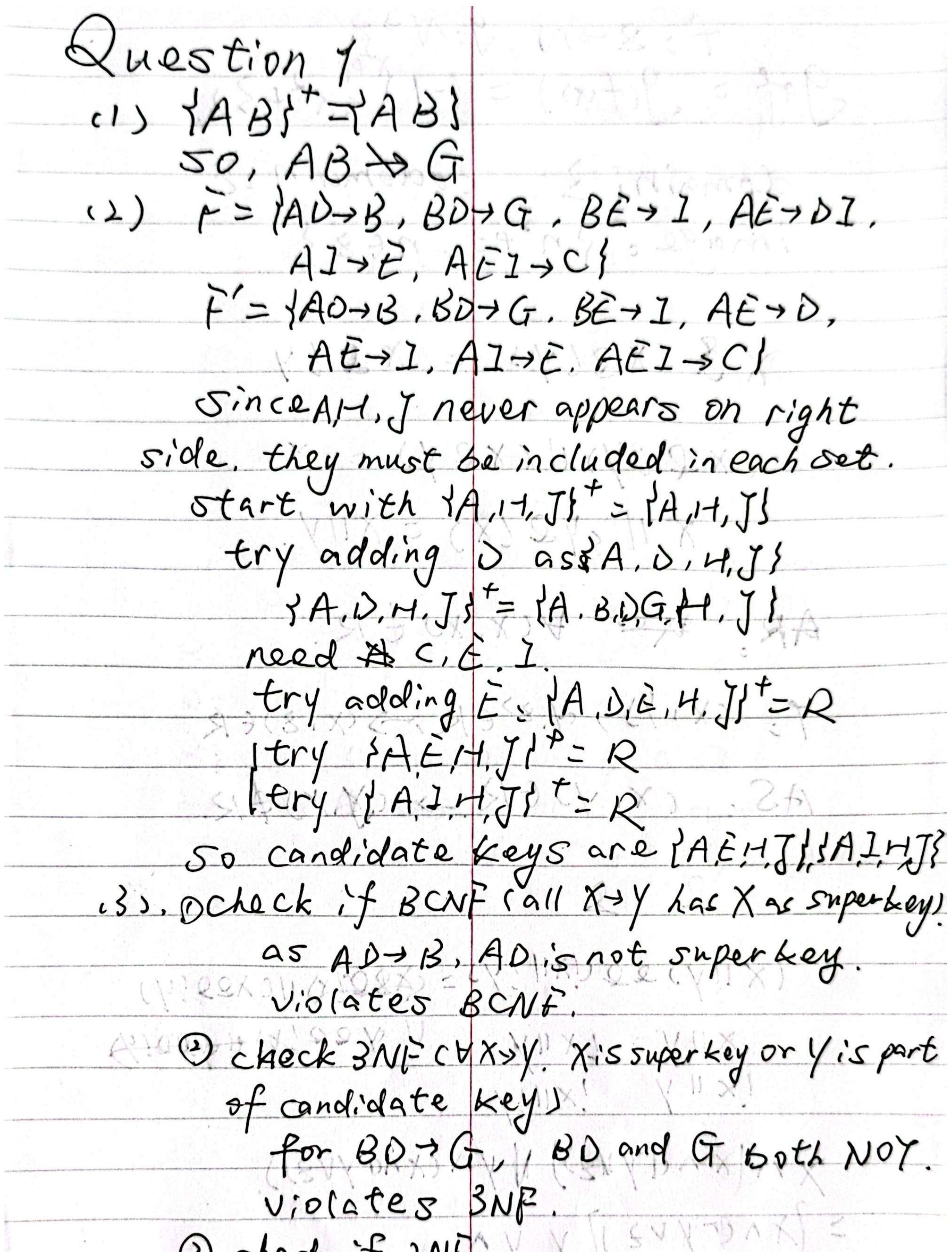


Question 1

(1) $\{AB\}^+ = \{AB\}$

so, $AB \nrightarrow G$

(2) $F = \{AD \to B, BD \to G, BE \to I, AE \to DI, AI \to E, AEI \to C\}$

$F' = \{AD \to B, BD \to G, BE \to I, AE \to D, AE \to I, AI \to E, AEI \to C\}$

Since A, H, J never appears on right side, they must be included in each set.

start with $\{A, H, J\}^+ = \{A, H, J\}$

try adding D as $\{A, D, H, J\}$

$\{A, D, H, J\}^+ = \{A, B, D, G, H, J\}$

need ~~A~~ C, E, I.

try adding E: $\{A, D, E, H, J\}^+ = R$

try $\{A, E, H, J\}^+ = R$

try $\{A, I, H, J\}^+ = R$

So candidate keys are $\{A, E, H, J\}, \{A, I, H, J\}$

(3). ① check if BCNF (all $X \to Y$ has X as superkey).

as $AD \to B$, AD is not superkey.

violates BCNF.

② check 3NF ($\forall X \to Y$. X is superkey or Y is part of candidate key).

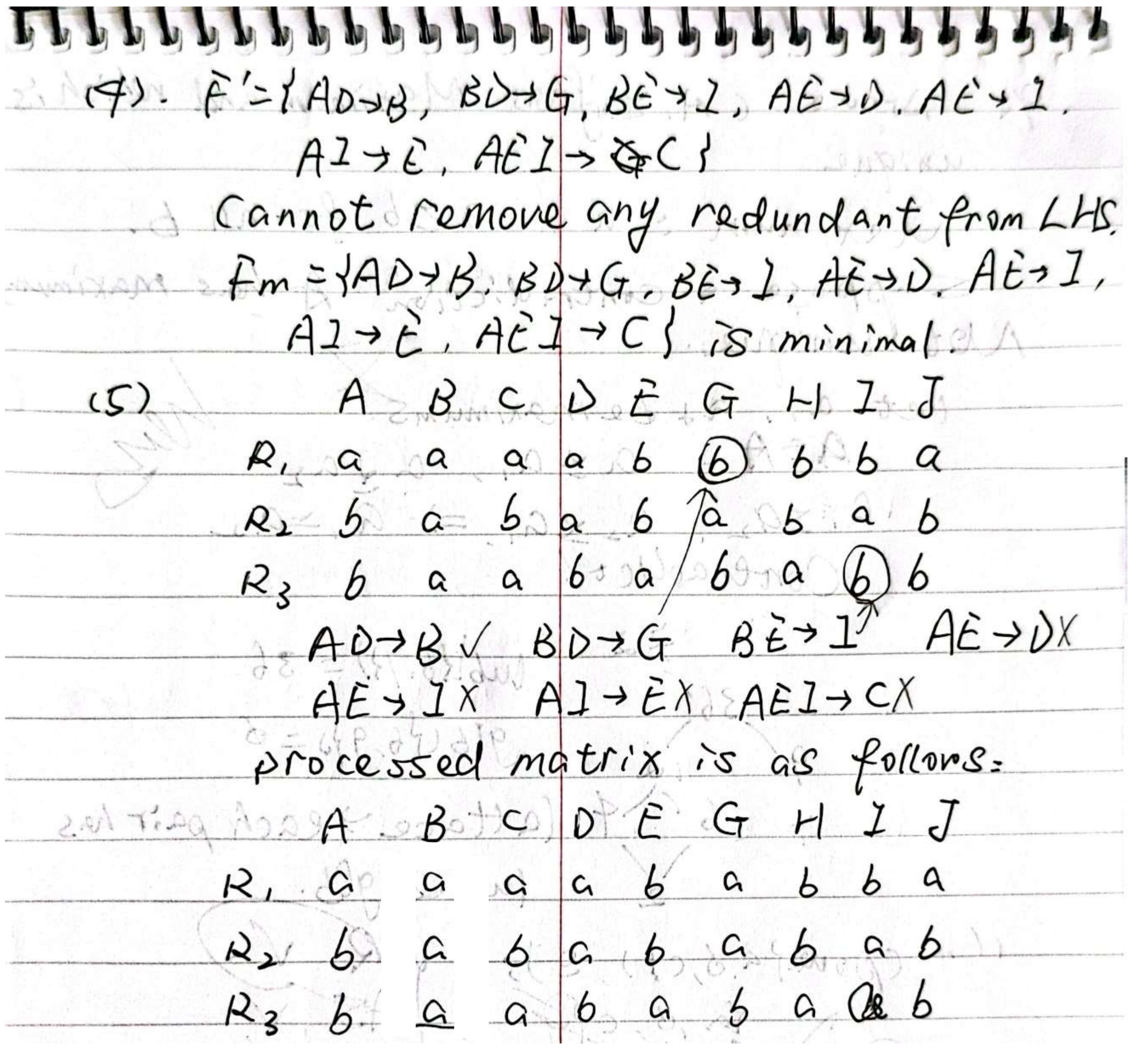
for $BD \to G$, BD and G both NOT.

violates 3NF.

③ check if 2NF:

Since there is partial dependency (AD->B), violates 2NF.

Since atomicity exists, R is 1NF.



(4). F' = {AD→B, BD→G, BE→I, AE→D, AE→I, AI→E, AEI→ ~~G~~C}

Cannot remove any redundant from LHS.

Fm = {AD→B, BD→G, BE→I, AE→D, AE→I, AI→E, AEI→C} is minimal.

(5)

	A	B	C	D	E	G	H	I	J
R1	a	a	a	a	b	b	b	b	a
R2	b	a	b	a	b	a	b	a	b
R3	b	a	a	b	a	b	a	b	b

AD→B ✓ BD→G BE→I AE→D ✗

AE→I ✗ AI→E ✗ AEI→C ✗

processed matrix is as follows:

	A	B	C	D	E	G	H	I	J
R1	a	a	a	a	b	a	b	b	a
R2	b	a	b	a	b	a	b	a	b
R3	b	a	a	b	a	b	a	a	b

Since there's not any row with full of a's

R is NOT lossless join.

6. decompose by $AD \to B$.

$R_1(A, D, B)$. $AD \to B$ $R_2(A, C, D, E, G, H, I, J)$

$BD \to G$, $BE \to I$, $AE \to DI$, $AI \to E$, $AEI \to C$.

decompose R_2 by $BD \to G$.

$R_3(B, D, G)$, $BD \to G$ $R_4(A, C, D, E, H, I, J)$

$BE \to I$, $AE \to DI$, $AI \to E$, $AEI \to C$

decompose R_4 by $BE \to I$

$R_5(B, E, I)$ $BE \to I$, $R_6(A, C, D, E, H, J)$

$AE \to DI$, $AI \to E$, $AEI \to C$.

decompose R_6 by $AE \to DI$

$R_7(A, E, D)$ $AE \to DI$, $R_8(A, C, E, H, J)$

decompose R_8 by $AI \to E$

$R_9(A, I, E)$ ~~$AI \to E$~~ $R_{10}(A, D, I)$ $AEI \to C$

decompose R_{10} by $AEI \to C$

$R_{11}(AE, I, C)$ $R_{12}(A, D, I)$

BCNF decomposition:

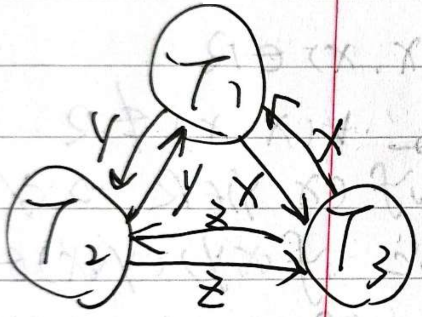
$R_1(A, D, B)$, $R_3(B, D, G)$, $R_5(B, E, I)$

$R_7(A, E, D,)$ $R_9(A, I, E)$, $R_{11}(A, E, I, C)$

$R_{12}(A, D, I)$

Question 2:

(1)



NOT serializable

(2)

Lock Sequence for T1:

1. t1: Acquire shared lock on x, then read x.
2. t4: Acquire shared lock on y, then read y.
3. t8: Upgrade to exclusive lock on x, then write x.
4. t12: Upgrade to exclusive lock on y, then write y.
5. After committing: Release all locks.

Lock Sequence for T2:

1. t3: Acquire shared lock on y, then read y.
2. t5: Acquire shared lock on z, then read z.
3. t9: Upgrade to exclusive lock on y, then write y.
4. t12: Upgrade to exclusive lock on z, then write z.
5. After committing: Release all locks.

Lock Sequence for T3:

1. t2: Acquire shared lock on z, then read z.
2. t6: Acquire shared lock on x, then read x.
3. t7: Upgrade to exclusive lock on z, then write z.
4. t10: Upgrade to exclusive lock on x, then write x.
5. After committing: Release all locks.

(3)

1) Timeline Analysis:

t1: T1 acquires slock(x) and reads x ✓ (successful)

t2: T3 acquires slock(z) and reads z ✓ (successful)

t3: T2 acquires slock(y) and reads y ✓ (successful)

t4: FIRST BLOCKING POINT

T1 tries to acquire slock(y). However, T1 will need xlock(y) later at t12. Cannot acquire just slock(y) due to 2PL upgrade requirement

T2 already holds slock(y). Therefore, T1 must WAIT for T2

t5: SECOND BLOCKING POINT

T2 tries to acquire slock(z)

- However, T2 will need xlock(z) later at t12

- Cannot acquire just slock(z) due to 2PL upgrade requirement

- T3 already holds slock(z)

- Therefore, T2 must WAIT for T3

t6: DEADLOCK COMPLETION

- T3 tries to acquire slock(x)

- However, T3 will need xlock(x) later at t10

- Cannot acquire just slock(x) due to 2PL upgrade requirement

- T1 already holds slock(x)

- Therefore, T3 must WAIT for T1

2) Wait-For Graph Analysis:

- T1 waits for T2 (needs y which T2 holds)

- T2 waits for T3 (needs z which T3 holds)

- T3 waits for T1 (needs x which T1 holds)

This creates a cycle: T1 → T2 → T3 → T1

Therefore, this schedule will result in a deadlock at t6, making it impossible to complete as specified in the original schedule without deadlock resolution mechanisms.

Question 3:

LRU Cache Simulation:

```
Access P1: Miss - Cache: ['P1']

Access P2: Miss - Cache: ['P1', 'P2']

Access P1: Hit - Cache: ['P2', 'P1']

Access P4: Miss - Cache: ['P2', 'P1', 'P4']

Access P3: Miss - Cache: ['P1', 'P4', 'P3']

Access P7: Miss - Cache: ['P4', 'P3', 'P7']

Access P2: Miss - Cache: ['P3', 'P7', 'P2']

Access P1: Miss - Cache: ['P7', 'P2', 'P1']

Access P4: Miss - Cache: ['P2', 'P1', 'P4']

Access P5: Miss - Cache: ['P1', 'P4', 'P5']

Access P8: Miss - Cache: ['P4', 'P5', 'P8']

Access P6: Miss - Cache: ['P5', 'P8', 'P6']

Access P8: Hit - Cache: ['P5', 'P6', 'P8']

Access P2: Miss - Cache: ['P6', 'P8', 'P2']

Access P8: Hit - Cache: ['P6', 'P2', 'P8']

LRU Hit Rate: 20.00%
```

MRU Cache Simulation:

```
Access P1: Miss - Cache: ['P1']

Access P2: Miss - Cache: ['P1', 'P2']
Access P1: Hit - Cache: ['P2', 'P1']

Access P4: Miss - Cache: ['P2', 'P1', 'P4']

Access P3: Miss - Cache: ['P2', 'P1', 'P3']

Access P7: Miss - Cache: ['P2', 'P1', 'P7']

Access P2: Hit - Cache: ['P1', 'P7', 'P2']

Access P1: Hit - Cache: ['P7', 'P2', 'P1']

Access P4: Miss - Cache: ['P7', 'P2', 'P4']

Access P5: Miss - Cache: ['P7', 'P2', 'P5']

Access P8: Miss - Cache: ['P7', 'P2', 'P8']

Access P6: Miss - Cache: ['P7', 'P2', 'P6']

Access P8: Miss - Cache: ['P7', 'P2', 'P8']

Access P2: Hit - Cache: ['P7', 'P8', 'P2']

Access P8: Hit - Cache: ['P7', 'P2', 'P8']

MRU Hit Rate: 33.33%
```

FIFO Cache Simulation:

```
Access P1: Miss - Cache: ['P1']

Access P2: Miss - Cache: ['P1', 'P2']

Access P1: Hit - Cache: ['P1', 'P2']

Access P4: Miss - Cache: ['P1', 'P2', 'P4']

Access P3: Miss - Cache: ['P2', 'P4', 'P3']

Access P7: Miss - Cache: ['P4', 'P3', 'P7']
Access P2: Miss - Cache: ['P3', 'P7', 'P2']

Access P1: Miss - Cache: ['P7', 'P2', 'P1']

Access P4: Miss - Cache: ['P2', 'P1', 'P4']

Access P5: Miss - Cache: ['P1', 'P4', 'P5']

Access P8: Miss - Cache: ['P4', 'P5', 'P8']

Access P6: Miss - Cache: ['P5', 'P8', 'P6']

Access P8: Hit - Cache: ['P5', 'P8', 'P6']

Access P2: Miss - Cache: ['P8', 'P6', 'P2']

Access P8: Hit - Cache: ['P8', 'P6', 'P2']

FIFO Hit Rate: 20.00%
```

Python Code is as follows:

```
from collections import OrderedDict

from typing import List, Any, Optional

import time


class CacheBase:

    def __init__(self, capacity: int):

        self.capacity = capacity

        self.cache: OrderedDict = OrderedDict()

        self.hits = 0

        self.total_accesses = 0
    def get_hit_rate(self) -> float:
        """计算缓存命中率"""
        if self.total_accesses == 0:
            return 0.0
        return self.hits / self.total_accesses * 100

    def _debug_print(self, page: Any, is_hit: bool):
        """打印当前缓存状态，用于调试"""
        status = "Hit" if is_hit else "Miss"
        print(f"Access {page}: {status} - Cache: {list(self.cache.keys())}")

    def access(self, page: Any, debug: bool = False) -> bool:
        """访问一个页面，返回是否命中"""
        raise NotImplementedError("Subclasses must implement access()")

class LRUCache(CacheBase):
    def access(self, page: Any, debug: bool = False) -> bool:
        """
        访问一个页面，如果页面在缓存中，更新其位置；
        如果不在缓存中，添加到缓存，必要时删除最久未使用的页面
        """
        self.total_accesses += 1
        is_hit = False

        if page in self.cache:
            # 页面命中，移动到 OrderedDict 的末尾（表示最近使用）
            self.cache.move_to_end(page)
            self.hits += 1
            is_hit = True
        else:
            # 页面未命中，需要载入
            if len(self.cache) >= self.capacity:
                # 缓存已满，删除最久未使用的页面（OrderedDict 的第一个元素）
                self.cache.popitem(last=False)
            self.cache[page] = True  # 值不重要，我们只关心键

        if debug:
            self._debug_print(page, is_hit)
        return is_hit


class MRUCache(CacheBase):
    def access(self, page: Any, debug: bool = False) -> bool:
        """
        访问一个页面，如果页面在缓存中，更新其位置；
```

```
如果不在缓存中，添加到缓存，必要时删除最近使用的页面
"""
self.total_accesses += 1
is_hit = False

if page in self.cache:
    # 页面命中
    self.cache.move_to_end(page)
    self.hits += 1
    is_hit = True
else:
    # 页面未命中，需要载入
    if len(self.cache) >= self.capacity:
        # 缓存已满，删除最近使用的页面（OrderedDict 的最后一个元素）
        self.cache.popitem(last=True)  # last=True 表示移除最近的元素
    self.cache[page] = True
    self.cache.move_to_end(page)  # 将新页面移到末尾

if debug:
    self._debug_print(page, is_hit)
return is_hit
```

```
class FIFOCache(CacheBase):
    def access(self, page: Any, debug: bool = False) -> bool:
        """
        访问一个页面，如果页面在缓存中，保持其位置不变；
        如果不在缓存中，添加到缓存，必要时删除最先进入的页面
        """
        self.total_accesses += 1
        is_hit = False

        if page in self.cache:
            # 页面命中，位置保持不变
            self.hits += 1
            is_hit = True
        else:
            # 页面未命中，需要载入
            if len(self.cache) >= self.capacity:
                # 缓存已满，删除最先进入的页面
                self.cache.popitem(last=False)
            self.cache[page] = True

        if debug:
            self._debug_print(page, is_hit)
        return is_hit


def test_cache_replacement():
    # 测试序列
    page_sequence = ['P1', 'P2', 'P1', 'P4', 'P3', 'P7', 'P2', 'P1', 'P4', 'P5', 'P8', 'P6', 'P8', 'P2',
'P8']
    buffer_size = 3

    # 创建三种不同的缓存
    lru_cache = LRUCache(buffer_size)
    mru_cache = MRUCache(buffer_size)
    fifo_cache = FIFOCache(buffer_size)

    print("\nLRU Cache Simulation:")
    for page in page_sequence:
        lru_cache.access(page, debug=True)
    print(f"LRU Hit Rate: {lru_cache.get_hit_rate():.2f}%")

    print("\nMRU Cache Simulation:")
    for page in page_sequence:
        mru_cache.access(page, debug=True)
    print(f"MRU Hit Rate: {mru_cache.get_hit_rate():.2f}%")
    print("\nFIFO Cache Simulation:")
    for page in page_sequence:
        fifo_cache.access(page, debug=True)
    print(f"FIFO Hit Rate: {fifo_cache.get_hit_rate():.2f}%")


if __name__ == "__main__":
    test_cache_replacement()
```