Implementation of DBMS
Exercise Sheet 9-10
K.m., WS 2022 / 2023

Similar Questions for Each Exercise:

Exercise 1: File Hashing into Buckets

1. **New Question 1:**
   Suppose we have a file of 500,000 records that we want to hash into a table with 500 buckets. Each block can hold 200 records, and we wish to keep blocks as full as possible, but not allow two buckets to share a block. Empty buckets do not consume a block. What is the minimum and maximum number of blocks required to store this hash table?
2. **New Question 2:**
   Suppose a database contains 2,000,000 records, which need to be hashed into 2000 buckets. A block can hold 150 records, and blocks should remain as full as possible without sharing blocks between buckets. What is the minimum and maximum number of blocks needed for the hash table?

Exercise 2: Overflow Handling in Extensible Hash Tables

1. **New Question 1:**
   In an extensible hash table with nnn records per block, what is the probability that two consecutive overflows occur, i.e., all members of the block split again into the same new block twice in a row?
2. **New Question 2:**
   For an extensible hash table where each block holds nnn records, consider a scenario where the records have a uniform distribution. What is the probability of handling an overflow recursively when inserting mmm records?

Exercise 3: Hash Table Evolution

1. **New Question 1:**
   Suppose keys are hashed to five-bit sequences, and blocks can hold four records. If we start with two empty blocks (0 and 1), show how the hash table evolves using:
   a) Extensible hashing for hash values 00000, 00001, ..., 11111.
   b) Linear hashing with a capacity threshold of 75% for hash values 00000, 00001, ..., 11111.
2. **New Question 2:**
   Suppose keys are hashed to three-bit sequences, and each block can hold two records. If we start with three empty blocks (0, 1, and 2), explain how the hash table evolves for the following:
   a) Extensible hashing for hash values 000, 001, ..., 111.
   b) Linear hashing with a capacity threshold of 90% for hash values 000, 001, ..., 111.

Concept of Hashing in Database Management Systems
Hashing is a technique used to map data to a specific location (bucket) in memory or storage, enabling quick retrieval. It is widely used in database systems for indexing and storing data efficiently. The following details break down the core concepts mentioned in the exercises above, explaining hashing with examples.

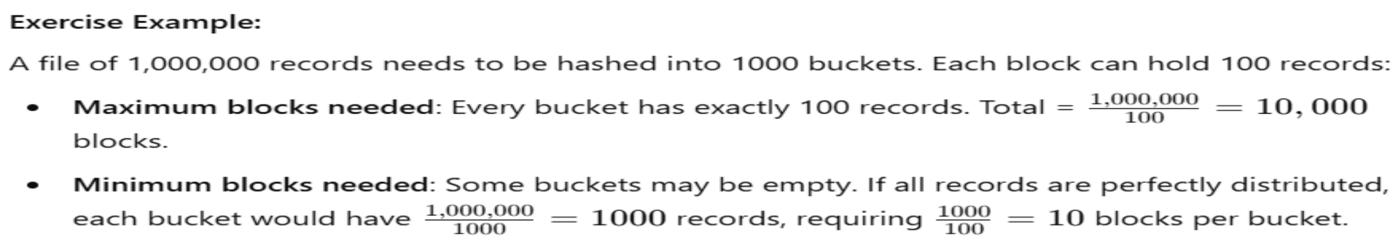
---

1. Hash Tables
A **hash table** is a data structure that organizes data into "buckets" based on a **hash function**. Each data record is assigned to a specific bucket, determined by applying the hash function to a key associated with the record.
Example:
Suppose we have a dataset of student IDs and names:
rust
Code kopieren
```
101 -> Alice
102 -> Bob
103 -> Charlie
104 -> Diana
```
A simple hash function might calculate the bucket as:
makefile
Code kopieren
```
bucket = (studentID % 10)
```
For our dataset, this would map the student IDs to buckets as follows:

- 101 % 10 = 1 → Bucket 1
- 102 % 10 = 2 → Bucket 2
- 103 % 10 = 3 → Bucket 3
- 104 % 10 = 4 → Bucket 4

---

2. Fixed-Size Hash Tables
A **fixed-size hash table** has a predetermined number of buckets. If there are more records than buckets, multiple records may hash to the same bucket, leading to **collisions**. Collisions are resolved using methods like **chaining** (storing multiple records in the same bucket) or **open addressing** (finding the next available bucket).



Exercise Example:

A file of 1,000,000 records needs to be hashed into 1000 buckets. Each block can hold 100 records:

- **Maximum blocks needed**: Every bucket has exactly 100 records. Total = $\frac{1,000,000}{100} = 10,000$ blocks.
- **Minimum blocks needed**: Some buckets may be empty. If all records are perfectly distributed, each bucket would have $\frac{1,000,000}{1000} = 1000$ records, requiring $\frac{1000}{100} = 10$ blocks per bucket.

---

3. Extensible Hashing
Extensible hashing dynamically adjusts the number of buckets as more data is inserted. It starts with a small number of buckets and splits them when they become full, ensuring efficient memory usage.
Example:
Start with 2 buckets for hash keys "0" and "1":

- Insert 0000 → Goes into bucket 0.
- Insert 0001 → Goes into bucket 0.

- Bucket 0 is full. Split bucket 0 into 2 new buckets (00 and 01).
- Redistribute records in bucket 0:
    - 0000 → Bucket 00
    - 0001 → Bucket 01

---

4. Linear Hashing

Linear hashing grows the hash table incrementally, adding new buckets as needed, based on a threshold (e.g., 75% full). Unlike extensible hashing, it does not split buckets recursively.

Example:

Start with 2 buckets (0 and 1), each holding 3 records:

- Insert records: 0000, 0001, 0010, 0011.
- Capacity is exceeded (4 records, threshold = 75%). Add a new bucket (2).
- Redistribute records:
    - 0000 → Bucket 0
    - 0001 → Bucket 1
    - 0010 → Bucket 0
    - 0011 → Bucket 1

---

5. Overflow Handling

When a bucket becomes full, **overflow** occurs. Overflow can be handled using:

- **Chaining**: Add a linked list to the bucket to store additional records.
- **Recursive splitting** (in extensible hashing): Split the bucket into smaller buckets.

Probability of Recursive Splitting:

If n records fit in a block, the probability of all records going into one of the two new blocks during a split is:

$$P = \frac{1}{2^n}$$

For example, if $n = 3$, the probability is:

$$P = \frac{1}{2^3} = \frac{1}{8}$$

---

6. Practical Example

Consider keys hashed to 4-bit sequences and a block size of 3 records. Using **extensible hashing**:

1. Insert 0000, 0001, 0010 → All go into bucket 0.
2. Insert 0011 → Bucket 0 is full. Split into buckets 00 and 01.
3. Redistribute:
    - 0000, 0001 → Bucket 00
    - 0010, 0011 → Bucket 01

---

7. Applications of Hashing

- **Database Indexing**: Fast lookup of data.
- **Load Balancing**: Distributing tasks across servers.
- **Cryptography**: Hash functions like SHA-256.
- **Caching**: Storing frequently accessed data for quick retrieval.