数据库管理系统性能测试实验报告

注：本项目开源在这儿。

1、环境简介

- 操作系统
  - 系统版本：Windows 11家庭版
  - 补丁版本：23H2
  - 内核版本：22631.2361
- 部分硬件信息
  - 处理器：Intel Core i5-1135G7@2.40GHz
  - RAM：DDR4 4*64-bit 16.0 GB
  - 内存频率：2133MHz
  - 内存CAS Latency：36
  - 硬盘性能：32MiB负载单线程随机4K读 25MB/s 写 80MB/s
- 数据库管理系统
  - Postgresql-x64-16rc1
  - 配置文件：config.xlsx
- 数据库客户端
  - Datagrip-2023.2.1
- 程序开发环境
  - 项目名称：TinyDB
  - 语言标准：C++17
  - 编译套件：GCC 13.2.0 x86_64-w64-mingw32
  - 自动化构建工具：CMake 3.27.0
  - 单元测试框架：GoogleTest-1.14.0
  - 性能分析工具：Intel VTune Profiler 2023.2

2、背景

数据集

使用 flights.sql SQL脚本构建。选择该数据集的原因如下：此数据集只有一张表，另外两个数据集films和shenzhen_metro有超过一张表，且存在外键等更多约束条件，TinyDB无法支持这些特性。因此为了控制变量确定DBMS的性能优势，故选择较为简洁的数据集。

性能测试项目

在TinyDB的底层实现中，重点操作是插入、查找、删除。我基于现实需求，判定该数据集的工作负荷是查询密集型，因此对模糊查询功能有针对性优化，精确查询的复杂度与模糊查询基本一致。在DBMS的执行计划中，该数据集对模糊查询和精确查询的查询逻辑并无较大差别，因此对查询功能的测试选择模糊查询。

从数据库操作命令角度看，性能测试涉及的命令有以下部分：

```sql
INSERT INTO flights (departure, arrival, day_op, dep_time, carrier, airline, flightnum, duration, aircraft) VALUES ('ACC', 'AMS', '1234567', '21:50', 'KL', 'KL', 590, 420, '330');
DELETE FROM flights WHERE departure like '%DEN%';
UPDATE flights set departure = 'BBC' WHERE departure like '%DEN%';
SELECT COUNT(*) FROM flights WHERE departure like '%DEN%' or arrival like '%DEN%';
```

测试项目解释：

注1：由于个人计算机对性能测试的不稳定因素过多，包括但不限于：处理器性能波动、操作系统调度进程优先级差异、内存频率波动、数据库管理系统配置（例如auto）、后台其他程序的影响、单个操作的定义等。我无法精准说明每个“操作”的性能表现。因此在数据库管理系统的性能分析中，我使用自带的 `EXPLAIN` 命令分析不同语句的执行效果，所有测试结果仅对我本地数据集当时测试环境有效。虽然底层性能分析工具可以识别详细的处理器缓存命中、内存读写、硬盘读写性能，有助于分析DBMS的性能瓶颈。但不考虑使用 Intel VTune 等处理器级别性能分析工具来测试DBMS的原因是，我不知道DBMS的后台服务具体执行了哪些操作。我会在TinyDB的性能分析中使用此类软件，详细说明性能瓶颈。

注2：在刚启动数据库后台服务时，执行的第一条语句是没有缓冲区加速的。在粗略测试中，只观察语句执行结果的运行时间，无加速的查询语句的 `fetching time` 是有缓冲区加速的三倍左右，具体数值为 $44ms$ 与 $15ms$，该数据明显受到硬盘读写性能影响。我认为该数据集是作为查询平台的后端，仅管理员会进行航线的增减，因此是长期部署的查询密集型任务。对于刚启动服务的第一个查询语句性能损失并不是重点关注对象，因此在DBMS的性能分析中不会将读写硬盘的性能瓶颈作为主要考虑对象。

- `INSERT`
  - `INSERT` 每次只能插入单条数据，因此在测试时会连续插入多条数据取平均值，计算单次插入的耗时。此命令仅作为单条插入的示例，实际测试的工作负荷为整张表共 74349 条数据。
  - 由于本数据集不涉及索引建立，不涉及网络延迟和多线程读写，且插入的数据结构单一，因此性能瓶颈在存储器上。鉴于数据量不超过数据库管理系统设置的缓冲区上限，可以判断RAM是主要的瓶颈。总插入执行耗时如下：

    > 摘要：在 19秒549毫秒中74,352/74,352 条语句已执行 (文件中有 13,010,756 个符号)

- `DELETE`
  - `DELETE` 支持模糊删除，根据执行计划确定可以借此测试删除单条数据的耗时。
  - 性能分析结果如下：

```sql
BEGIN TRANSACTION;
EXPLAIN (analyse, buffers, verbose)
    select count(*) from flight.flights where departure like '%DEN%';
EXPLAIN (analyse, buffers, verbose)
    DELETE FROM flight.flights WHERE departure like '%DEN%';
EXPLAIN (analyse, buffers, verbose)
    select count(*) from flight.flights where departure like '%DEN%';
ROLLBACK;
```

> Aggregate (cost=1695.19..1695.20 rows=1 width=8) (actual time=7.841..7.842 rows=1 loops=1)
> Output: count(*)
> Buffers: shared hit=759 dirtied=23
> -> Seq Scan on flight.flights (cost=0.00..1688.36 rows=2731 width=0) (actual

```
time=7.395..7.720 rows=2725 loops=1)
"       Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum,
duration, aircraft"
    Filter: ((flights.departure)::text ~~ '%DEN%'::text)
    Rows Removed by Filter: 71624
    Buffers: shared hit=759 dirtied=23
Planning Time: 0.092 ms
Execution Time: 7.866 ms

Delete on flight.flights  (cost=0.00..1688.36 rows=0 width=0) (actual
time=6.066..6.066 rows=0 loops=1)
Buffers: shared hit=3484
->  Seq Scan on flight.flights  (cost=0.00..1688.36 rows=2731 width=6) (actual
time=4.720..5.119 rows=2725 loops=1)
    Output: ctid
    Filter: ((flights.departure)::text ~~ '%DEN%'::text)
    Rows Removed by Filter: 71624
    Buffers: shared hit=759
Planning Time: 0.056 ms
Execution Time: 6.083 ms

Aggregate  (cost=1695.19..1695.20 rows=1 width=8) (actual time=4.883..4.884
rows=1 loops=1)
Output: count(*)
Buffers: shared hit=759
->  Seq Scan on flight.flights  (cost=0.00..1688.36 rows=2731 width=0) (actual
time=4.880..4.880 rows=0 loops=1)
"       Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum,
duration, aircraft"
    Filter: ((flights.departure)::text ~~ '%DEN%'::text)
    Rows Removed by Filter: 71624
    Buffers: shared hit=759
Planning Time: 0.058 ms
Execution Time: 4.902 ms
```

- 利用事务机制回滚防止 `DELETE` 生效的同时，可以注意到几点关键性能信息。
  - 此分析是我多次执行这段语句的结果，两次查询的耗时差别较大，但顺序查询节点耗时不同。一方面是查询有数据和没有数据，另一方面是存在缓冲区数据页被污染的情况。前一个影响因素在两次运行之间等待时间较长时，会得到另一个结果：

```
Aggregate  (cost=1695.19..1695.20 rows=1 width=8) (actual time=5.604..5.605
rows=1 loops=1)
Output: count(*)
Buffers: shared hit=759
->  Seq Scan on flight.flights  (cost=0.00..1688.36 rows=2731 width=0) (actual
time=5.149..5.489 rows=2725 loops=1)
"       Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum,
duration, aircraft"
    Filter: ((flights.departure)::text ~~ '%DEN%'::text)
    Rows Removed by Filter: 71624
    Buffers: shared hit=759
```

> Planning Time: 0.092 ms
> Execution Time: 5.645 ms
>
> Delete on flight.flights (cost=0.00..1688.36 rows=0 width=0) (actual time=6.104..6.105 rows=0 loops=1)
> Buffers: shared hit=3484
> -> Seq Scan on flight.flights (cost=0.00..1688.36 rows=2731 width=6) (actual time=4.844..5.238 rows=2725 loops=1)
> Output: ctid
> Filter: ((flights.departure)::text ~~ '%DEN%'::text)
> Rows Removed by Filter: 71624
> Buffers: shared hit=759
> Planning Time: 0.057 ms
> Execution Time: 6.120 ms
>
> Aggregate (cost=1695.19..1695.20 rows=1 width=8) (actual time=5.166..5.167 rows=1 loops=1)
> Output: count(*)
> Buffers: shared hit=759
> -> Seq Scan on flight.flights (cost=0.00..1688.36 rows=2731 width=0) (actual time=5.162..5.162 rows=0 loops=1)
> " Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum, duration, aircraft"
> Filter: ((flights.departure)::text ~~ '%DEN%'::text)
> Rows Removed by Filter: 71624
> Buffers: shared hit=759
> Planning Time: 0.081 ms
> Execution Time: 5.189 ms

- 注意到此时被污染的缓冲页面已经被写回磁盘，两次查询差别仅有数据量的不同。线性扫描节点的耗时差距为0.340ms，总时间差距为0.454ms。扫描节点的瓶颈为74.8%，统计函数的瓶颈为25.2%。而含有污染页面的耗时差距为2.964ms，可分析判定其中85%的性能耗时由23个被污染的缓冲页面所导致。
- `DELETE` 操作在线性扫描节点的耗时与查询一致，额外耗时主要因为对较多缓冲区进行修改。由此可以判断，在对数据做删除操作时，性能瓶颈在缓冲区的读写性能和缓冲策略上。

- `UPDATE`
  - `UPDATE` 支持模糊更新，根据执行计划确定可以判断修改数据的耗时。
  - 性能分析结果如下：

```
1 EXPLAIN (analyse, buffers, verbose)
2     update flight.flights set departure = 'BBC' where departure like '%DEN%';
3 EXPLAIN (analyse, buffers, verbose)
4     update flight.flights set departure = 'DEN' where departure like '%BBC%';
```

> Update on flight.flights (cost=0.00..1671.81 rows=0 width=0) (actual time=9.770..9.771 rows=0 loops=1)
> Buffers: shared hit=8827 dirtied=23 written=23
> -> Seq Scan on flight.flights (cost=0.00..1671.81 rows=3054 width=30) (actual

> time=4.663..5.338 rows=2725 loops=1)
> " Output: 'BBC'::character varying(5), ctid"
> Filter: ((flights.departure)::text ~~ '%DEN%'::text)
> Rows Removed by Filter: 71624
> Buffers: shared hit=669
> Planning Time: 0.080 ms
> Execution Time: 9.790 ms
>
> Update on flight.flights (cost=0.00..1729.29 rows=0 width=0) (actual time=9.886..9.887 rows=0 loops=1)
> Buffers: shared hit=8847 dirtied=22 written=22
> -> Seq Scan on flight.flights (cost=0.00..1729.29 rows=81 width=30) (actual time=5.676..6.300 rows=2725 loops=1)
> " Output: 'DEN'::character varying(5), ctid"
> Filter: ((flights.departure)::text ~~ '%BBC%'::text)
> Rows Removed by Filter: 71624
> Buffers: shared hit=692
> Planning Time: 0.063 ms
> Execution Time: 9.905 ms

  - 两次的顺序查询耗时分别为0.675ms和0.624ms，但总耗时接近10ms。由此可分析得知在对数据进行更新操作时，与删除操作一致，性能瓶颈在缓冲区的读写上，且查询的耗时占比在6.5%左右，可以忽略。

- `SELECT`
  - 若 `SELECT` 查询的条目过多，检索数据的时间较高，即 `fetching time`。该性能指标与各级缓存的数据吞吐量相关，性能瓶颈在其他硬件上而不是数据库管理系统。根据执行计划确定，Count函数对性能没有显著影响，故使用此以减少输出数据量，减少查询外耗时。
  - 性能分析结果如下：

```
EXPLAIN (analyse, buffers, verbose) select * from flight.flights
where departure like '%DEN%' or arrival like '%DEN%';
EXPLAIN (analyse, buffers, verbose) select count(*) from
flight.flights where departure like '%DEN%' or arrival like '%DEN%';
```

> Seq Scan on flight.flights (cost=0.00..1746.43 rows=5681 width=36) (actual time=0.308..9.555 rows=5576 loops=1)
> " Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum, duration, aircraft"
> Filter: (((flights.departure)::text ~~ '%DEN%'::text) OR ((flights.arrival)::text ~~ '%DEN%'::text))
> Rows Removed by Filter: 68773
> Buffers: shared hit=624
> Planning Time: 0.069 ms
> Execution Time: 9.722 ms
>
> Aggregate (cost=1760.64..1760.65 rows=1 width=8) (actual time=8.415..8.416 rows=1 loops=1)
> Output: count(*)
> Buffers: shared hit=624
> -> Seq Scan on flight.flights (cost=0.00..1746.43 rows=5681 width=0) (actual time=0.230..8.205 rows=5576 loops=1)
> " Output: departure, arrival, day_op, dep_time, carrier, airline, flightnum,

```
duration, aircraft"
  Filter: (((flights.departure)::text ~~ '%DEN%'::text) OR ((flights.arrival)::text ~~
'%DEN%'::text))
  Rows Removed by Filter: 68773
  Buffers: shared hit=624
Planning Time: 0.073 ms
Execution Time: 8.436 ms
```

- 可以发现Count函数的运行时间不超过0.001ms，且两次查询都是使用顺序查询节点，过滤器一致，缓存命中一致。虽然多使用一次Count函数，但全条目查询的时间比Count的结果还多了1.286ms，因此查询外耗时为1.287ms。由该性能分析知晓，在进行 `SELECT` 测试时，建议使用Count函数，以减少线性查询器以外的耗时，便于对比性能。

3、TinyDB系统设计与实现

需求分析

功能性需求

该数据集将作为在线航线查询系统的后端数据库，TinyDB作为基于内存的简易数据库管理系统后端实现，需要支持四种基本功能：插入、删除、修改、查询。TinyDB不需要支持的功能包括但不限于：多用户系统、网络交互、命令解析、高并发交互等。

TinyDB的交互方式为命令行交互，测试时使用项目测试文件。目标工作负载为查询密集型，处理数据的量级最高为100MB。采用多线程文件读写分离，文件分片存储以减少硬盘读写对主程序的影响。安全方面对用户输入完全可信，不支持预编译等注入攻击防护策略，也不支持通用SQL语言的执行。

非功能性需求

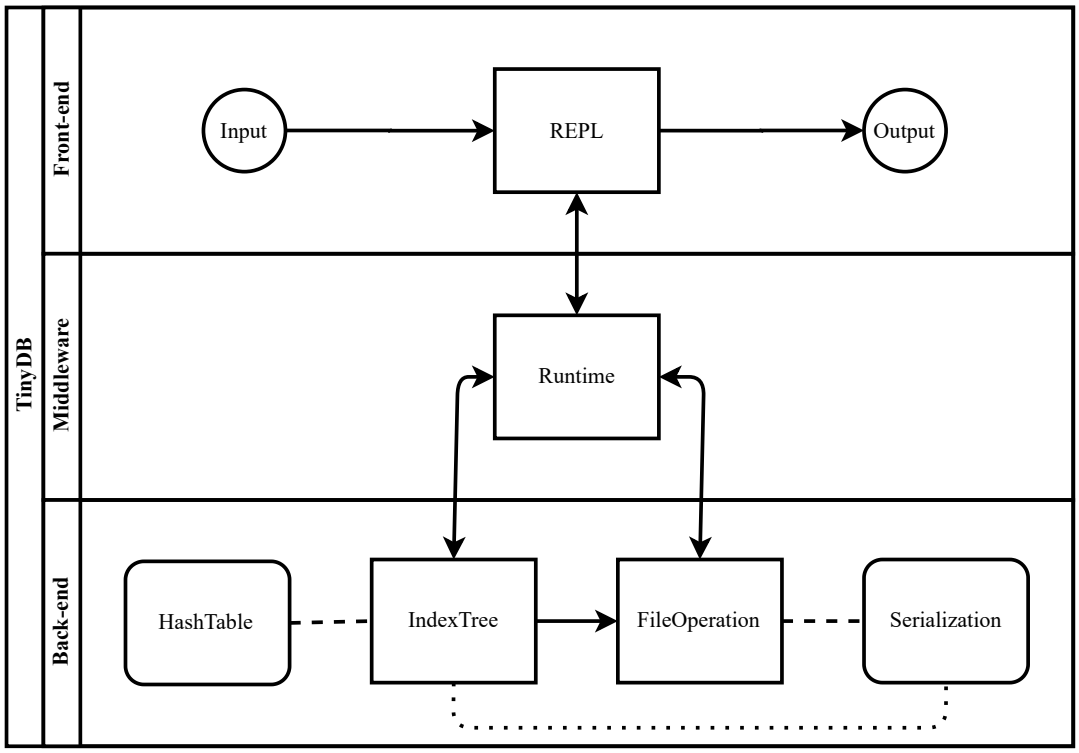
TinyDB将使用C++17标准开发，主要遵循 CERT 开发规范，使用CMake自动化构建工具便于跨平台模块化编译，发布版使用-O3 -march=native -mtune=native编译选项。

概要设计

总体设计

- 系统架构

- 架构图如下：



- 模块概述
  - REPL：前端交互模块，用于将用户输入命令转化为操作码，并接受来自Runtime的返回信息，输出到用户界面。
  - Runtime：运行时系统模块，用于将操作码转化为后端函数调用，执行异常处理和资源调度。
  - IndexTree：索引树模块，用于管理数据节点的各级缓存，在内存中执行增删改查等数据库系统管理操作。
  - HashTable：散列表模块，为索引树模块的子模块，用于生成缓存和散列表，加速模糊查询。
  - FileOperation：文件操作模块，用于管理缓存与硬盘的读写。
  - Serialization：序列化模块，为文件操作模块的子模块，用于处理缓存和数据节点与二进制流的互相转换。
- 处理流程
  1. 用户命令作为字符串，输入到REPL中。REPL将字符串解析为Statement类型，传递到Runtime。
  2. Runtime解析Statement，产生对应的函数调用。
  3. IndexTree根据来自Runtime的调用，与HashTable配合执行，返回调用结果。当IndexTree需要进行文件读写时，生成任务传递到FileOperation，两个后端主模块并行执行。
  4. Runtime调度FileOperation的任务执行，接受来自IndexTree和FileOperation的返回值，再返回到REPL。
  5. REPL接收Runtime的执行结果，输出到用户界面。

外部接口

- 外部用户通过命令行或命令文件与系统交互，没有提供硬件分析接口。
- 用户交互的元命令限定为以下四种：

```
.help
.last
.load
.exit
```

其中 `.load` 可以自定义输入文件路径，`.last` 返回上次输入的命令。

- 用户交互的输入SQL命令限定为以下五种：

```
CREATE tablename (column1,column2,...,columnn);
INSERT INTO tablename VALUES (column1,column2,...,columnn);
DELETE FROM tablename WHERE column LIKE 'data';
UPDATE tablename SET column='data' WHERE column LIKE 'data';
SELECT column1,column2,...,columnn FROM tablename WHERE column LIKE 'data';
```

其中，小写部分表示可根据需求自定义字符串，大写部分不可更改，特殊符号和空格不可更改。不同命令的返回值在模块设计中说明。

模块设计

- REPL
  - 该模块主要包含两部分，元命令解析和SQL命令解析。
    - 元命令解析函数 `parse_meta_cmd()` 仅涉及REPL，用于打印帮助信息、上次命令、退出交互端等交互操作。
    - SQL命令解析函数 `parse_statement()` 用于解析用户输入的SQL命令，另有三个辅助函数：解析表名 `parse_table()`、解析WHERE子句 `parse_where()`、解析数据 `parse_datas()`。这三个辅助函数只被SQL命令解析函数调用，分别与主解析函数内容耦合。
  - REPL不直接传递数据到Runtime，由TinyDB框架间接传递到Runtime中。REPL的SQL命令解析函数返回Runtime模块内特有数据结构变量，两模块特征耦合。
  - REPL需要由TinyDB先判断输入是元命令还是SQL命令，再分别调用不同的解析函数。因此模块核心接口是 `bool parse_meta_cmd(const std::string &)` 和 `Runtime::Statement parse_statement(const std::string &)`。
- Runtime
  - 该模块主要包含两部分，解析执行Statement和资源调度。
    - 解析执行函数 `run_statement()` 用于解析Statement操作码，调用不同的IndexTree模块内函数以执行操作。
    - 资源调度函数 `schedule()` 用于管理FileOperation的读写任务。该函数调用FileOperation模块内函数将缓冲区内未写入硬盘的数据写入。此函数的执行线程与主线程分离，对队列施加互斥锁避免读写竞争，使用条件变量同步执行线程和主线程的任务队列情况，不使用时间轮询以降低处理器开销。
  - Runtime直接调用IndexTree和FileOperation模块内函数，在Runtime模块内解析调用返回值和异常处理，只以 `std::vector<std::string>` 数据类型返回给REPL执行结果。当产生异常时，`vector` 内有且仅有两个元素，第一个元素为异常类型，第二个元素为异常信息。Runtime与IndexTree和FileOperation控制耦合。
  - Runtime的核心接口是 `std::vector<std::string> run_statement(const Runtime::Statement &)`，重要接口是 `bool is_finish(void)`，该接口返回任务队列是否执行完毕，REPL会利用该接口判断能否结束交互。

- IndexTree
  - 该模块是后端核心模块，实现了较多函数，其中重要函数是插入、查询、模糊查询、删除。
    - 插入函数 `insert()` 用于向索引树中插入指定键值的一条数据，在数据插入后需要调用HashTable模块内函数用于构建散列表，并在索引树中修改缓存。
    - 查询函数 `find_leaf()` 用于在索引树中查询指定键值的叶节点，叶节点内**可能**包含不止一条数据。该函数只用于找到**可能**存在该键值的叶节点，不保证返回的叶节点一定包含指定键值的数据，但不在返回结果内的叶节点一定不包含指定键值。
    - 模糊查询函数 `fuzzy_find_leaf()` 用于在索引树中以**键值子串匹配**形式查找可能符合条件的多个叶节点。该函数可能返回多个叶节点，其他特性与查询函数基本一致。
    - 删除函数 `remove()` 用于在索引树中删除指定键值的一条数据，若不存在指定键值则不做操作。
    - 在以上函数外，还有其他函数用于维护索引树和散列表的结构。这些维护用函数只被重要函数调用，不被其他模块函数调用。
  - IndexTree在数据修改时会先调用Serialization模块内函数用于序列化数据结构，再调用FileOperation模块内函数用于添加任务，在更新索引和构建散列表时会调用HashTable内函数。IndeTree与HashTable和Serialization特征耦合，与FileOperation数据耦合。
  - IndexTree的核心接口是 `void insert(std::shared_ptr<IndexTree::Tree>, const std::string &, const std::vector<std::string> &)`，其中索引树以共享指针的形式支持对多个树的操作。还有两个核心接口是 `std::vector<std::shared_ptr<IndexTree::LeafNode>> fuzzy_find_leaf(std::shared_ptr<IndexTree::Tree>, const std::string &)` 和 `void remove(std::shared_ptr<IndexTree::Tree>, const std::string &)`。
- HashTable
  - 该模块是IndexTree的子模块，主要是创建散列表和查询散列表。
    - 创建散列表函数 `create_map()` 用于根据键值生成离散点集，用于标记散列表。
    - 查询散列表函数 `find()` 用于根据指定散列表，查询指定键值是否可能存在子串匹配。
  - HashTable模块不会调用其他模块的函数，只会被IndexTree调用，不存在以该模块触发的耦合情景。
  - HashTable的核心接口是 `std::vector<int> create_map(const std::string &, unsigned long long)` 和 `bool find(const std::bitset<HashTable::prime> &, const unsigned long long &, const std::string &)`。其中HashTable的数据结构使用 `std::bitset` 实现，离散点集使用 `std::vector<int>` 传递下标。
- FileOperation
  - 该模块用于对数据文件进行修改操作，主要由任务队列和文件读写两部分组成。
    - 任务创建函数 `produce_task()` 与任务提取函数 `consume_task()` 用于维护任务队列，任务由IndexTree模块发起，由Runtime模块调度任务队列。
    - 文件读写函数 `work()` 用于执行任务，在其中根据不同请求对文件进行操作。
  - FileOperation在保存索引树时会调用Serialization模块内函数，两模块特征耦合。
  - FileOperation的核心接口是 `void produce_task(const FileOperation::Task &)`，`FileOperation::Task consume_task(void)` 和 `void work(const FileOperation::Task &)`，`FileOperation::Task` 是该模块内特有的数据结构。
- Serialization

- 该模块用于将IndexTree和HashTable模块内数据结构序列化为字符串，再被传递到FileOperation进行文件读写操作。
  - HashTable序列化与反序列化函数 `serialize_map(), deserialize_map()` 用于将 `std::bitset` 与 `std::string` 互相转换。
  - IndexTree序列化与反序列化函数 `serialize_tree(), deserialize_tree()` 用于将 `IndexTree::Tree` 与 `std::string` 互相转换。
- Serialization不会调用其他模块内函数，不存在以该模块触发的耦合情景。
- Serialization的核心接口是序列化与反序列化函数 `std::string serialize_map(const std::bitset<HashTable::prime> &), std::bitset<HashTable::prime> deserialize_map(const std::string &), std::string serialize_tree(const std::shared_ptr<IndexTree::Tree> &), std::shared_ptr<IndexTree::Tree> deserialize_tree(const std::string &)`。

详细设计

TODO 分为不同模块，各包含以下部分：算法设计、数据结构设计、接口实现、属性、参数。

- REPL
  - 元命令解析函数 `parse_meta_cmd()` 使用 `std::unordered_map<std::string, std::function<bool()>>` 匹配命令集，在 `std::function` 中实现对应操作。
  - SQL命令解析函数 `parse_statement()` 基于朴素字符串匹配，使用 `std::string.compare()` 比较特定位置的关键字。对于括号内不定长字串使用 `std::string_view.find()` 枚举分隔符。
  - 假设命令串长度为 n，则解析时空复杂度为 $\Theta(n)$。
- Runtime
  - 执行函数 `run_statement()` 对不同操作类型有以下不同的实现方式。
    - 对于 `CREATE` 命令，调用 `IndexTree::build()` 函数。
    - 对于 `INSERT` 命令，先根据数据特征拼接键值，再调用 `IndexTree::insert()` 函数。
    - 对于 `DELETE` 命令，先调用 `IndexTree::fuzzy_find_leaf()` 函数获取所有可能要删除的叶节点。再遍历该集合，确定每个叶节点内被删除的数据键值。最后调用 `IndexTree::remove()` 函数逐一删除每个键值。
    - 对于 `UPDATE` 命令，与 `DELETE` 类似获取所有可能要修改的叶节点和被修改的数据键值。之后对每一个键值先调用 `IndexTree::remove()` 删去后，再调用 `IndexTree::insert()` 插入更新后的键值。
    - 对于 `SELECT` 命令，调用 `IndexTree::fuzzy_find_leaf()` 函数获取所有可能匹配的叶节点。再遍历该集合，使用 `std::string.find()` 逐一确定数据条，并返回匹配的数据集合。
  - 调度函数 `schedualer()` 作为独立线程运行，在调用执行函数时，在执行函数结束后被唤醒。调用 `FileOperation` 模块内函数执行任务时，在调度函数线程内运行。单次任务队列清空后该线程结束。
- IndexTree
  - 索引树采用 `B+` 树数据结构维护，单点最大索引数与叶节点最大数据条数均为 64。所有索引树相关指针均使用 `shared_ptr`，避免内存分配相关安全问题。
  - 建树函数 `build()` 会设置树根指针和随机数种子，时空复杂度为 $\Theta(1)$。

    - 插入函数 `insert()` 会根据各层索引结点向下找到待修改的叶节点，将数据插入叶节点后递归返回根结点，同时维护索引树。假设数据长度为 n，索引树中叶节点个数为 m，叶节点的最大数据条数为 k，则时间复杂度为 $\Theta(n + k\log m + k\log k)$。此处 k 被设定为 64，n 不会超过 200，m 量级在 70000 以上，因此时间复杂度为 $\Theta(n + k\log m)$，空间复杂度为 $\Theta(nm)$。
    - 模糊查找函数 `fuzzy_find_leaf()` 会在索引树上进行宽度优先搜索，当前版本为非递归单线程形式。对于索引树上结点，会逐一比对散列表和键值特征点，对于叶节点在判断为可能存在后加入返回值中。单结点比对时间复杂度为 $\Theta(1)$，总时间复杂度为 $O(m), \Omega(\log m)$，最坏情况是每个节点都被返回，最优情况是仅命中单一叶节点。设散列表大小为 p，空间复杂度为 $\Theta(mp)$。
    - 删除函数 `remove()` 会先调用 `find_leaf()` 函数向下找到待修改叶节点，修改后递归返回根结点，同时维护索引树。`find_leaf()` 函数的时间复杂度为 $\Theta(\log m)$，递归回根结点的维护时间复杂度也是 $\Theta(k\log m)$。
- HashTable
    - 散列表采用 `Bloom Filter`，对数据键值的每个子串提取特征点后加入散列表。预设不同子串最大可能值是 500，在误报率大约为 0.0001 时，使用散列表大小为 9587（质数），特征点个数为 13。在主程序首次运行时，调用系统真随机数发生器作为 `mt19937` 伪随机数生成器种子，使用该生成器为每个索引树发放种子。散列函数选择 `Murmurhash3_64`，将两个64位结果以不同倍数组合后对 9587 取模，得到 13 个特征点。底层数据结构使用 `std::bitset`，在索引树非叶结点合并子结点散列表时，使用按位或操作加速运算。
    - 根据数据长度，计算特征点的时空复杂度是 $\Theta(n)$，因此插入数据和查询数据的效率也是这个。
    - 在合并散列表时，根据64位字长机器和编译器的优化，散列表大小为 p 时，时空复杂度为 $\Theta(\frac{p}{\omega})$，其中 ω 为机器字长。
- FileOperation
    - 任务队列中使用 `std::lock_guard<std::mutex>` 对任务创建函数和任务提取函数上锁，两处使用的锁在命名空间内相同且唯一。另创建 `is_empty()` 函数供调度函数判断任务队列是否为空。任务创建函数和任务提取函数仅负责操作任务队列，任务执行由其他函数进行，该模块内仅这两个函数涉及多线程。
    - 任务执行函数即文件读写部分使用标准输入输出流控制，以字符形式读写文件。
- Serialization
    - 统一使用 `Base64` 将字节编码为可读写字符，供 `FileOperation` 模块使用。
    - 对散列表的序列化中，会将每 8 个 `bit` 按位存入字符数组中，再交由编码函数进行编码，反序列化部分同理。
    - 对索引树的序列化中，会将每个叶节点的数据条目转存到字符数组中，仅编码原始数据不编码索引部分。对非叶结点会完整记录结点连边信息后，将散列表序列化后拼接为数据段的一部分，整体再次编码。反序列化时先根据连边信息重建非叶结点，再根据叶节点顺序还原索引树。

4、TinyDB系统性能分析

注：由于TinyDB系统架构将文件读写和内存中数据操作分离（使用 `thread.detach()`），且单线程每次读写文件块大小不超过10KB，因此测试时仅需计算主线程性能。且由于系统性能主要与内存中数据操作有关，因此没有完全实现文件读写模块（`Serialization` 的 `serialize_tree` 与 `deserialize_tree`）。针对文件读写部分，存在多线程并发读写某文件的情景，我编写了相关单元测试，证明该模块支持多线程并发。

- 多线程相关单元测试代码

```
TEST(FileOperationTest, MultiThread)
{
    FileOperation fileOp;
    std::ofstream out("test.txt");
    out << "Hello, world!" << std::endl;
    out.close();
    std::ofstream out2("test2.txt");
    out2 << "This is a test." << std::endl;
    out2.close();
    FileOperation::Task task(FileOperation::Operation::APPEND,
"test.txt", "test2.txt");
    std::thread t1(&FileOperation::work, &fileOp, task);
    std::thread t2(&FileOperation::work, &fileOp, task);
    std::thread t3(&FileOperation::work, &fileOp, task);
    t1.join();
    t2.join();
    t3.join();
    std::ifstream in("test.txt");
    std::string content((std::istreambuf_iterator<char>(in)),
std::istreambuf_iterator<char>());
    EXPECT_EQ(content, "Hello, world!\nThis is a test.\nThis is a
test.\nThis is a test.\n");
    in.close();
    std::remove("test.txt");
    std::remove("test2.txt");
}
```

- 模块中实际部分代码（启用锁）

```
std::mutex *file_mutex, *file_mutex2 = nullptr;
{
    std::unique_lock<std::mutex> map_lock(map_mutex);
    file_mutex = &file_mutex_map[task.filename];
    if (task.opt == Operation::APPEND)
        file_mutex2 = &file_mutex_map[task.content];
}
std::unique_lock<std::mutex> lock(*file_mutex);
std::unique_lock<std::mutex> lock2;
if (task.opt == Operation::APPEND)
    lock2 = std::unique_lock<std::mutex>(*file_mutex2);
```

  - CTest测试结果如下

> [build] 生成已完成，退出代码为 0
> [proc] 执行命令: "C:\Program Files\CMake\bin\ctest.exe" -j10 -C Debug -T test --output-on-failure -R ^test_FileOperation$
> [ctest] Site: Cerulime
> [ctest] Build name: Win32-mingw32-make
> [ctest] Test project C:/Users/11247/Desktop/cs307proj1/build
> [ctest] Start 1: test_FileOperation
> [ctest] 1/1 Test #1: test_FileOperation ..... Passed 0.04 sec
> [ctest]

> [ctest] 100% tests passed, 0 tests failed out of 1
> [ctest]
> [ctest] Total Test time (real) =   0.06 sec
> [ctest] CTest 已完成，返回代码 0

- 不使用锁
  - CTest测试结果如下

> [build] 生成已完成，退出代码为 0
> [proc] 执行命令: "C:\Program Files\CMake\bin\ctest.exe" -j10 -C Debug -T test --output-on-failure -R ^test_FileOperation$
> [ctest]   Site: Cerulime
> [ctest]   Build name: Win32-mingw32-make
> [ctest] Test project C:/Users/11247/Desktop/cs307proj1/build
> [ctest]   Start 1: test_FileOperation
> [ctest] 1/1 Test #1: test_FileOperation .....***Failed   0.07 sec
> [ctest] Running main() from C:\Users\11247\Desktop\cs307proj1\build_deps\googletest-src\googletest\src\gtest_main.cc
> [ctest] [==========] Running 4 tests from 1 test suite.
> [ctest] [----------] Global test environment set-up.
> [ctest] [----------] 4 tests from FileOperationTest
> [ctest] [RUN   ] FileOperationTest.CreateFile
> [ctest] [OK] FileOperationTest.CreateFile (1 ms)
> [ctest] [RUN   ] FileOperationTest.AppendToFile
> [ctest] [OK] FileOperationTest.AppendToFile (4 ms)
> [ctest] [RUN   ] FileOperationTest.DeleteFile
> [ctest] [OK] FileOperationTest.DeleteFile (1 ms)
> [ctest] [RUN   ] FileOperationTest.MultiThread
> [ctest] C:\Users\11247\Desktop\cs307proj1\tests\test_FileOperation.cpp:67: Failure
> [ctest] Expected equality of these values:
> [ctest]   content
> [ctest]   Which is: "Hello, world!\nThis is a test.\nThis is a test.\n"
> [ctest]   "Hello, world!\nThis is a test.\nThis is a test.\nThis is a test.\n"
> [ctest] With diff:
> [ctest] @@ +1,4 @@
> [ctest] Hello, world!
> [ctest] +This is a test.
> [ctest] This is a test.
> [ctest] This is a test.\n
> [ctest]
> [ctest]
> [ctest] [FAILED] FileOperationTest.MultiThread (6 ms)
> [ctest] [----------] 4 tests from FileOperationTest (14 ms total)
> [ctest]
> [ctest] [----------] Global test environment tear-down
> [ctest] [==========] 4 tests from 1 test suite ran. (14 ms total)
> [ctest] [PASSED] 3 tests.
> [ctest] [FAILED] 1 test, listed below:
> [ctest] [FAILED] FileOperationTest.MultiThread
> [ctest]

> [ctest] 1 FAILED TEST
> [ctest]
> [ctest]
> [ctest] 0% tests passed, 1 tests failed out of 1
> [ctest]
> [ctest] Total Test time (real) =   0.09 sec
> [ctest]
> [ctest] The following tests FAILED:
> [ctest]   1 - test_FileOperation (Failed)
> [ctest] Errors while running CTest
> [proc] 命令“"C:\Program Files\CMake\bin\ctest.exe" -j10 -C Debug -T test --output-on-failure -R ^test_FileOperation$”已退出，代码为 8
> [ctest] CTest 已完成，返回代码 8

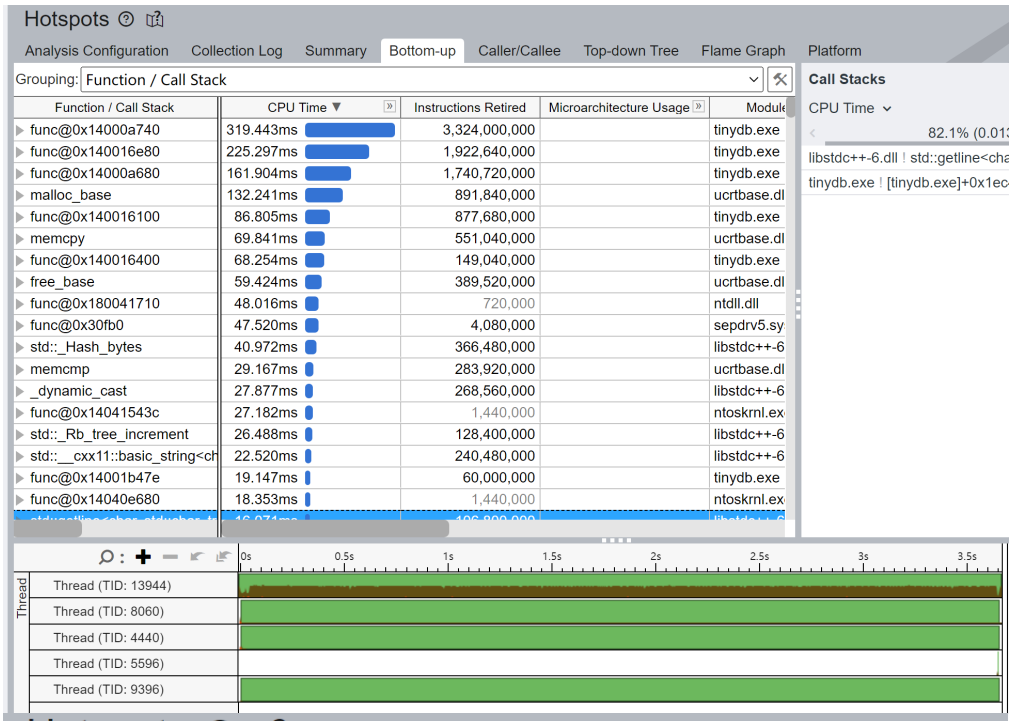
    - 在三线程情况下出现条件竞争，证明加锁可以避免出现该问题。

性能测试项目

该测试数据集使用 `dataset\flights.tiny` 文件。该测试计时采用项目内嵌 `std::chrono::high_resolution_clock::now()` 高精度计时器，计算从命令行中输入到完整执行单次命令结尾的时间。由于性能分析会影响处理器性能（硬件采样存在较高开销），在外部使用Intel VTune的性能分析信息作为辅助信息。所有结果采用 `0.1ms` 间隔与调用栈信息收集的硬件采样，VTune预计开销较高（共 9 级不同程度额外开销，该配置为 7 级）。

- `INSERT`
    - 测试时的 `.load` 为减少人手动输入导致的耗时，在项目内采用固定文件路径的方式测试。
    - 测试结果如下：

> TinyDB > The file path: Loading... 10000 of 10000 lines.
> Loading... 20000 of 20000 lines.
> Loading... 30000 of 30000 lines.
> Loading... 40000 of 40000 lines.
> Loading... 50000 of 50000 lines.
> Loading... 60000 of 60000 lines.
> Loading... 70000 of 70000 lines.
> ----------
> Task finished
> Load 74350 of 74350 lines.
> Table name: flights
> departure arrival day_op dep_time carrier airline flightnum duration aircraft
> 74349 rows in set.
> Time measured: 1331.74ms.
> ----------



Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Cal

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time ▼	Instructions Retired
tinydb.exe	984.221ms	8,735,040,000
func@0x14000a740	319.443ms	3,324,000,000
func@0x140016e80	225.297ms	1,922,640,000
func@0x14000a680	161.904ms	1,740,720,000
func@0x140016100	86.805ms	877,680,000
func@0x140016400	68.254ms	149,040,000
func@0x14001b47e	19.147ms	60,000,000
func@0x140013b40	13.393ms	36,960,000
func@0x140002630	9.325ms	87,360,000
func@0x14000ab00	8.234ms	38,880,000
func@0x14001b3e0	7.639ms	56,880,000
[tinydb.exe]	6.548ms	29,520,000
func@0x14001cbb0	6.448ms	76,320,000
func@0x140006e80	5.456ms	35,520,000
func@0x14001db20	5.059ms	67,200,000
func@0x14000b870	4.266ms	36,480,000
func@0x140018dd0	4.266ms	39,360,000
func@0x14000a930	3.472ms	6,480,000

Hotspots

Analysis Configuration | Collection Log | Summary | Bottom-up | Caller/Callee | Top-down Tree | Flame Graph

Grouping: Module / Function / Call Stack

Module / Function / Call Stack	CPU Time ▼	Instructions Retired	Microarchitecture Usage	Module
tinydb.exe	984.221ms	8,735,040,000		
ucrtbase.dll	304.264ms	2,166,240,000		
malloc_base	132.241ms	891,840,000		ucrtba…
memcpy	69.841ms	551,040,000		ucrtba…
free_base	59.424ms	389,520,000		ucrtba…
memcmp	29.167ms	283,920,000		ucrtba…
func@0x180050e90	7.242ms	5,280,000		ucrtba…
memchr	2.579ms	29,760,000		ucrtba…
free	1.190ms	6,480,000		ucrtba…
o_memset	1.190ms	5,280,000		ucrtba…
malloc	0.694ms	2,160,000		ucrtba…
calloc_base	0.099ms	0		ucrtba…
configure_narrow_argv	0.099ms	0		ucrtba…
rand_s	0.099ms	0		ucrtba…
func@0x180004280	0.099ms	0		ucrtba…
endthreadex	0.099ms	240,000		ucrtba…
func@0x18001bc60	0.099ms	240,000		ucrtba…
execute_onexit_table	0.099ms	240,000		ucrtba…
func@0x180051000	0ms	240,000		ucrtba…
libstdc++-6.dll	155.753ms	1,267,200,000		
ntoskrnl.exe	82.539ms	84,240,000		

    - 再次说明，性能分析由于存在采样开销，仅作为程序瓶颈分析工具。由热点分析可知，TinyDB主程序中处理器运行时间前五个的函数分别为：`IndexTree::merge_map(Internal)`, `HashTable::create_map()`, `IndexTree::merge_map(Leaf)`, `HashTable::get_hash()` 以及 `HashTable::create_map()` 内调用 `std::unordered_map<std::string_view, std::bitset<PRIME>> cache` 的成员函数 `find()`。在ucrtbase.dll（通用C运行时动态链接库）中，内存读写也有相当大的耗时。此外值得注意的是，`std::getline()`（位于libstdc++-6.dll中）用时 $16.071ms$。
    - 与数据库管理系统（耗时 $19549ms$）对比，TinyDB的用时仅为 $1331.74ms$。性能差异较大的原因可能有以下几点：
        1. DBMS需要对命令进行编译和优化后才可以执行。TinyDB没有实现SQL命令解释器，因此采用字符串匹配确定操作类型。所以该部分耗时DBMS会显著高于TinyDB。
        2. Postgresql是基于硬盘的数据库管理系统，DBMS在插入数据时，会创建多个文件维护该数据库的信息，而不只有数据条。TinyDB是基于内存的简易数据库管理系统，只会将索引和数据保存在硬盘中，且读写线程分离，因此主线程没有文件写耗时。
        3. 数据库客户端与DBMS通过本地回环地址相互通信。虽然通信流量不经过物理网卡，不存在网络延迟。但是网络协议栈的处理是通过操作系统内核完成，在批量执行命令时，系统调用开销较大。而TinyDB基本上在用户空间内完成操作，因此该部分耗时显著低于DBMS。
        4. 由于该数据集不存在可索引的键值，DBMS不会对其建立索引，只是顺序存储。TinyDB的索引树构建耗时是程序主要瓶颈，在此部分会比DBMS慢。
- `DELETE`
    - 测试时的由文件重定向至标准输入，程序中在完成数据读取后开启采样分析，即仅收集删除操作时数据。相关文件配置如下：

```
.load
DELETE FROM flights WHERE departure LIKE "DEN";
DELETE FROM flights WHERE arrival LIKE "DEN";
```

```
#include <ittnotify.h>
int command_cnt;
```

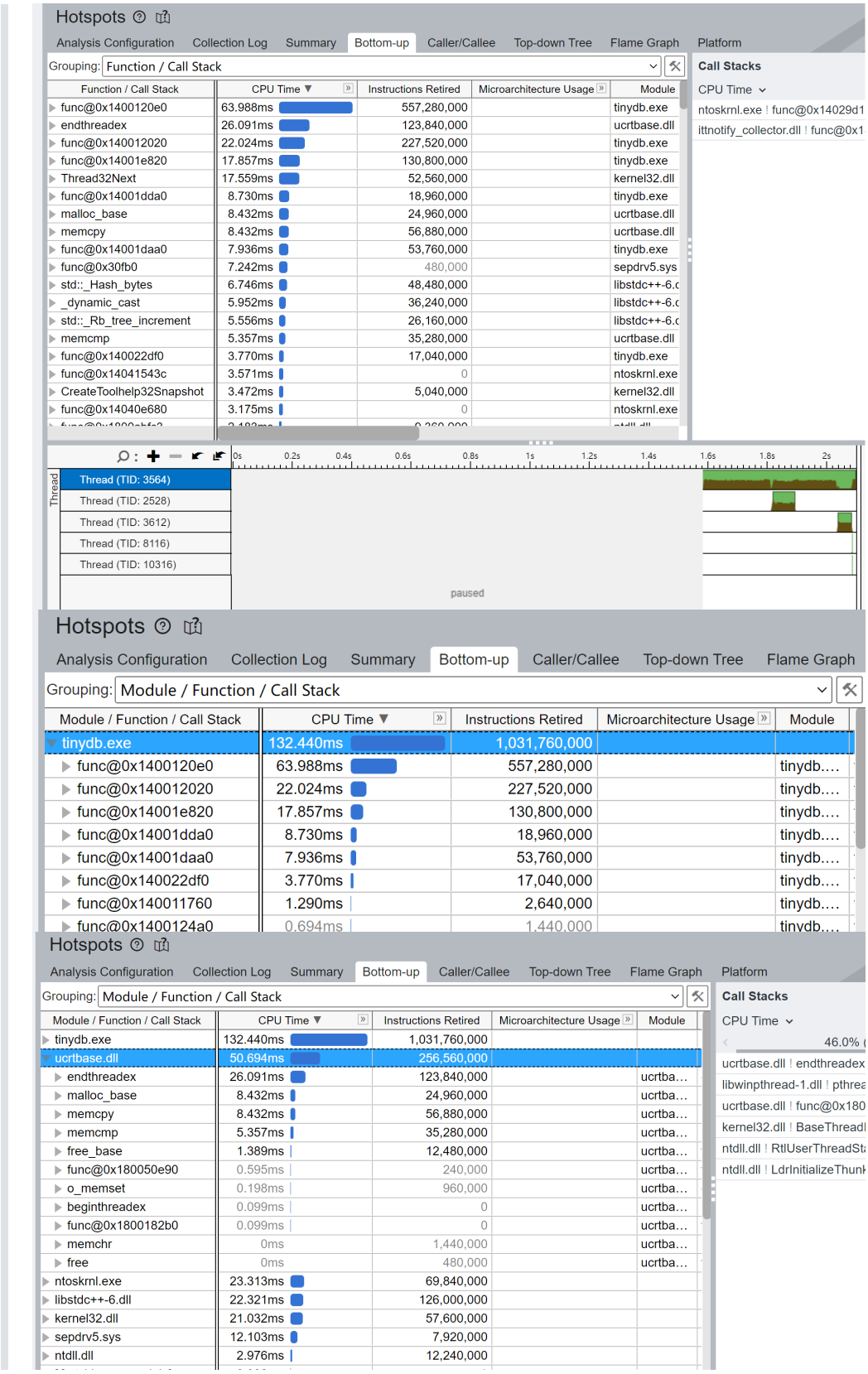
```
...
int main()
{
    while (true)
    {
        repl.start();
        std::string now_input = repl.get_now_input();

        command_cnt++;
        auto timer_begin =
std::chrono::high_resolution_clock::now();
        if (command_cnt == 2)
            __itt_resume();
        ...

        auto timer_end = std::chrono::high_resolution_clock::now();
        auto duration =
std::chrono::duration_cast<std::chrono::microseconds>(timer_end -
timer_begin);
        std::cout << "Time measured: " << duration.count() * 1e-3 <<
"ms." << std::endl;
        std::cout << "-------------" << std::endl;
        if (command_cnt == 3)
            break;
    }
    ...
}
```

- 测试结果如下：

```
-------------
Task finished
Load 74350 of 74350 lines.
Table name: flights
departure arrival day_op dep_time carrier airline flightnum duration aircraft
74349 rows in set.
Time measured: 1292.51ms.
-------------
TinyDB > Hit leafs: 186
Table name: flights
departure arrival day_op dep_time carrier airline flightnum duration aircraft
71624 rows in set.
Time measured: 73.69ms.
-------------
TinyDB > Hit leafs: 113
Table name: flights
departure arrival day_op dep_time carrier airline flightnum duration aircraft
68773 rows in set.
Time measured: 64.623ms.
-------------
```



- TinyDB中耗时前五的函数分别是：`IndexTree::merge_map(Internal)`，`IndexTree::merge_map(Leaf)`，`HashTable::create_map()`，和`std::unordered_map<std::string_view, std::bitset<PRIME>>`对`std::string_view`计算散列值的函数以及`HashTable::get_hash()`。注意到ucrtbase.dll中线程终止函数`endthreadex()`和kernel32.dll中获取线程信息函数`Thread32Next()`的耗时较高。TinyDB主程序耗时占比仅为 49.42%，此时多线程和其他系统调用的耗时占比较高。
- 与数据库管理系统（耗时 $6.083ms$）相比，TinyDB的耗时为 $138.313ms$，显著高于DBMS，是它的22倍多。性能差异较大的原因可能有以下几点：

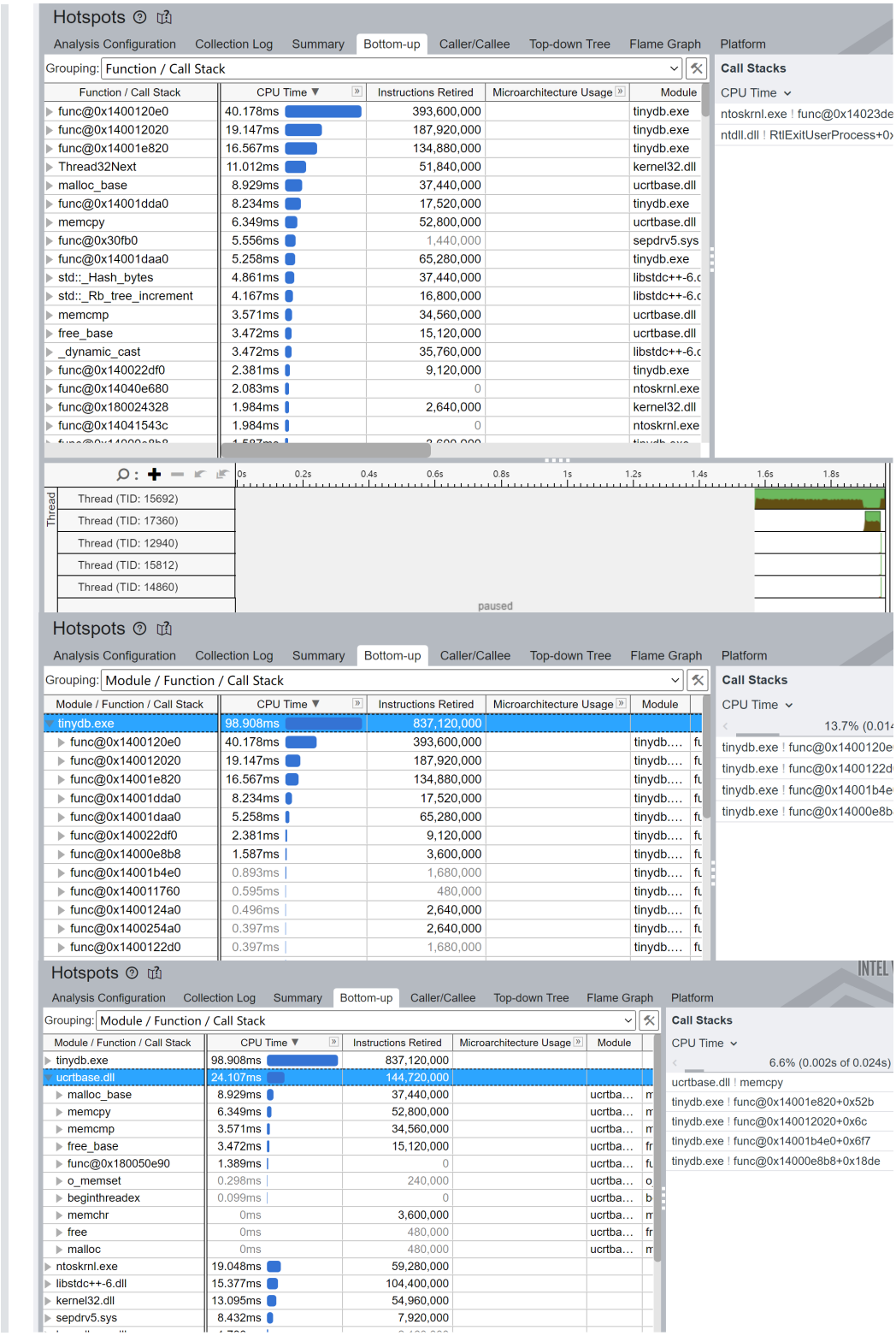
1. DBMS的删除操作不涉及索引更改。该数据集没有设定可索引键值，因此DBMS的删除是采用顺序扫描节点，直接对数据条目进行操作。TinyDB由于建立了可观的索引树，因此有 91% 以上的耗时用于更新索引表。假设去除这部分耗时，则相比DBMS的时间会降低到12倍左右。
2. DBMS的删除操作仅对数据条目做标记。由官方文档可知，在Notes中提到，删除操作仅会标记被选中的数据条目为删除状态，之后在官方文档可知，被标记的行会在后台的Routine Vacuuming过程中从硬盘上清除。因此DBMS的删除仅需要遍历所有数据并做标记，无其他开销。TinyDB由于文件读写线程与主线程分离，需要进行线程中信息交换。因此TinyDB主程序耗时占比不到一半，有相当大开销是线程相关操作。需要说明的是，TinyDB采用索引树结构是为了将不同数据分片，保存为小文件。在做类似删除操作时，不需要对整个数据集的文件做覆写。该设计的初衷是减少文件读写耗时，但在没有后台服务进程的前提下，不可避免会出现进程调度的开销。若继续开发作为实际产品，与进程有关开销应当不计入主程序。因此，若仅计算主程序的部分，在考虑不添加索引时，主程序的耗时推测约为 $11ms$。该耗时与DBMS是相同数量级，但仍然较高。

- `UPDATE`
  - 测试时的由文件重定向至标准输入，程序中在完成数据读取后开启采样分析，即仅收集更新操作时数据。相关文件配置如下，主程序设置与删除操作测试类似：

    ```
    .load
    UPDATE flights SET departure="BBC" WHERE departure LIKE "DEN";
    ```

  - 测试结果如下：

    ```
    -------------
    Task finished
    Load 74350 of 74350 lines.
    Table name: flights
    departure arrival day_op dep_time carrier airline flightnum duration aircraft
    74349 rows in set.
    Time measured: 1409.46ms.
    -------------
    TinyDB > Hit leafs: 186
    Table name: flights
    departure arrival day_op dep_time carrier airline flightnum duration aircraft
    74349 rows in set.
    Time measured: 115.665ms.
    -------------
    ```



- TinyDB中耗时前五的函数分别是：`IndexTree::merge_map(Internal)`，`IndexTree::merge_map(Leaf)`，`HashTable::create_map()`，和`std::unordered_map<std::string_view, std::bitset<PRIME>>`对`std::string_view`计算散列值的函数以及`HashTable::get_hash()`。与删除操作的耗时排行前五完全一致的原因是，在TinyDB中的更新操作是先找到符合条件的数据节点，之后先调用删除函数，再调用插入函数。之所以不能直接在叶节点更新，是因为需要维护索引树的结构及数据合法。由于此时有较高的内存读写请求，因此ucrtbase.dll内有较多内存操作相关耗时。在kernel32.dll中，`Thread32Next()`耗时占比 84.9%，同样存在一定的线程交换耗时。
- 与数据库管理系统（$9.790ms$）相比，TinyDB的耗时为 $115.665ms$，也显著高于DBMS。与删除操作进行对比，性能差异可能有以下几点：

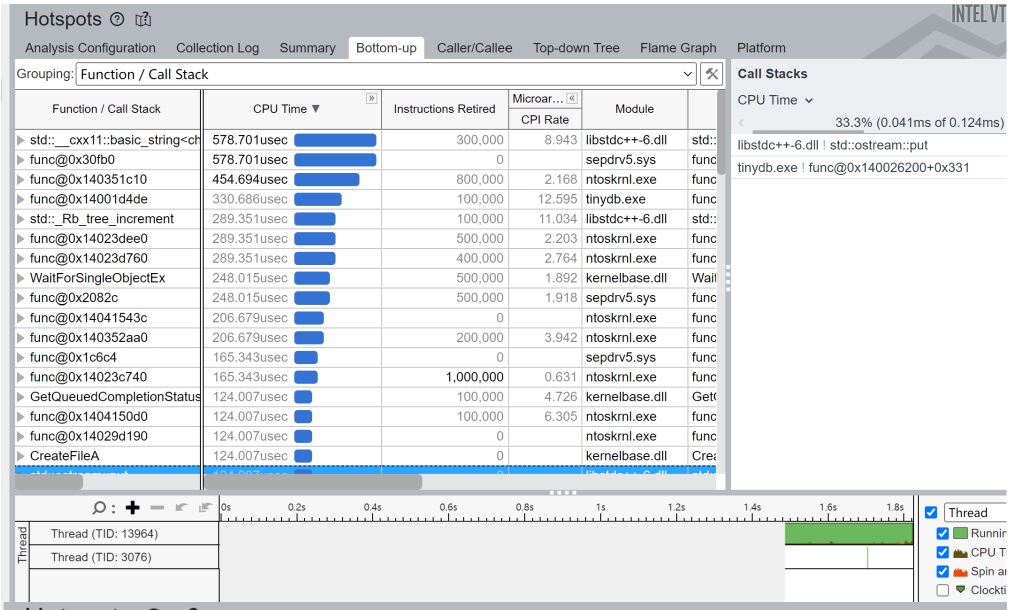
1. DBMS使用多版本并发控制机制。根据官方文档，在其中使用了可序列化快照隔离（Serializable Snapshot Isolation）机制。更新操作会被认为一个事务，事务具有时间轴上标记和插入删除命令标记（xid与cid）。通过检查事务信息，可以发现更新行操作是由删除操作和插入操作组成，这两个操作在一个事务中被提交。因此DBMS的耗时有所增加。
2. TinyDB耗时与数据集大小有关。TinyDB内也是先删除后插入，在进行修改后，索引树有所减小。因此 `IndexTree::merge_map(Internal)` 与 `IndexTree::merge_map(leaf)` 的耗时比率有所减小。此外，线程调度耗时减少可能是由于操作系统调度波动。因为两种操作对线程调度函数都只调用一次。此处就展示了性能测量误差。

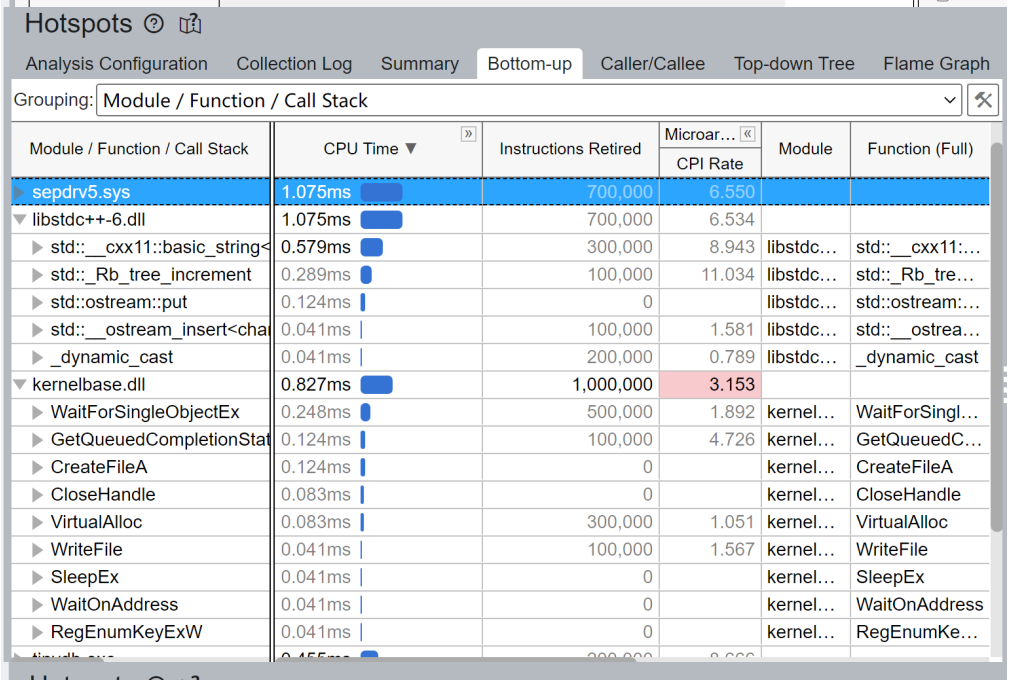
- `SELECT`
  - 测试时的由文件重定向至标准输入，程序中在完成数据读取后开启采样分析，即仅收集查询操作时数据。注意，由于该项测试耗时极短，因此采样间隔为硬件所支持最高精度（$0.01ms$），额外开销级别是第 9 级最高开销。相关文件配置如下，主程序设置与更新操作测试类似：

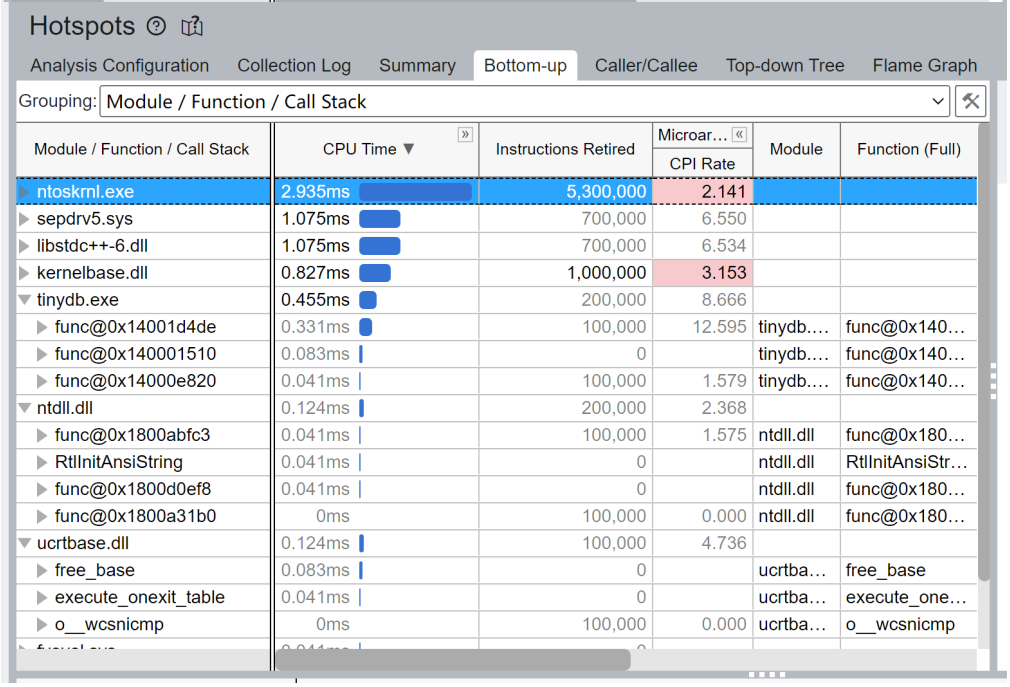
    ```
    .load
    SELECT * FROM flights WHERE departure LIKE "DEN";
    SELECT * FROM flights WHERE arrival LIKE "DEN";
    ```

  - 测试结果如下：

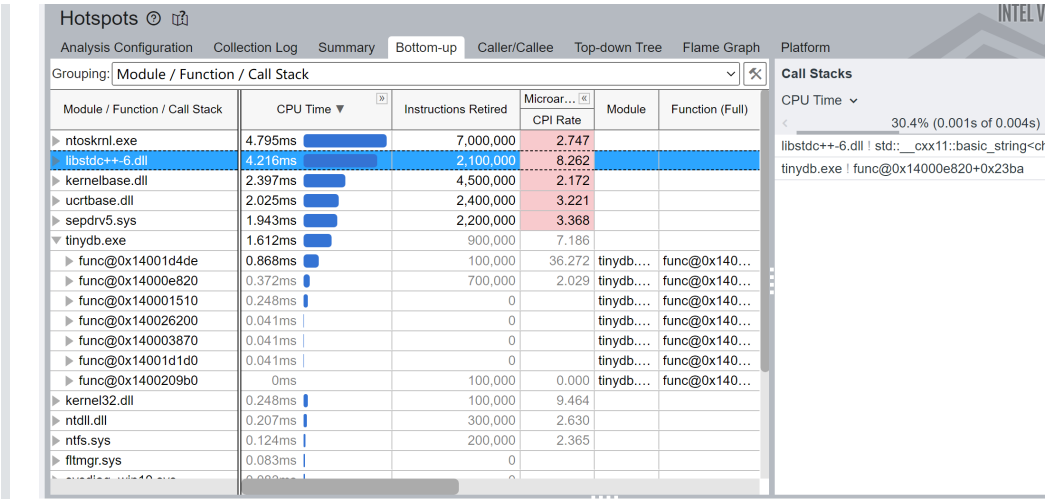
    ```
    -------------
    Task finished
    Load 74350 of 74350 lines.
    Table name: flights
    departure arrival day_op dep_time carrier airline flightnum duration aircraft
    74349 rows in set.
    Time measured: 1329.71 ms.
    -------------
    TinyDB > Hit leafs: 186
    2725
    Time measured: 0.903 ms.
    -------------
    TinyDB > Hit leafs: 186
    2851
    Time measured: 0.758 ms.
    -------------
    ```


Hotspots
INTEL VT
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Platform
Grouping: Function / Call Stack
Function / Call Stack
CPU Time
Instructions Retired
Microar…
CPI Rate
Module
std::__cxx11::basic_string<ch 578.701usec 300,000 8.943 libstdc++-6.dll
func@0x30fb0 578.701usec 0 sepdrv5.sys
func@0x140351c10 454.694usec 800,000 2.168 ntoskrnl.exe
func@0x14001d4de 330.686usec 100,000 12.595 tinydb.exe
std::_Rb_tree_increment 289.351usec 100,000 11.034 libstdc++-6.dll
func@0x14023dee0 289.351usec 500,000 2.203 ntoskrnl.exe
func@0x14023d760 289.351usec 400,000 2.764 ntoskrnl.exe
WaitForSingleObjectEx 248.015usec 500,000 1.892 kernelbase.dll
func@0x2082c 248.015usec 500,000 1.918 sepdrv5.sys
func@0x14041543c 206.679usec 0 ntoskrnl.exe
func@0x140352aa0 206.679usec 200,000 3.942 ntoskrnl.exe
func@0x1c6c4 165.343usec 0 sepdrv5.sys
func@0x14023c740 165.343usec 1,000,000 0.631 ntoskrnl.exe
GetQueuedCompletionStatus 124.007usec 100,000 4.726 kernelbase.dll
func@0x1404150d0 124.007usec 100,000 6.305 ntoskrnl.exe
func@0x14029d190 124.007usec 0 ntoskrnl.exe
CreateFileA 124.007usec 0 kernelbase.dll
Call Stacks
CPU Time
33.3% (0.041ms of 0.124ms)
libstdc++-6.dll ! std::ostream::put
tinydb.exe ! func@0x140026200+0x331
Thread
Thread (TID: 13964)
Thread (TID: 3076)


Hotspots
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Grouping: Module / Function / Call Stack
Module / Function / Call Stack
CPU Time
Instructions Retired
Microar…
CPI Rate
Module
Function (Full)
sepdrv5.sys 1.075ms 700,000 6.550
libstdc++-6.dll 1.075ms 700,000 6.534
std::__cxx11::basic_string< 0.579ms 300,000 8.943 libstdc… std::__cxx11:…
std::_Rb_tree_increment 0.289ms 100,000 11.034 libstdc… std::_Rb_tre…
std::ostream::put 0.124ms 0 libstdc… std::ostream:…
std::__ostream_insert<cha 0.041ms 100,000 1.581 libstdc… std::__ostrea…
_dynamic_cast 0.041ms 200,000 0.789 libstdc… _dynamic_cast
kernelbase.dll 0.827ms 1,000,000 3.153
WaitForSingleObjectEx 0.248ms 500,000 1.892 kernel… WaitForSingl…
GetQueuedCompletionStat 0.124ms 100,000 4.726 kernel… GetQueuedC…
CreateFileA 0.124ms 0 kernel… CreateFileA
CloseHandle 0.083ms 0 kernel… CloseHandle
VirtualAlloc 0.083ms 300,000 1.051 kernel… VirtualAlloc
WriteFile 0.041ms 100,000 1.567 kernel… WriteFile
SleepEx 0.041ms 0 kernel… SleepEx
WaitOnAddress 0.041ms 0 kernel… WaitOnAddress
RegEnumKeyExW 0.041ms 0 kernel… RegEnumKe…


Hotspots
Analysis Configuration
Collection Log
Summary
Bottom-up
Caller/Callee
Top-down Tree
Flame Graph
Grouping: Module / Function / Call Stack
Module / Function / Call Stack
CPU Time
Instructions Retired
Microar…
CPI Rate
Module
Function (Full)
ntoskrnl.exe 2.935ms 5,300,000 2.141
sepdrv5.sys 1.075ms 700,000 6.550
libstdc++-6.dll 1.075ms 700,000 6.534
kernelbase.dll 0.827ms 1,000,000 3.153
tinydb.exe 0.455ms 200,000 8.666
func@0x14001d4de 0.331ms 100,000 12.595 tinydb.… func@0x140…
func@0x140001510 0.083ms 0 tinydb.… func@0x140…
func@0x14000e820 0.041ms 100,000 1.579 tinydb.… func@0x140…
ntdll.dll 0.124ms 200,000 2.368
func@0x1800abfc3 0.041ms 100,000 1.575 ntdll.dll func@0x180…
RtlInitAnsiString 0.041ms 0 ntdll.dll RtlInitAnsiStr…
func@0x1800d0ef8 0.041ms 0 ntdll.dll func@0x180…
func@0x1800a31b0 0ms 100,000 0.000 ntdll.dll func@0x180…
ucrtbase.dll 0.124ms 100,000 4.736
free_base 0.083ms 0 ucrtba… free_base
execute_onexit_table 0.041ms 0 ucrtba… execute_one…
o__wcsnicmp 0ms 100,000 0.000 ucrtba… o__wcsnicmp

- 由于总耗时过短，采样数据出现一定误差。TinyDB位于 `0x14001d4de` 的指令并不是函数入口，但是在 `IndexTree::fuzzy_find_leaf()` 函数内；位于 `0x140001510` 的函数是 `REPL` 类的析构函数；第三个函数是 `Runtime::run_statement()`。整体耗时最高的函数是 `std::string` 的 `find()` 函数。ntoskrnl.exe是系统内核模块，sepdrv5.sys是Intel VTune的驱动。该采样结果不足以反映TinyDB内瓶颈所在，因此将执行 10 次查询操作后的程序再采样，得到以下结果：



Module / Function / Call Stack	CPU Time ▼	Instructions Retired	Microar… CPI Rate	Module	Function (Full)
ntoskrnl.exe	4.795ms	7,000,000	2.747		
libstdc++-6.dll	4.216ms	2,100,000	8.262		
kernelbase.dll	2.397ms	4,500,000	2.172		
ucrtbase.dll	2.025ms	2,400,000	3.221		
sepdrv5.sys	1.943ms	2,200,000	3.368		
tinydb.exe	1.612ms	900,000	7.186		
func@0x14001d4de	0.868ms	100,000	36.272	tinydb.…	func@0x140…
func@0x14000e820	0.372ms	700,000	2.029	tinydb.…	func@0x140…
func@0x140001510	0.248ms	0		tinydb.…	func@0x140…
func@0x140026200	0.041ms	0		tinydb.…	func@0x140…
func@0x140003870	0.041ms	0		tinydb.…	func@0x140…
func@0x14001d1d0	0.041ms	0		tinydb.…	func@0x140…
func@0x1400209b0	0ms	100,000	0.000	tinydb.…	func@0x140…
kernel32.dll	0.248ms	100,000	9.464		
ntdll.dll	0.207ms	300,000	2.630		
ntfs.sys	0.124ms	200,000	2.365		
fltmgr.sys	0.083ms	0			

- 增加操作再测量后，TInyDB内函数耗时较为准确，与上一个相比二三位互换。
- 与数据库管理系统（$8.436ms$）相比，TinyDB的耗时约为 $1.661ms$，显著低于DBMS，性能差异可能有以下原因：
    1. TinyDB索引效果优秀。索引后命中叶节点数量占所有节点约 15%，实际数据为 4% 左右。存在一定误差的原因是在对出发和到达的索引中，两个字段会被放到同一个索引表内（即总索引表），而飞机航线不可能存在单程飞本场的航班。因此两个数据段完全不重合，会造成一定误差。
    2. DBMS需要扫描所有数据并做过滤。由于该数据集没有索引键值，DBMS使用顺序扫描节点过滤每一条数据。这导致在删除、更新、查询三个操作中，DBMS的效率几乎一样。因为查询操作不涉及数据修改，因此耗时较另两种较低。
    3. TinyDB的时间测量不精确。由于查询操作的效率足够快，导致连自定义类的默认析构函数开销都能进入性能分析前列。理论上的性能瓶颈应该是 `IndexTree::fuzzy_find_leaf()` 函数有显著峰值，其余相关函数跟在后面，且 `std::string` 的 `find()` 函数也应该有显著开销。这需要有更大数量级的数据才能精确刻画TinyDB的查询效率。
- 整体而言，TinyDB在读取 $4.82MB$ 的文件后，内存占用约为 $35MB$。插入、删除、修改、查询四个基本数据库操作能够完成。与DBMS相比，TinyDB的优势是对数据集有特殊设计底层结构，但TinyDB无法做到与DBMS一致的通用性和高效性。

5、实验结论

我在进行了两个数据库管理系统的性能测试实验后，对DBMS有更深刻的理解，有以下结论：

1. 数据库管理系统为了减少输入输出瓶颈的影响，在缓存结构与内存使用上有细致且高效的算法设计。
2. 数据库管理系统有广泛的通用性和稳定的性能，但需要使用合适的SQL语句执行命令。
3. 数据库管理系统十分复杂，具有很高的工程量。
4. 数据表中建立索引能有效提高查询效率，但会降低修改效率。
5. 数据表内不同的索引方式各有优劣，需要根据数据集设定。

6. 数据表的结构会很大程度影响数据库管理系统的效率，需要针对特定数据集设计表结构。
7. TinyDB在特定的数据集上表现出色，是因为有针对性的优化，通过性能分析和对比我能学习如何优化数据库系统。
8. 对该实验的个人想法：
    - 这种对比可以让我更好理解数据库管理系统的工作原理、锻炼中型项目的编程能力、学习项目管理周期的技术点。
    - 重要的是，这能激发我对数据库原理学习的兴趣。我希望能更深刻了解如何设计数据表，以及如何搭建数据库管理系统。
    - 开放性的实验要求，可以有更广的探索空间。这相比于被规则限制死的项目更有趣味。我希望能学习其他优秀报告的性能测试，了解其他同学的关注点。