

软件学院

数据库报告：实验二算法实现

|  |  |
| --- | --- |
| 选课编号： | 113007 |
| 分组编号： | 24 |
| 姓名： | 程鹏宇 |
| 学号： | 3124336012 |
| 班级： | 1班 |
| 指导老师： | 白秀秀 |
| 完成日期： | 2024年11月31号 |

**目 录**

[1. 索引 1](#_Toc27192)

[1.1线性散列索引的构建、检索、插入、删除的代码实现与测试 1](#_Toc17645)

[2. 并发控制 3](#_Toc30203)

[2.1冲突可串行性判别算法 3](#_Toc18301)

[2.2 基于时间戳的并发控制算法 5](#_Toc27749)

[3．完成Paxos协议的代码实现与测试 8](#_Toc6930)

# 索引

## 1.1线性散列索引的构建、检索、插入、删除的代码实现与测试

class LinearHash:

    def \_\_init\_\_(self, initial\_buckets=8):

        self.buckets = [{} for \_ in range(initial\_buckets)]

        self.bucket\_count = initial\_buckets

        self.split\_index = 0

        self.threshold = 0.75  # 装填因子阈值

    def \_hash(self, key, level=None):

        if level is None:

            level = self.bucket\_count

        return hash(key) % level

    def \_rehash(self):

        # 当前桶满了，需要分裂

        bucket\_to\_split = self.buckets[self.split\_index]

        new\_bucket = {}

        # 将要分裂的桶中的元素重新分配

        for key, value in bucket\_to\_split.items():

            new\_index = self.\_hash(key, self.bucket\_count \* 2)

            if new\_index < self.bucket\_count:

                del self.buckets[new\_index][key]

                self.buckets[new\_index][key] = value

            else:

                new\_bucket[key] = value

        # 更新桶数组

        self.buckets.append(new\_bucket)

        self.bucket\_count \*= 2

        self.split\_index += 1

        if self.split\_index >= len(self.buckets):

            self.split\_index = 0

    def insert(self, key, value):

        index = self.\_hash(key)

        bucket = self.buckets[index]

        # 如果桶已经存在该键，则更新值

        if key in bucket:

            bucket[key] = value

        else:

            bucket[key] = value

            # 检查是否需要重哈希

            if sum(len(bucket) for bucket in self.buckets) / self.bucket\_count > self.threshold:

                self.\_rehash()

    def retrieve(self, key):

        index = self.\_hash(key)

        bucket = self.buckets[index]

        return bucket.get(key, None)

    def delete(self, key):

        index = self.\_hash(key)

        bucket = self.buckets[index]

        if key in bucket:

            del bucket[key]

    def \_\_str\_\_(self):

        return '\n'.join(f'Bucket {i}: {bucket}' for i, bucket in enumerate(self.buckets))

# 示例使用

if \_\_name\_\_ == '\_\_main\_\_':

    lh = LinearHash()

    lh.insert('a', 1)

    lh.insert('b', 2)

    lh.insert('c', 3)

    print(lh.retrieve('a'))

    lh.delete('b')

    print(lh.retrieve('b'))

    print(lh)

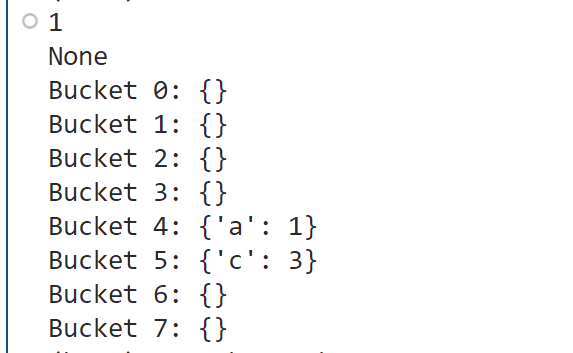


图 1 实验1测试结果

# 并发控制

## 2.1冲突可串行性判别算法

from collections import defaultdict, deque

def build\_conflict\_graph(schedule):

    # 构建冲突图

    conflict\_graph = defaultdict(set)

    operations = []

    # 解析调度

    for operation in schedule:

        action, transaction, item = operation

        operations.append((action, transaction, item))

    # 记录每个事务对每个数据项的最后一次操作

    last\_operations = defaultdict(lambda: defaultdict(tuple))

    for idx, (action, transaction, item) in enumerate(operations):

        if action == 'W':

            # 写操作

            if item in last\_operations:

                for t, op\_idx in last\_operations[item].items():

                    if t != transaction:

                        conflict\_graph[t].add(transaction)

                        conflict\_graph[transaction].add(t)

            last\_operations[item][transaction] = (action, idx)

        elif action == 'R':

            # 读操作

            if item in last\_operations:

                for t, (last\_action, op\_idx) in last\_operations[item].items():

                    if t != transaction and last\_action == 'W':

                        conflict\_graph[t].add(transaction)

                        conflict\_graph[transaction].add(t)

            last\_operations[item][transaction] = (action, idx)

    return conflict\_graph

def is\_cyclic(conflict\_graph):

    # 检测冲突图中是否存在环

    visited = set()

    recursion\_stack = set()

    def dfs(node):

        visited.add(node)

        recursion\_stack.add(node)

        for neighbor in conflict\_graph[node]:

            if neighbor not in visited:

                if dfs(neighbor):

                    return True

            elif neighbor in recursion\_stack:

                return True

        recursion\_stack.remove(node)

        return False

    for node in conflict\_graph:

        if node not in visited:

            if dfs(node):

                return True

    return False

def is\_conflict\_serializable(schedule):

    conflict\_graph = build\_conflict\_graph(schedule)

    return not is\_cyclic(conflict\_graph)

# 示例使用

if \_\_name\_\_ == '\_\_main\_\_':

    schedule = [

        ('R', 'T1', 'B'),

        ('W', 'T1', 'B'),

        ('R', 'T2', 'A'),

        ('W', 'T2', 'B'),

        ('R', 'T3', 'B'),

        ('W', 'T3', 'A')

    ]

    print(schedule)

    if is\_conflict\_serializable(schedule):

        print("The schedule is conflict serializable.")

    else:

        print("The schedule is not conflict serializable.")

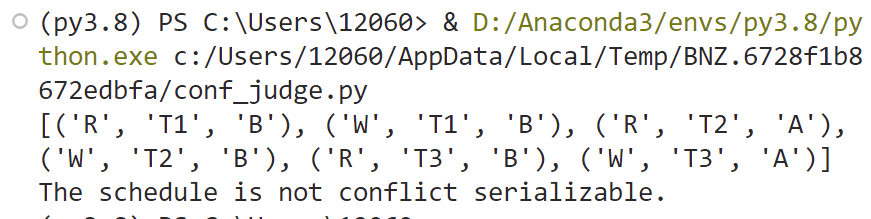


图 2 实验2.1测试结果

## 2.2 基于时间戳的并发控制算法

from collections import defaultdict

class TimestampBasedConcurrencyControl:

    def \_\_init\_\_(self):

        self.data\_items = {}

        self.write\_timestamps = {}

        self.read\_locks = defaultdict(set)

        self.write\_locks = {}

        self.transactions = {}

        self.next\_timestamp = 0

    def start\_transaction(self, transaction\_id):

        self.transactions[transaction\_id] = self.next\_timestamp

        self.next\_timestamp += 1

    def read(self, transaction\_id, data\_item):

        ts = self.transactions[transaction\_id]

        if data\_item in self.write\_locks and self.write\_locks[data\_item] != transaction\_id:

            print(f"Transaction {transaction\_id} aborted due to write lock on {data\_item}")

            return None

        if data\_item in self.write\_timestamps and self.write\_timestamps[data\_item] >= ts:

            print(f"Transaction {transaction\_id} aborted due to write timestamp on {data\_item}")

            return None

        self.read\_locks[data\_item].add(transaction\_id)

        return self.data\_items.get(data\_item, None)

    def write(self, transaction\_id, data\_item, value):

        ts = self.transactions[transaction\_id]

        if data\_item in self.write\_locks and self.write\_locks[data\_item] != transaction\_id:

            print(f"Transaction {transaction\_id} aborted due to write lock on {data\_item}")

            return False

        if data\_item in self.write\_timestamps and self.write\_timestamps[data\_item] >= ts:

            print(f"Transaction {transaction\_id} aborted due to write timestamp on {data\_item}")

            return False

        if data\_item in self.read\_locks and any(t != transaction\_id for t in self.read\_locks[data\_item]):

            print(f"Transaction {transaction\_id} aborted due to read lock on {data\_item}")

            return False

        self.write\_locks[data\_item] = transaction\_id

        self.data\_items[data\_item] = value

        self.write\_timestamps[data\_item] = ts

        return True

    def commit(self, transaction\_id):

        for data\_item in list(self.read\_locks.keys()):

            if transaction\_id in self.read\_locks[data\_item]:

                self.read\_locks[data\_item].remove(transaction\_id)

                if not self.read\_locks[data\_item]:

                    del self.read\_locks[data\_item]

        if transaction\_id in self.write\_locks.values():

            for data\_item, tid in list(self.write\_locks.items()):

                if tid == transaction\_id:

                    del self.write\_locks[data\_item]

        del self.transactions[transaction\_id]

        print(f"Transaction {transaction\_id} committed")

    def abort(self, transaction\_id):

        for data\_item in list(self.read\_locks.keys()):

            if transaction\_id in self.read\_locks[data\_item]:

                self.read\_locks[data\_item].remove(transaction\_id)

                if not self.read\_locks[data\_item]:

                    del self.read\_locks[data\_item]

        if transaction\_id in self.write\_locks.values():

            for data\_item, tid in list(self.write\_locks.items()):

                if tid == transaction\_id:

                    del self.write\_locks[data\_item]

        del self.transactions[transaction\_id]

        print(f"Transaction {transaction\_id} aborted")

# 示例使用

if \_\_name\_\_ == '\_\_main\_\_':

    tcc = TimestampBasedConcurrencyControl()

    tcc.start\_transaction('T1')

    tcc.start\_transaction('T2')

    tcc.start\_transaction('T3')

    print(tcc.read('T1', 'A'))

    tcc.write('T1', 'A', 10)

    print(tcc.read('T2', 'A'))

    tcc.write('T2', 'A', 20)

    print(tcc.read('T3', 'A'))

    tcc.write('T3', 'A', 30)

    tcc.commit('T1')

    tcc.abort('T2')

tcc.abort('T3')

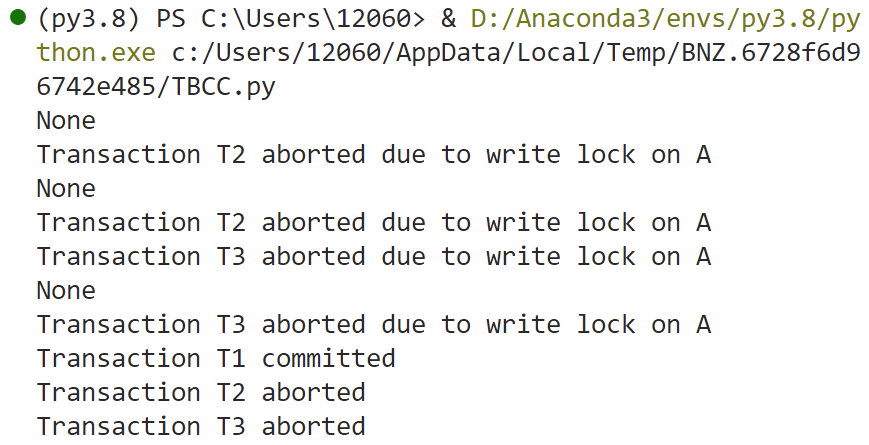


图 3 实验2.2测试结果

# 3．完成Paxos协议的代码实现与测试

import random

import threading

import time

class Proposer:

    def \_\_init\_\_(self, proposer\_id, acceptors, learners):

        self.proposer\_id = proposer\_id

        self.acceptors = acceptors

        self.learners = learners

        self.n = 0

        self.value = None

        self.majority = len(acceptors) // 2 + 1

    def propose(self, value):

        self.value = value

        self.n += 1

        self.prepare()

    def prepare(self):

        print(f"Proposer {self.proposer\_id} sending prepare({self.n})")

        responses = []

        for acceptor in self.acceptors:

            response = acceptor.receive\_prepare(self.n)

            if response:

                responses.append(response)

            if len(responses) >= self.majority:

                break

        if len(responses) >= self.majority:

            self.accept(responses)

    def accept(self, responses):

        max\_n = -1

        max\_value = None

        for n, value in responses:

            if n > max\_n:

                max\_n = n

                max\_value = value

        value = max\_value if max\_value is not None else self.value

        print(f"Proposer {self.proposer\_id} sending accept({self.n}, {value})")

        responses = []

        for acceptor in self.acceptors:

            response = acceptor.receive\_accept(self.n, value)

            if response:

                responses.append(response)

            if len(responses) >= self.majority:

                break

        if len(responses) >= self.majority:

            self.learn(value)

    def learn(self, value):

        print(f"Proposer {self.proposer\_id} learned value {value}")

        for learner in self.learners:

            learner.learn(value)

class Acceptor:

    def \_\_init\_\_(self, acceptor\_id):

        self.acceptor\_id = acceptor\_id

        self.promised\_n = -1

        self.accepted\_n = -1

        self.accepted\_value = None

    def receive\_prepare(self, n):

        if n > self.promised\_n:

            self.promised\_n = n

            print(f"Acceptor {self.acceptor\_id} promised {n}")

            return (self.accepted\_n, self.accepted\_value)

        print(f"Acceptor {self.acceptor\_id} rejected prepare({n})")

        return None

    def receive\_accept(self, n, value):

        if n >= self.promised\_n:

            self.accepted\_n = n

            self.accepted\_value = value

            print(f"Acceptor {self.acceptor\_id} accepted ({n}, {value})")

            return True

        print(f"Acceptor {self.acceptor\_id} rejected accept({n}, {value})")

        return False

class Learner:

    def \_\_init\_\_(self, learner\_id):

        self.learner\_id = learner\_id

    def learn(self, value):

        print(f"Learner {self.learner\_id} learned value {value}")

# 示例使用

if \_\_name\_\_ == '\_\_main\_\_':

    # 创建 Acceptors

    acceptors = [Acceptor(i) for i in range(3)]

    # 创建 Learners

    learners = [Learner(i) for i in range(2)]

    # 创建 Proposers

    proposers = [Proposer(i, acceptors, learners) for i in range(2)]

    # 启动 Proposers

    threads = []

    for proposer in proposers:

        thread = threading.Thread(target=proposer.propose, args=(random.randint(1, 100),))

        threads.append(thread)

        thread.start()

    # 等待所有 Proposers 完成

    for thread in threads:

        thread.join()

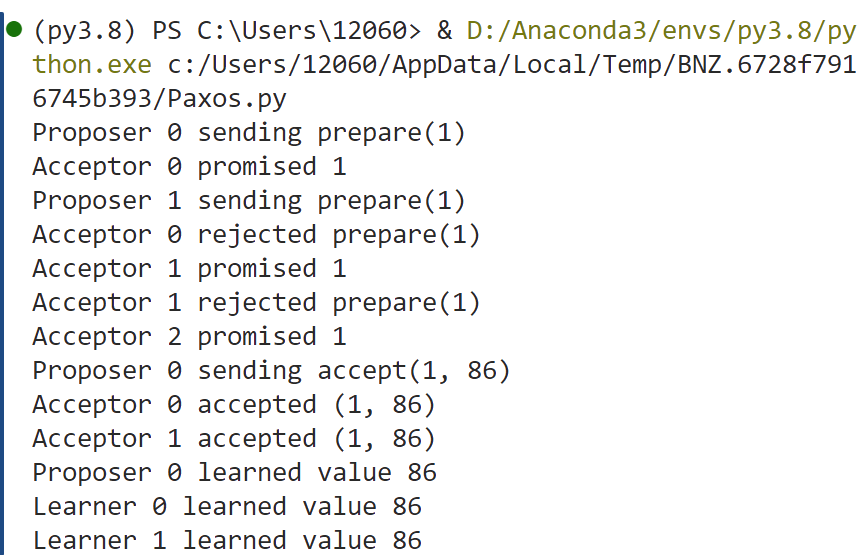


图 4 实验3测试结果