.405\ s$

100000次循环： $100000 \times (58 \times 20 + 96 + 65 \times 0.5 + 36 \times 0.5) \times 10^{-6} = 132\ s$

  - 使用读缓存时（容量为20），查找时间可忽略不计，则可估计需要时间大约为：

1000次循环： $1000 \times (35 \times 20 + 39 \times 0.5 + 19 \times 0.5) \times 10^{-6} = 0.73\ s$

10000次循环： $10000 \times (46 \times 20 + 43 \times 0.5 + 26 \times 0.5) \times 10^{-6} = 9.55\ s$

100000次循环： $100000 \times (58 \times 20 + 65 \times 0.5 + 36 \times 0.5) \times 10^{-6} = 121.1\ s$

可认为近似符合预期。

- 在插入后不久，进行查找时，使用读缓存，效率明显提升。实现该缓存主要也是从用户使用习惯上考虑，该缓存很适应用户的习惯操作，因此具有普遍意义。

2.2.2 写缓存及写缓存大小对操作时间影响

写缓存 插入时间对比/s

数据量/万	无写缓存	写缓存（容量为4）	写缓存（容量为8）	写缓存（容量为16）	写缓存（容量为32）	最大差占百分比
0.1	0.092	0.068	0.037	0.053	0.059	59.78%
1	1.015	0.806	0.496	0.612	0.500	51.13%
10	10.618	8.032	6.012	5.865	5.902	44.76%

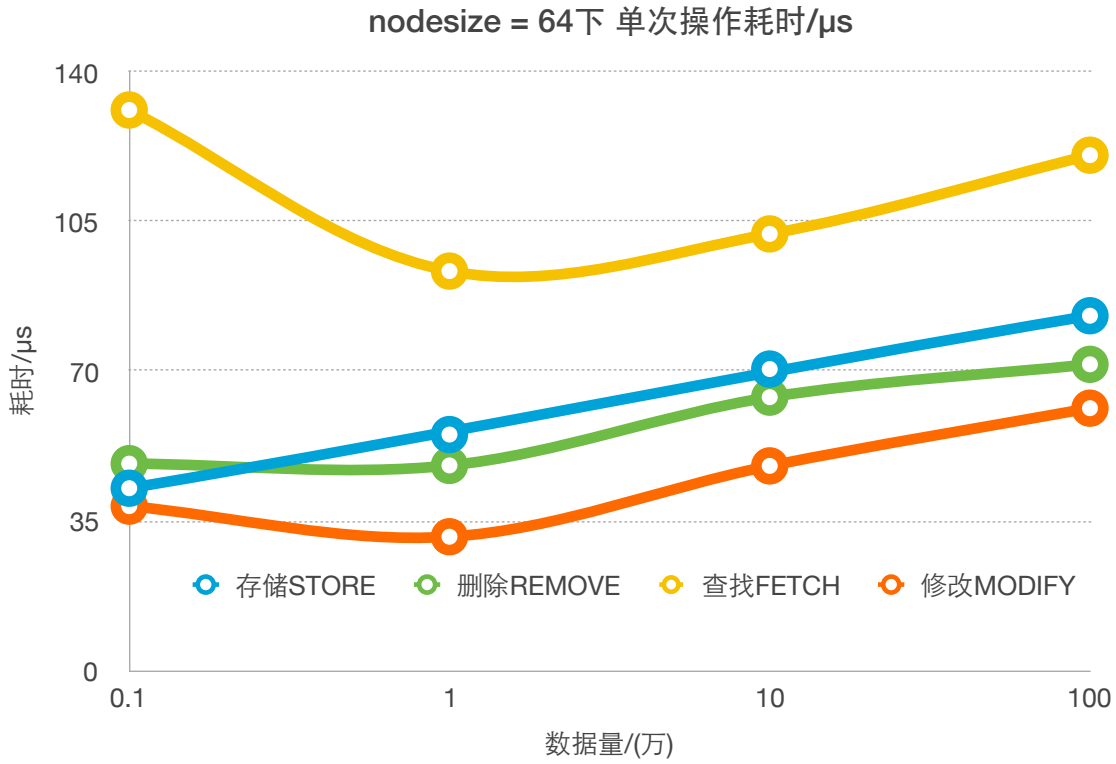
- 本次测试比较有无缓存及缓存容量对插入操作时间的影响。测试操作为：随机插入一定量数据，比较插入时间
- 结果分析：
  - 写缓存主要是在插入及修改时，操作数量达到一定值后一并对数据文件进行操作。很清晰的看到，在测试数据中，有写缓存的插入比无写缓存的插入快不少，最快效率可以提高50%
  - 观察图标可知，写缓存容量大约为8时速度最快。由于是单纯的`int`操作，写入的数据都是连续的，可以认为一次连续写入 4 x 8 = 32 `byte`时速度最快，我暂时还不理解写入速度与其他什么因素有关，只能控制这一个变量，因此该结论有待验证

2.2.3 操作时间与数据量关系(correctness_test.cpp)

- 节点大小固定的情况下，数据量对各个操作单次耗时影响：

nodesize = 64下 单次操作耗时/μs

数据量/(万)	0.1	1	10	100
存储STORE	42.603	55.154	70.484	82.957
删除REMOVE	48.353	47.933	63.903	71.586
查找FETCH	131.09	93.4	102.1	120.523
修改MODIFY	38.4	31.25	47.826	61.325

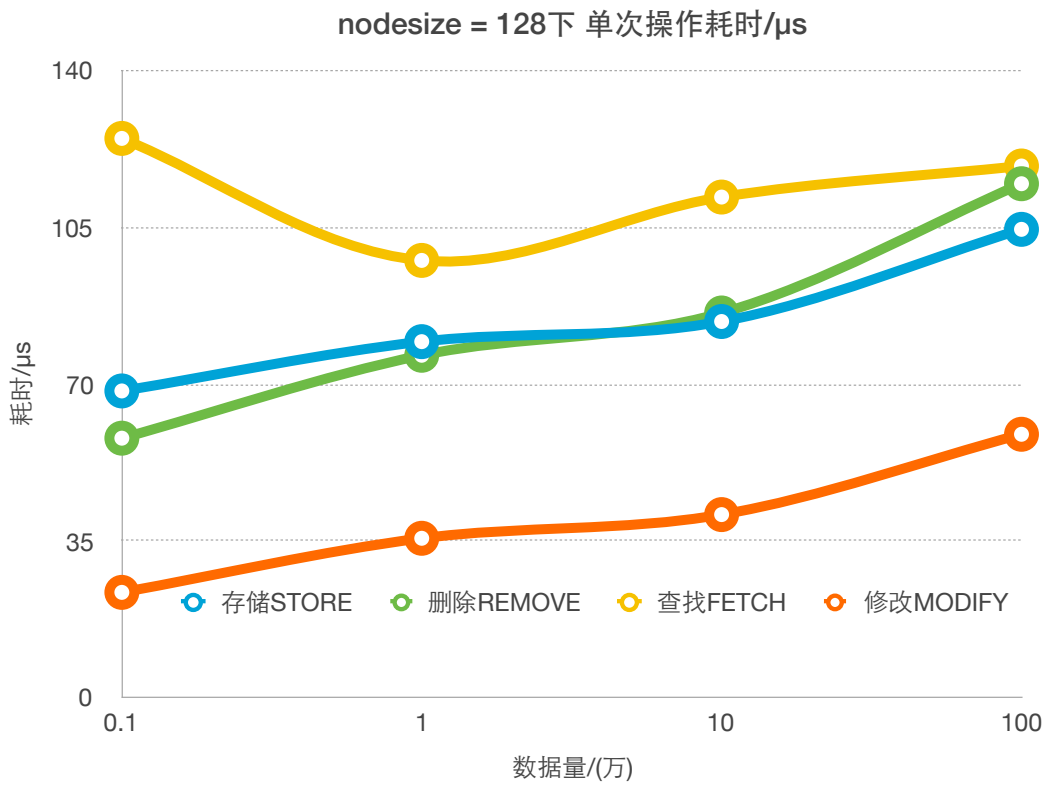


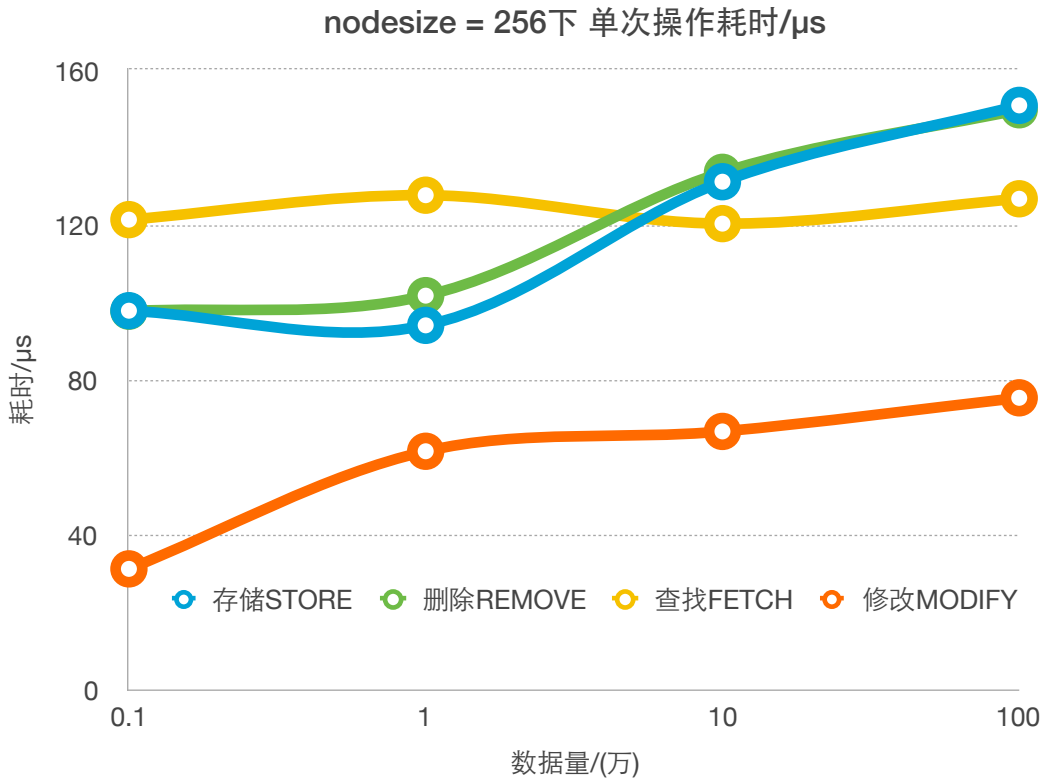
nodesize = 128下 单次操作耗时/μs

数据量/(万)	0.1	1	10	100
存储STORE	68.471	79.477	83.973	104.55
删除REMOVE	57.924	76.437	85.878	114.736
查找FETCH	124.854	97.615	111.759	118.695
修改MODIFY	23.5	35.574	40.864	58.796

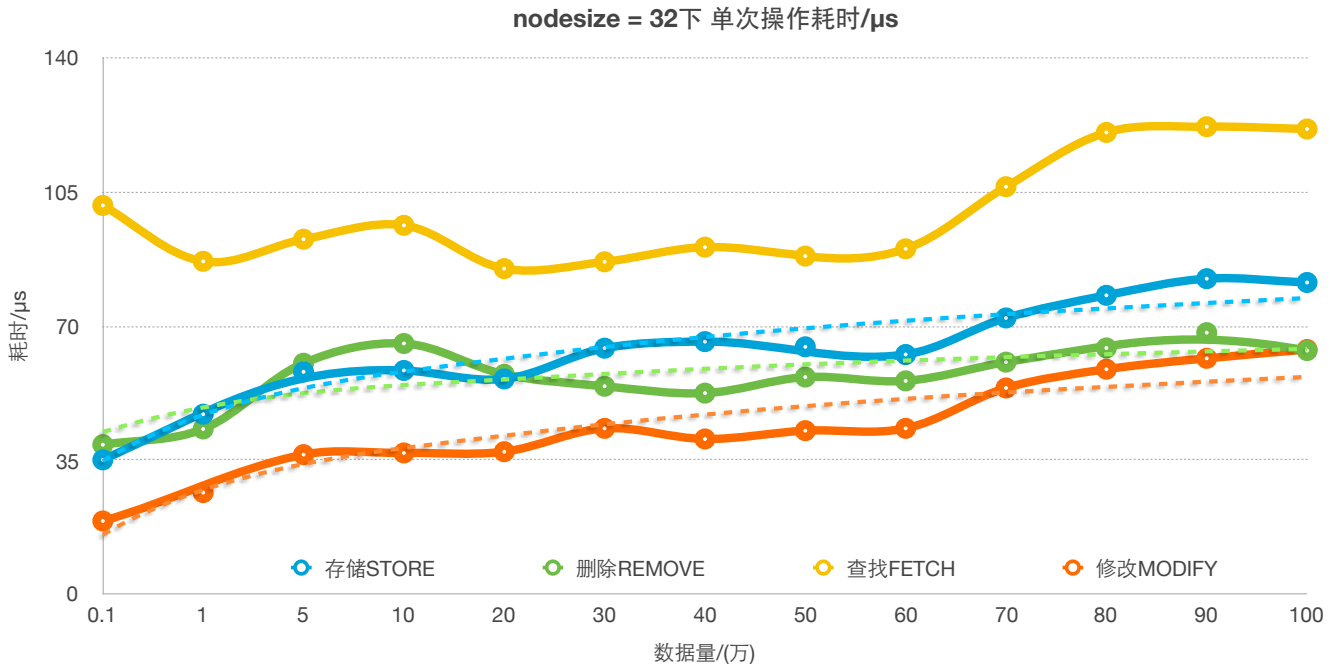
nodesize = 256下 单次操作耗时/μs

数据量/(万)	0.1	1	10	100
存储STORE	97.823	94.045	131.143	150.861
删除REMOVE	97.694	101.715	133.57	149.725
查找FETCH	121.349	127.659	120.316	126.738
修改MODIFY	31.14	61.519	66.643	75.326





- 可见，此时各个操作耗时对比与趋势基本相同，因此取出nodesize = 32进行具体分析：



nodesize = 32下 单次操作耗时/μs

数据量/(万)	0.1	1	5	10	20	30	40	50	60	70	80	90	100
存储STORE	35.073	46.997	58.07	58.421	56.21	64.24	65.954	64.6	62.6	72.14	78.05	82.4	81.4
删除REMOVE	39.025	43.141	60.306	65.469	57.413	54.3	52.5	56.7	55.7	60.6	64.3	68.3	63.621
查找FETCH	101.536	86.933	92.7	96.334	85.02	86.8	90.635	88.3	90.2	106.42	120.6	122.1	121.5
修改MODIFY	19.108	26.524	36.446	36.879	37.2	43.3	40.525	42.7	43.3	53.85	58.75	61.6	63.9

- **存储STORE与删除REMOVE操作：**
- 需要先搜索树，内存中比较计算若忽略不计，则需要读取树高h个节点；在内存中插入后，查找时经过的沿路节点（在内存中）可能发生上溢或下溢，需要更新这些节点，此时主要操作实际是在内存中进行，随后把树高h个节点加入写缓存一并对索引文件操作；对数据文件进行操作。所以存储删除操作消耗时间在节点大小固定时，大致正比于树高。由于树高基本随数据量成对数增长，因此存储及删除操作应随数据量成对数增长

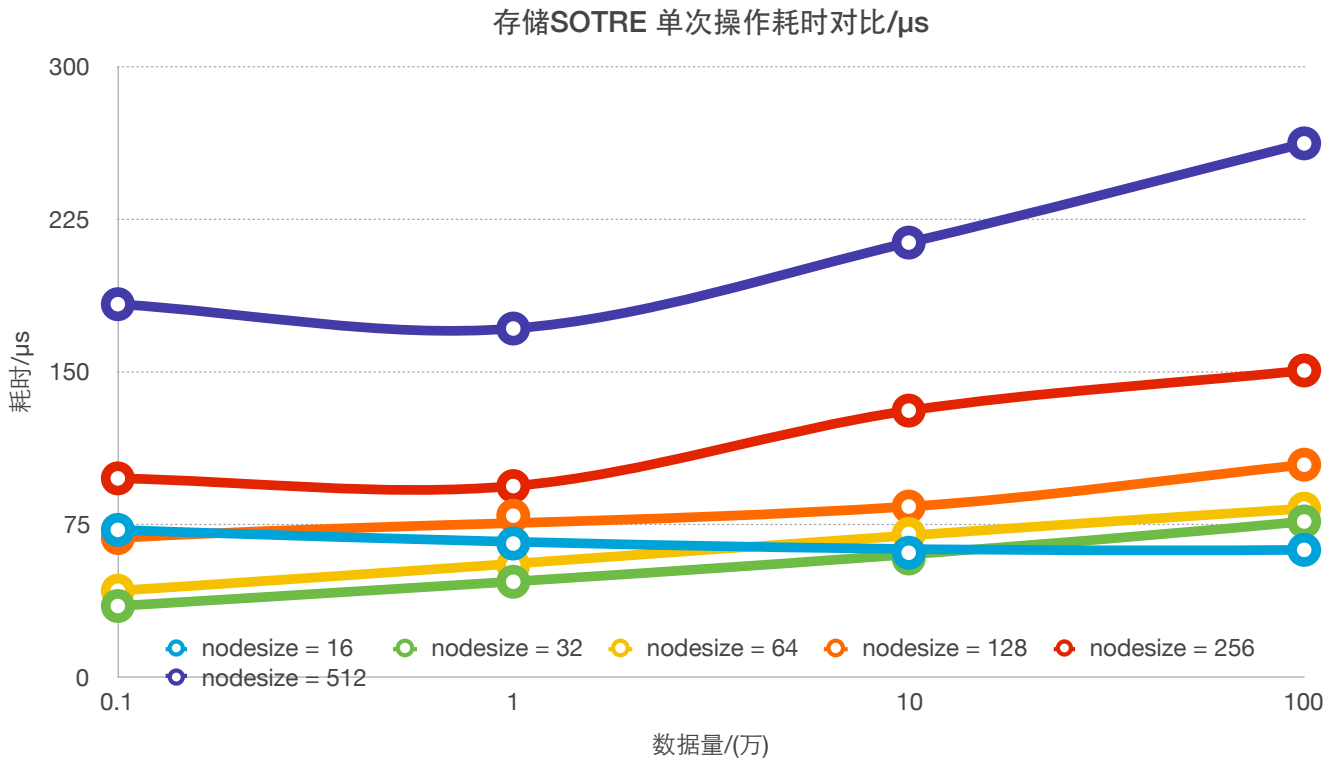
观察图表，可大致认为随数据量增加，单次存储或删除操作成对数式增加（趋势线）。且存储与删除操作的曲线非常接近。

- **查找FETCH操作：**
- 搜索树时间与树高h成正比；但需要直接到数据文件中读取数据，因此速度在节点大小较小时较慢，但随着节点大小增加，速度快于存储与删除操作。

  观察图表，可近似认为查找数据时间有对数曲线趋势，但不明显
- **修改MODIFY操作：**
- 搜索树时间与树高h成正比；此后不需要解决其他问题，只需要把修改数据信息放入写缓存。因此速度快于上述三种操作，且最严格的成对数形增长（趋势线）。

2.2.4 操作时间与节点大小关系(correctness_test.cpp)

1. 存储STORE



存储SOTRE 单次操作耗时对比/μs

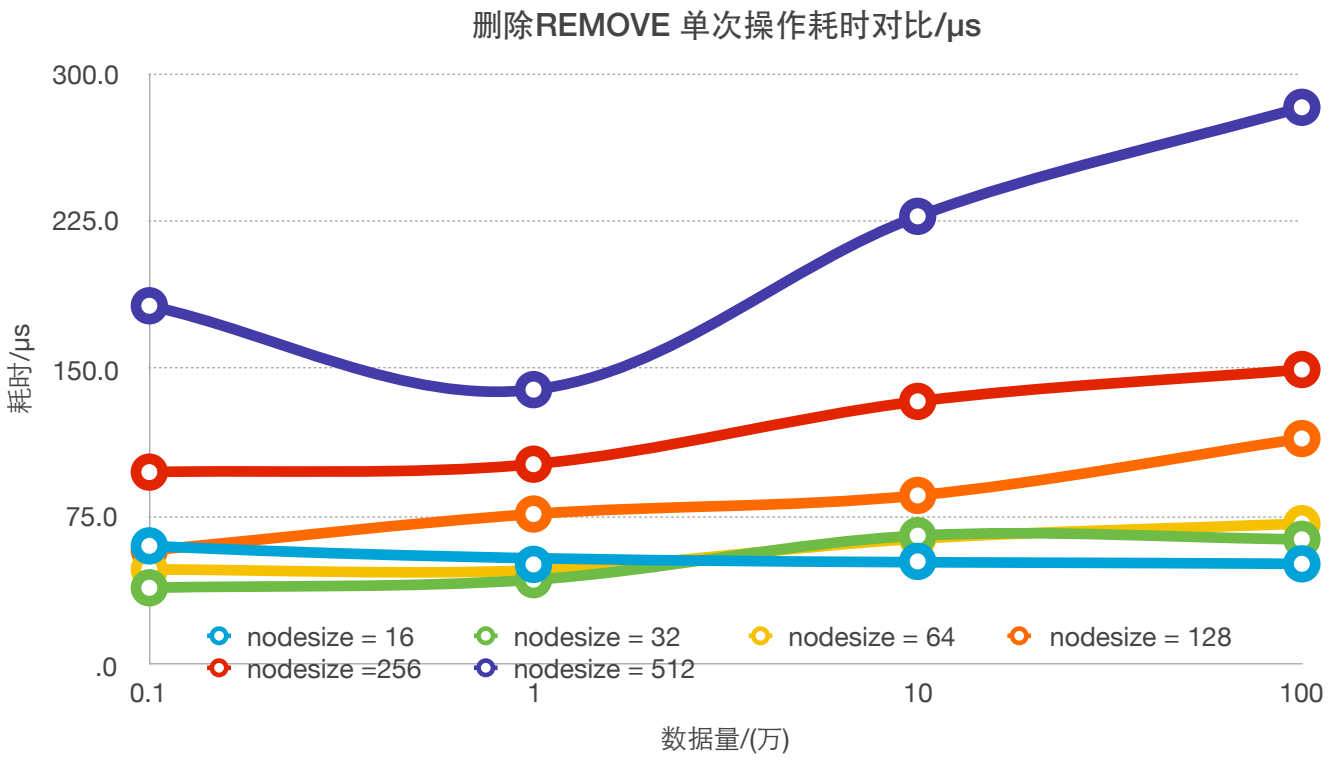
数据量/(万)	0.1	1	10	100
nodesize = 16	72.5	65.8	61.25	62.682
nodesize = 32	35.073	46.997	58.421	76.634
nodesize = 64	42.603	55.154	70.484	82.957

数据量/(万)	0.1	1	10	100
nodesize = 128	68.471	79.477	83.973	104.55
nodesize = 256	97.823	94.045	131.143	150.861
nodesize = 512	183.455	171.489	213.744	262.522

在数据量约为0.5M时，存储速度最快为120k/s；数据量为4M时，存储速度最快为70k/s

由图可知，节点大小在16-64之间速度最快。我有一些疑惑，一般会认为树高越低，即节点大小越大，操作时间越短。我猜测是内存中的操作可以优化，缩短时间，来体现大节点的优越性

2. 删除REMOVE



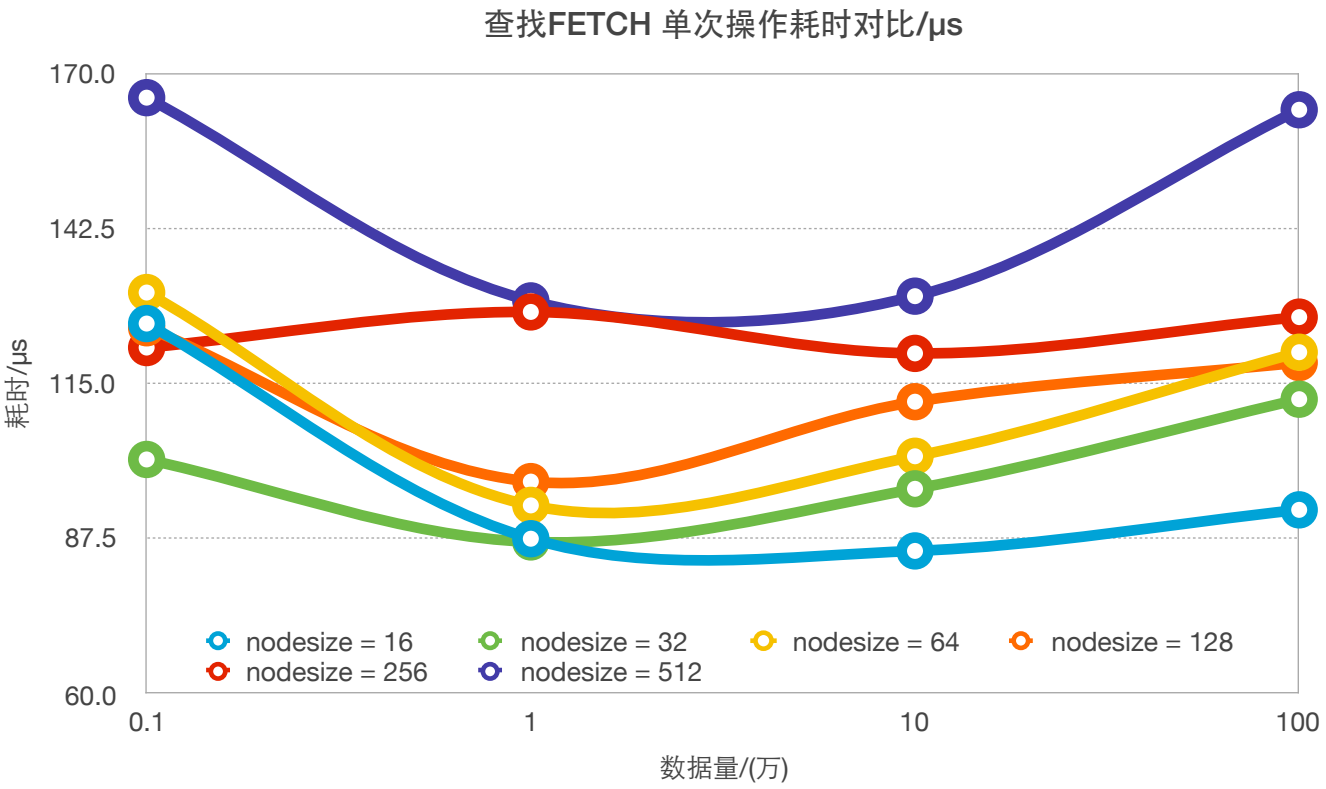
删除REMOVE 单次操作耗时对比/μs

数据量/(万)	0.1	1	10	100
nodesize = 16	60.3	50.9	52.4	51.187
nodesize = 32	39.025	43.141	65.469	63.621
nodesize = 64	48.353	47.933	63.903	71.586

数据量/(万)	0.1	1	10	100
nodesize = 128	57.924	76.437	85.878	114.736
nodesize =256	97.694	101.715	133.57	149.725
nodesize = 512	182.056	139.52	227.4	282.704

分析基本与存储操作相同。

3. 查找FETCH

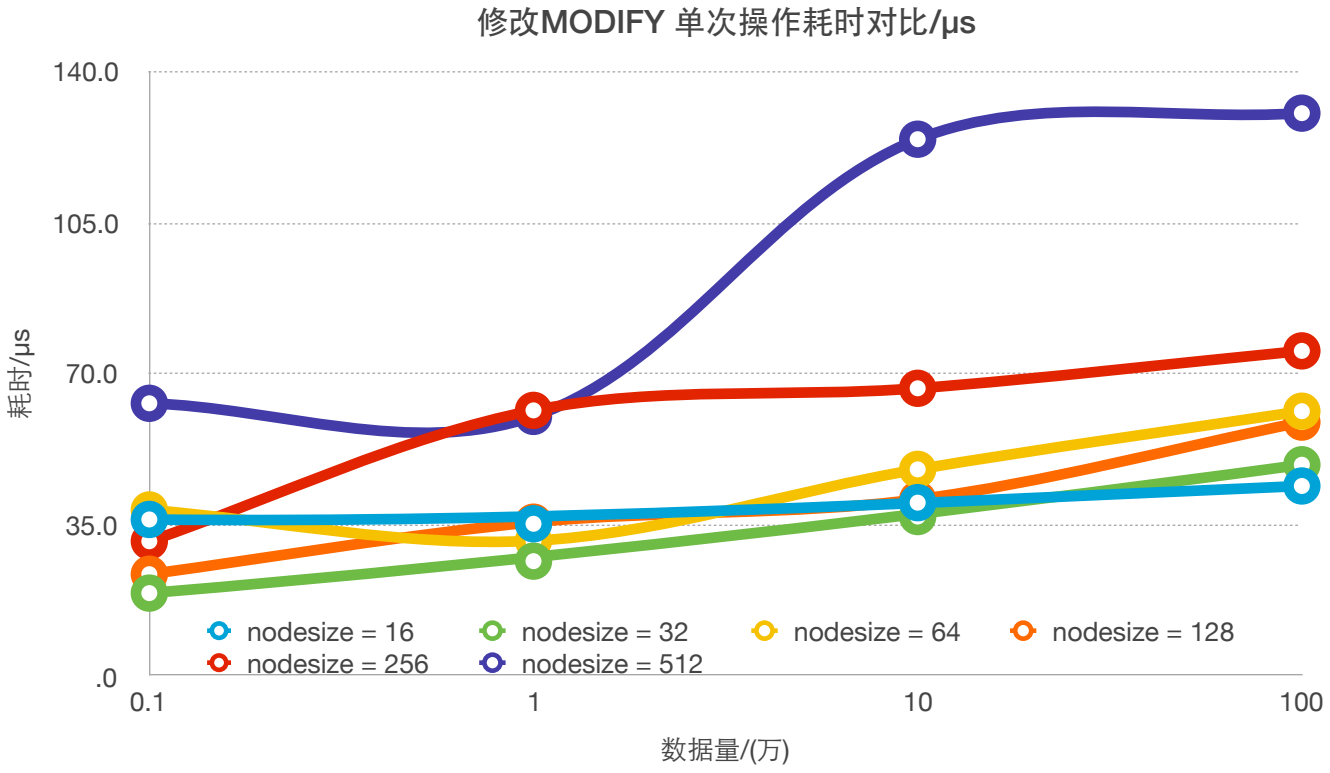


查找FETCH 单次操作耗时对比/μs

数据量/(万)	0.1	1	10	100
nodesize = 16	125.6	87.5	85.34	92.613
nodesize = 32	101.536	86.933	96.334	112.206
nodesize = 64	131.09	93.4	102.1	120.523
nodesize = 128	124.854	97.615	111.759	118.695
nodesize = 256	121.349	127.659	120.316	126.738
nodesize = 512	165.597	129.611	130.448	163.455

图表中值得注意的是小数据(0.1万)时不同大小的节点速度几乎相同。对此我有这样的解释：数据量小时，树高都较低，查找操作访问的节点数相近，因此耗费时间几乎相同。但费解的是在数据量达到1万时，操作时间集体下降，有可能是测量误差造成的。

4. 修改MODIFY



修改MODIFY 单次操作耗时对比/μs

数据量/(万)	0.1	1	10	100
nodesize = 16	36.2	35.2	40.1	43.97
nodesize = 32	19.108	26.524	36.879	48.901
nodesize = 64	38.4	31.25	47.826	61.325
nodesize = 128	23.5	35.574	40.864	58.796
nodesize = 256	31.14	61.519	66.643	75.326
nodesize = 512	63.161	60.119	124.455	130.503

修改操作中节点较小时，单次操作耗费时间相近；但当节点大小为512时，耗费时间大大增加，有些疑惑。测量上也许存在误差

七、待改进

- 空间重利用类始终存在于内存中，没有加入读入、保存的操作，使得重利用只能发生在一次打开数据库操作的过程中。由于时间限制，暂未完善

- 内存中的比较算法采用了向量中的顺序比较，可以采取更高效的算法进行比较，来体现大节点的时间优势