Chapter 4

Data & Storage

A computer does more than just computation. In particular, a computer can also store and retrieve large amounts of data efficiently. In this chapter, we want to understand some of the key ingredients when it comes to dealing with data and storage.

4.1 Dictionary

We manage a library and want to be able to quickly tell whether we carry a given book or not. We need the capability to *insert*, *delete*, and *search* books.

Definition 4.1 (Dictionary)**.** *A **dictionary** is a data structure that manages a set of **objects**. Each object is uniquely identified by its **key**. The relevant operations are* → notebook

- `search`*: find an object with a given key*
- `insert`*: put an object into the set*
- `delete`*: remove an object from the set*

Remarks:

- There are alternative names for dictionary, e.g. key-value store, associative array, map, or just set.
- If the dictionary only offers `search`, it is called *static*; if it also offers `insert` and `delete`, it is *dynamic*.
- When discussing the algorithms, we will often ignore that we actually have a set of objects, each of which is identified by a unique key, and just talk about the set of keys. With regard to the library example, books are globally uniquely identified by a key called `ISBN`. Whenever we say we insert/delete/search a key, we can just drag the key's object along.
- The classic data structure for dictionaries is a binary search tree.

Definition 4.2 (Binary search tree)**.** *A **binary search tree** is a rooted tree, where each node stores a key. Additionally, each node may have a pointer to a left and/or right child tree. For all nodes, if existing, the nodes in the left child tree store smaller keys, and those in the right child tree store larger keys.*

→ notebook

```
1 def search(self, key): # self is current node, initially root
2     if key < self.key:
3       if self.left is None: return None
4       else: return self.left.search(key)
5     elif key > self.key:
6       if self.right is None: return None
7       else: return self.right.search(key)
8     return self.val
```

Algorithm 4.3: Search Tree: Search

Remarks:

- The cost of searching in a binary search tree is proportional to the *depth* of the key, which is the distance between the node with the key and the root.
- There are search trees called *splay trees* that keep frequently searched keys close to the root for quick access. On the other hand, there may be rarely accessed keys deep in a splay tree.
- Using *balanced search trees*, we can maintain a dictionary with worst-case logarithmic depth for all keys, and thus worst-case logarithmic cost per insert/delete/search operation.
- Is there a way to build a dictionary with less than logarithmic cost and with keys that cannot be ordered?

4.2 Hashing

In this section we use hashing to implement an efficient dictionary.

Definition 4.4 (Universe, Key Set, Hash Table, Buckets)**.** *We consider a **universe** U containing all possible keys. We want to maintain a subset of this universe, the **key set** $N \subseteq U$ with $|N| =: n$, where $|N| \ll |U|$. We will use a **hash table** M, i.e. an array M with m **buckets** $M[0], M[1], \ldots, M[m-1]$.*

Remarks:

- The standard library of almost every widely used programming language provides hash tables, sometimes by another name. In C++, they are called `unordered_map`, in Python `dictionary`, in Java `HashMap`.

- The translation from virtual memory to physical memory uses a piece of hardware called *translation lookaside buffer* (TLB), which is a hardware implementation of a hash table. It has a fixed size and acts like a cache for frequently looked up virtual addresses.
- Compilers make use of hash tables to manage the symbol table.

Definition 4.5 (Hash Function)**.** *Given a universe U and a hash table M, a **hash function** is a function $h : U \to M$. Given some key $k \in U$, we call $h(k)$ the **hash** of k.*

Remarks:

- A hash function should be fast to compute and distribute hashes nicely, e.g. $h(k) = k \bmod m$ for a key $k \in \mathbb{N}$; in contrast to Chapter 3, we do not care whether a hash function is one-way.
- If we use `ISBN` mod m as our library hash function, can we insert/delete/search books in constant time?!
- What if two keys $k \neq k'$ have $h(k) = h(k')$?

Definition 4.6 (Collision)**.** *Given a hash function $h : U \to M$, two distinct keys $k, k' \in U$ produce a **collision** if $h(k) = h(k')$.*

Remarks:

- Since keys may experience collisions, the key must be stored in the bucket.
- There are competing objectives we want to optimize for when hashing. On the one hand, we want to make the hash table small since we want to save memory. On the other hand, small tables will have more collisions. How likely is it to get a collision for a given n and m?

Theorem 4.7 (Birthday Problem)**.** *If we throw a fair m-sided dice $n \leq m$ times, let D be the event that all throws show different numbers. Then D satisfies*

$$\mathbb{P}[D] \leq \exp\left(-\frac{n(n-1)}{2m}\right).$$

Proof. We have that

$$\mathbb{P}[D] = \frac{m}{m} \cdot \frac{m-1}{m} \cdot \ldots \cdot \frac{m-(n-1)}{m} = \prod_{i=0}^{n-1} \frac{m-i}{m}$$
$$= \prod_{i=0}^{n-1} \left(1 - \frac{i}{m}\right) = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right)$$

We can use that $\ln(1+x) \leq x$ for all $x > -1$ and the monotonicity of e^x:

$$\mathbb{P}[D] = \exp\left(\sum_{i=0}^{n-1} \ln\left(1 - \frac{i}{m}\right)\right) \leq \exp\left(\sum_{i=0}^{n-1} -\frac{i}{m}\right) = \exp\left(-\frac{n(n-1)}{2m}\right)$$

□

Remarks:

- Theorem 4.7 is called the "birthday problem" since traditionally, people use birthdays for illustration: In order to have a chance of at least 50% that two people in a group share a birthday, we only need 23 people.
- If we insert more than roughly $n \approx \sqrt{m}$ keys into a hash table, the probability of a collision approaches 1 quickly. In other words, unless we are willing to use at least $m \approx n^2$ space for our hash table, we will need a good strategy for resolving collisions.
- Theorem 4.7 assumes totally random hash functions — for non-random distributions of hashes, we might have more collisions. In particular, if we fix a hash function, then we can always end up with a key set N that suffers from many collisions. E.g., if many books have an `ISBN` that ends in 000, then `ISBN` mod 1000 is a terrible hash function.
- Maybe we can use modulo, but with a different m?
- However, for any hash function there are bad key sets.
- On the other hand, for every key set there are good hash functions! How do we efficiently pick a good hash function, i.e. one that is likely to distribute hashes well?

Definition 4.8 (Universal Family)**.** *Let $\mathcal{H} \subseteq \{h : U \to M\}$ be a family of hash functions from U to M. If for all pairs of distinct keys $k \neq k' \in U$, the probability of a collision is $\mathbb{P}[h(k) = h(k')] \leq \frac{1}{m}$ when we choose $h \in \mathcal{H}$ uniformly, then $\mathcal{H}$ is called a **universal family (of hash functions)**.*

Remarks:

- In other words: if we choose a hash function from a universal family, we can expect the hashes to be distributed well, regardless of the key set.
- We cannot just pick a random function from U to M because there are $|M|^{|U|}$ many, so we need $|U| \log |M|$ bits to encode such a random function. That is even more bits than keys in our huge universe U.

Lemma 4.9. *For prime m, $a \in \{0, \ldots, m-1\}$, $\delta \in \{1, \ldots, b-1\}$, and $b \leq m$, the linear function $f_\delta(a) := a \cdot \delta \bmod m$ is a bijection.*

Proof. Suppose there are integers a, a' with $0 \leq a < a' < m$ such that $f_\delta(a) = f_\delta(a') \bmod m$. Then, there exists an integer k such that $a' \cdot \delta = a \cdot \delta + km \Leftrightarrow \delta(a' - a) = km$, which implies that $m \mid \delta(a' - a)$. Since δ is relative prime to m, we have $m \mid (a' - a)$, thus $a = a' \bmod m \Rightarrow a = a'$. □

Theorem 4.10 (Universal Hashing)**.** *Let m be prime and $s \in \mathbb{N}$. Let $U = \{0, \ldots, b-1\}^s$ and let $M = \{0, \ldots, m-1\}$ with $b \leq m$. For a key $k = (k_0, \ldots, k_{s-1}) \in U$ and coefficient tuple $a = (a_0, \ldots, a_{s-1}) \in \{0, \ldots, m-1\}^s$, define*

$$h_a(k_0, \ldots, k_{s-1}) = \sum_{i=0}^{s-1} a_i \cdot k_i \bmod m.$$

Then $\mathcal{H} := \{h_a : a \in \{0, \dots, m-1\}^s\}$ is a universal family of hash functions.

Proof. Let $(k_0, \dots, k_{s-1}) = k \neq k' = (k'_0, \dots, k'_{s-1}) \in U$. Using $\delta_i = k'_i - k_i$ we get

$$h_a(k') - h_a(k) = \sum_{i=0}^{s-1} a_i \cdot k'_i - \sum_{i=0}^{s-1} a_i \cdot k_i = \sum_{i=0}^{s-1} a_i \cdot \delta_i \mod m.$$

The terms with $\delta_i = 0$ are 0, and so we can ignore them. Let $X := \{i \in \{0, \dots, m-1\} : \delta_i \neq 0\}$ be the non-empty set of the indices of the non-zero terms, with $x = |X|$. There are m^x possibilities to choose the coefficients $\{a_i : i \in X\}$. So how many of these possibilities will have a conflict, i.e., a sum of 0? If we choose $x-1$ coefficients arbitrarily, then the last coefficient can always render the sum 0, as according to Lemma 4.9 the last coefficient can add any value $\{0, \dots, m-1\}$ to the sum. In other words, m^{x-1} sums are 0. Therefore, our chance of randomly picking an a that produces a collision is $\frac{m^{x-1}}{m^x} = \frac{1}{m}$. □

Remarks:

- Theorem 4.10 gives us a general method for picking hash functions from a universal family in an efficient manner. We simply choose a prime number m and uniformly at random some factors $a_0, \dots, a_r$. Thus, we can represent our hash function as the tuple $(m, a_0, \dots, a_r)$.
- In practice, hash tables perform really well, and if we detect that we had bad luck in choosing our hash function, we just choose a new one and rebuild our table with the new function — this is called *rehashing*.

4.3 Static Hashing

How can we state the tradeoff between space and collisions more precisely?

Definition 4.11 (Number of Collisions)**.** *Given a hash function $h : U \to M$ and a key set $N \subseteq U$, define the number of collisions that h produces on N as*

$$C(h, N) := |\{\{k, k'\} \subseteq N : k \neq k', h(k) = h(k')\}|.$$

Lemma 4.12 (Space vs. Collisions)**.** *We want a hash function h which produces less than c collisions, i.e. $C = C(h, N) < c$, with $n = |N|$. If h comes from a universal family, a hash table size $m = \lceil \frac{n(n-1)}{c} \rceil$ works.*

Proof. There are $\binom{n}{2}$ pairs of distinct keys in N, and each pair produces a collision with probability at most $1/m$ since h is chosen from a universal family. Using the linearity of expectation we can bound the number of expected collisions:

$$\mathbb{E}[C] \leq \binom{n}{2} \cdot \frac{1}{m} = \frac{n(n-1)}{2m}.$$

We choose m such that $2 \cdot \mathbb{E}[C] \leq c$:

$$2 \cdot \mathbb{E}[C] \leq c \Leftrightarrow \frac{n(n-1)}{m} \leq c \Leftrightarrow \frac{n(n-1)}{c} \leq m.$$

Now we use the Markov inequality which states that for any random variable X that only takes on non-negative integer values, we have $\mathbb{P}[X \geq k \cdot \mathbb{E}[X]] \leq \frac{1}{k}$. Hence, we have $Pr[C < 2 \cdot \mathbb{E}[C] \leq c] \geq \frac{1}{2}$. In other words, to find a hash function with $C < c$ we only need to sample 2 hash functions in expectation.

□

Remarks:

- According to Theorem 4.12, if we want no collisions, we set $c = 1$ and choose $m = \lceil \frac{n(n-1)}{1} \rceil = n(n-1)$.
- Similarly, if we can tolerate n collisions, we find that a hash table of size $m = n - 1$ suffices.
- Algorithm 4.13 defines perfect static hashing, which applies the result of Theorem 4.12.

→ notebook

```python
# Compute primary hash table M and secondary hash tables M_i
def perfect_static_hashing(N): # N = fixed set (list) of keys
  n = len(N)
  M = [None]*n
  h = None
  h2 = [None]*n
  while True:
    h = from_universal_family(n) # h: N -> range(n)
    if C(h,N) < n: break # C counts the number of collisions
  for i in range(n):
    N_i = [x for x in N if h(x) = i]
    n_i = len(N_i)
    if n_i > 1:
      M[i] = [None]*(n_i*(n_i-1))
      while True:
        h2[i] = from_universal_family(len(M[i]))
        if C(h2[i], N_i) < 1: break
  return M, h, h2
```

Algorithm 4.13: Perfect Static Hashing

Remarks:

- In the first stage, we find a hash function h with at most n collisions in linear space according to Theorem 4.12.
- In the second stage, we find a hash function h_i per bucket i without collisions by using an amount of space that is quadratic in the number of keys in the bucket n_i as per Theorem 4.12.

Theorem 4.14 (Perfect Static Hashing)**.** *Algorithm 4.13 returns a collision-free data structure of with a total size (M and all M_i) less than $3n$.*

Proof. The data structure is collision-free because of the *if* condition in the inner *while* loop. Bucket i is of size $n_i(n_i-1) = 2\binom{n_i}{2}$, holds n_i keys that produced $\binom{n_i}{2}$ collisions. Since the total number of collisions is less than n (*if* condition in the first *while* loop), and each bucket is of size twice its collisions, the total size of all buckets is less than $2n$. The size of the primary table M is n, so together the total size of the data structure is less than $n + 2n = 3n$. □

Remarks:

- We now have a hashing algorithm that can be built in linear space and expected linear time, and offers worst-case constant time search for a static set N.
- But what about a dynamic dictionary?

4.4 Dynamic Hashing

Definition 4.15 (Hashing with Chaining)**.** *In **hashing with chaining**, every bucket $M[i]$ stores a pointer to a secondary data structure that manages all keys k with $h(k) = i$. Insertion, search, and deletion of k are all relegated to those data structures. In the simplest implementation, we use a list for each bucket.*

→ notebook

Remarks:

- Algorithm 4.13 is an instance of hashing with chaining with the M_i being the secondary data structures managing the buckets.
- The Java standard library uses hashing with chaining to resolve collisions.

Definition 4.16 (Load Factor)**.** *The fraction $\frac{n}{m} =: \alpha$ is called the **load factor** of the hash table.*

Remarks:

- The performance of all three operations (insert/delete/search) depends on the load factor for all collision resolution strategies.
- Hashing with chaining allows for a load factor $\alpha > 1$ since the size of the table is the number of secondary data structures; performance deteriorates with growing α.
- If we use linked lists as secondary structures and use a hash function chosen from a universal family, the cost for an unsuccessful search is $1+\alpha$ in expectation, while that for a successful search is roughly $1+\frac{\alpha}{2}$ in expectation.
- If we use one of the strategies of this section and α grows too large, we should rehash with a bigger m in order to remain efficient. In the Java standard library, if a hash table surpasses a load factor of 0.75, it is rehashed into a hash table with twice the size of the old one.

Definition 4.17 (Hashing with Probing)**.** *In **hashing with probing**, keys are stored directly in the hash table. The sequence $(h_i(k) \bmod m)_{i\geq 0}$ is called the **probing sequence** of k, and each step of the iteration is a **probe**.*

```
def search(self, k):
  i = 0
  while i < self.size:
    j = self.h(i, k) % self.size  # j = h_i(k)
    key_value = self.M[j]
    if key_value == None: return None
    elif key_value[0] == k: return key_value[1]
    i += 1
  return None
```

Algorithm 4.18: Hashing with Probing: Search

Remarks:

- Algorithm 4.18 defines how to search for a key in hashing with probing. The search is *successful* if the *elif* condition is satisfied. If the preceding *if* is satisfied (or *None* returned at the end) the search is *unsuccessful*.
- To insert a key, we adapt Algorithm 4.18: with an unsuccessful search where *if* was satisfied, we insert in the empty bucket. Therefore, the cost of an insert is roughly the cost of an unsuccessful search. An unsuccessful search where the loop finished triggers a rehash.
- Table 4.19 describes three different types of hashing with probing, each together with the approximate time that a successful or unsuccessful search takes in expectation. More generally, linear probing uses some linear function $h_i(k) = h(k)+ci$ for some $c \neq 0$, and quadratic probing uses some quadratic function $h_i(k) = h(k) + ci + di^2$ with $d \neq 0$. As long as we guarantee that $h_i(k)$ is integer for all $i \in [m]$, the constants c and d can be rational.

Probing	$h_i(k)$	≈ cost successful	≈ cost unsuccess.
Linear	$h(k) + i$	$\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)$	$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$
Quadratic	$h(k) + i^2$	$\frac{1}{1-\alpha} + \ln\frac{1}{1-\alpha} - \alpha$	$1 + \ln\frac{1}{1-\alpha} - \frac{\alpha}{2}$
Double hashing	$h_1(k) + i \cdot h_2(k)$	$\frac{1}{1-\alpha}$	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$

Table 4.19: Different types of hashing with probing together with the expected number of probes per search, where α is the load factor of the table. For hashing with probing, we need $\alpha \leq 1$ since we must have $n \leq m$. Each of h, h_1, h_2 is a hash function drawn from a universal family.

Remarks:

- What is the reason for the different costs in Table 4.19?
- Linear probing suffers from so-called *primary clustering*: the probing sequences are simplistic; if two probing sequences meet in the same bucket j, they will continue to collide in buckets $j+1, j+2, \ldots$
- Quadratic probing does not suffer from primary clustering, but it is subject to *secondary clustering*: if two keys have the same hash, then their probing sequences are identical.
- The form of quadratic probing defined in Table 4.19 has one additional issue: the probing sequence of a key does not necessarily cover the whole table. Assume $m = 7$ buckets and $h(k) = 0$, then the probing sequence of k is $(0, 1, 4, 2, 2, 4, 1)$ — buckets $3, 5, 6$ do not appear.
- Double hashing does not suffer from either version of clustering. One can show that if the hash functions h_1 and h_2 used in double hashing are independently drawn from a universal family, then double hashing performs as well as an idealized hash function that assigns hashes uniformly at random.

4.5 Cuckoo Hashing

So far, the cost of all operations for dynamic key sets has been given in expected time cost. There are algorithms that allow us to do better and give us worst case guarantees on some of the operations. Two widely known possibilities to achieve this are called *dynamic perfect hashing* and *cuckoo hashing*.

→ notebook

```
1 Cuckoo hashing uses two hash tables, and as such provide two
  ↪ possible buckets M_1[h_1(k)] or M_2[h_2(k)] for each key k.
2 If one of the two buckets is empty, simply place k there.
3 If both buckets are occupied by other keys, k is anyway inserted
  ↪ in one of the two possible buckets, replacing some key k_1
  ↪ that currently resides in this bucket.
4 The kicked out key k_1 moves to its other bucket, potentially
  ↪ kicking out the currently resident key k_2; this is repeated
  ↪ recursively until an empty bucket is found.
5 If this recursion loops or takes too long (logarithmic in the
  ↪ table size), the hash table is rebuilt using two new hash
  ↪ functions.
```

Algorithm 4.20: Cuckoo Hashing: Insert

Remarks:

- Search and delete only need to check two buckets to figure out whether a given key is in the table, and so those operations are worst case constant time.
- One can show that the expected insert cost in *cuckoo hashing* is constant as long as the load factor α is below 0.5.
- Cuckoo hashing gets its name from cuckoo birds: they lay their eggs into the nests of other birds, and once the cuckoo chicks hatch, they push the other eggs/chicks out of the nest.
- The idea behind cuckoo hashing is to use the *"power of two choices"*, which can be roughly described as: if you can choose between two resources and use the one that is less busy, you gain efficiency.
- To adapt perfect static hashing to a dynamic setting where we can also handle inserts and deletions, all we have to do is choose the size of M_i twice as large as in Algorithm 4.13, and rehash appropriately: Whenever $C(h_i, N_i) > 0$ for some bucket i, we rehash that bucket until there are no collisions. Once some bucket reaches $n_i^2 \approx |M_i|$ due to insertions, we rehash the entire table. This leaves us with expected constant time insert and delete, and worst case constant time search. To keep the table linear-sized, we rehash everything after every m updates (inserts or deletes).

4.6 Key-Value Databases

Definition 4.21 (Key-Value Database System)**.** *The concept of dictionaries is used in key-value database systems. The server maintains the dictionary and clients can insert and query the stored data using the keys.*

Remarks:

- Popular key-value databases are Redis and Memcached. They are often used for caching in web services. Dynamically generated documents or results of queries to other databases can be stored temporarily to allow fast access to often requested data.
- The data is often kept in main memory to speed up the access and only duplicated to disk to recover the database in case of a system failure.
- Depending on the used database, different data types can be stored in the value. This can be an integer, a string, or even an array.
- Document databases are an extension of simple key-value database systems. The value has to be in a format that the database understands, such as a JSON or XML document. These databases allow queries on the content of the documents. MongoDB and CouchDB are popular document databases.

4.7 Relational Databases

However, most databases offer queries beyond simple key searches. Questions like "What is the movie with the largest cast?" or "How many directors have directed more than ten movies?" should be answered without first writing a new program. Relational databases can store large amounts of *structured* data and answer possibly complex questions about it.

Definition 4.22 (Table, Row, Column, Database)**.** *A* ***table*** *consists of* ***rows****, so that each row (data record) contains the same* ***fields****, i.e., kinds of entries. When the rows of a table are written line by line, the fields form the* ***columns*** *of the table. Each column is referred to by a descriptive name, and is associated with the type of the respective field, e.g., integer, floating point, string, or a date. A* ***database*** *is a collection of tables.*

Remarks:

- In the database context, tables are also called *relations*, because the entries in each row are related to each other, namely by belonging to the same row.

movies

title	director	year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966
⋮		

Figure 4.23: A database containing a single table called "movies" storing the title, director, and year of release for each movie.

Remarks:

- Databases as we study them are accessed using the so-called *structured query language* (SQL). Thus they are referred to as SQL or *relational* databases.
- MySQL and PostgreSQL are two popular open source SQL database systems.
- SQL database systems typically run as a daemon process on some server. Client applications connect to the server and authenticate themselves via username and password. Therefore, multiple users accessing the same database may result in concurrency issues. Some form of concurrency control is necessary!

- Other database systems are tailored to single-user processing. They relieve developers from the burden of implementing efficient data structures for relational data. SQLite is one such example, and is used, e.g., in Firefox, Chrome, Android, Adobe Lightroom, and Windows 10.

4.8 SQL Basics

Definition 4.24 (SQL Data Types)**.** *SQL defines the following types of columns.*

- CHARACTER(m) *and* CHARACTER VARYING(m) *for fixed and variable length strings of (maximum) length* m*,*
- BIT(m) *and* BIT VARYING(m) *for fixed and variable length bit strings of (maximum) length* m*,*
- NUMERIC, DECIMAL, INTEGER*, and* SMALLINT *for fixed point and integer numbers,*
- FLOAT, REAL*, and* DOUBLE PRECISION *for floating point numbers,*
- DATE, TIME*, and* TIMESTAMP *for points in time, or*
- INTERVAL *for ranges of time.*

Remarks:

- The range of each type includes the special value NULL. Note that NULL is different from the string 'NULL', the empty string, and from the number 0 (zero). NULL indicates that the row has *no value* for the corresponding field.
- Many database systems implement more types, e.g., geographic coordinates, IP addresses, geometric objects, or large integers.
- All SQL statements end with a semicolon. The SQL language is case insensitive, but by convention keywords are often typed in upper case.
- The SQL-92 specification is over 600 pages long, newer versions of the standard even longer. To add insult to injury there are lots of vendor specific "SQL dialects", i.e., modifications and extensions. However, the basic set of commands for creating, manipulating, and querying tables are largely the same across database implementations.

CREATE DATABASE *database-name*; → notebook

Additional parameters allow to set database-specific options, e.g., user-based permissions, or default character sets for text strings. How a database is opened depends on the implementation.

CREATE TABLE *table-name* (*field-name type, field-name type, . . .*);

To enforce that all rows have a value for a particular field, one can add NOT NULL to the type when creating the table. Fields have a default value, which is NULL if not specified by adding DEFAULT *value* to the type description.

Remarks:

- There are also GUI and web-based client applications (that execute locally or on an http-server, respectively) and offer access to the database in a more intuitive manner than the classic command line tools. Examples for PostgreSQL are pgAdmin, DataGrip and DBeaver.
- Such tools are especially helpful for creating the databases and tables and often support multiple database systems. They also feature importing data from various formats, e.g., CSV files, instead of using SQL statements to populate the tables.

INSERT INTO *table-name* (*field-name*, ...) VALUES (*value*, ...); → notebook
Values must be listed in the same order as the corresponding field names. When a field name (and thus its value) is omitted the field's default value is assumed. When the list of field names is omitted the field's values must be listed in the same order that was used when creating the table. To insert more than one row in one statement, multiple rows may be separated by commas.

```
SELECT * FROM movies;
SELECT * FROM movies WHERE director = 'Spielberg, Steven';
SELECT title FROM movies WHERE year BETWEEN 1990 AND 1999;
SELECT * FROM movies WHERE title IS NULL OR director IS NULL;
SELECT title, director FROM movies WHERE title LIKE '%the%';
```
→ notebook

Listing 4.25: Querying the movies table.

SELECT *field-name*, ... FROM *table-name* WHERE *condition*;
Lists all specified fields of all rows in the table that fulfill the condition. The special field * lists all fields. The WHERE condition may be omitted to list the whole table. A condition can include comparisons ($<, >, =, <>$) between fields constants. The special value NULL can be tested with IS NULL. Conditions can be joined using parenthesis and logic operators like AND, OR, and NOT. Strings can be matched with patterns using ***field-name* LIKE *pattern***. In the pattern, an underscore (_) matches a single character, whereas % matches arbitrarily many.

```
SELECT MIN(year) FROM movies;
SELECT AVG(year) FROM movies WHERE director='Lumet, Sidney';
SELECT COUNT(*) FROM movies;
SELECT COUNT(DISTINCT director) FROM movies;
```
→ notebook

Listing 4.26: Aggregation with SQL.

SELECT *aggregate*, ...;
Functions for aggregation include AVG to compute the average of a certain field, MIN and MAX for the minimum and maximum value, SUM for the sum of a field, and COUNT to count the number of occurrences. In an aggregation, the keyword DISTINCT indicates that only distinct values should be considered.

```
SELECT director, COUNT(title) FROM movies GROUP BY director;
SELECT director, COUNT(title) FROM movies GROUP BY director
HAVING COUNT(title) > 10;
SELECT year, director, COUNT(title) FROM movies
GROUP BY director, year
ORDER BY year DESC, director ASC;
```
→ notebook

Listing 4.27: Grouping and sorting.

SELECT *field-name*|*aggregate*, ... GROUP BY *field-name*,...;
Aggregations may be partitioned using the group-by clause. Similar to before, the query result can only include aggregates and fields by which the result is partitioned.

Since WHERE clauses are applied before GROUP BY the result of aggregations cannot appear in them. When the result should be conditioned on the result of an aggregation, a HAVING clause can be used.

SELECT ... ORDER BY *field-name*,...;
After each field-name, the keyword ASC or DESC can be used to determine ascending or descending sorting order, respectively.

```
UPDATE movies SET title = 'Star Wars Episode IV: A New Hope'
WHERE title = 'Star Wars';
DELETE FROM movies WHERE title = '';
```

Listing 4.28: Updating and removing rows.

UPDATE *table* SET *field-name* = *value*,... WHERE *condition*;
Updates the specified fields in all rows fulfilling the condition.

DELETE FROM *table-name* WHERE *condition*;
Removes all rows fulfilling the condition from the table.

4.9 Modeling

The way our example table from Figure 4.23 is designed results in lots of duplicate data—the director's name is stored anew for each row, and two directors with the same name cannot be distinguished. The situation worsens when we

want to store the cast of each movie. Clearly the way we modeled our data can be improved. *Entity-Relationship* (ER) diagrams are a tool to find good representations for data.

Definition 4.29 (ER Diagram)**.** *Rectangles denote* ***entities*** *(tables), and diamonds with edges to entities indicate* ***relations*** *between those entities. On such an edge, the number* 1 *or the letter* n *denotes whether the corresponding entity takes part once or arbitrarily many times in the relation. Entities and relations can have* ***attributes*** *(columns) with a name, drawn as ellipses. Italicised attributes are* ***key attributes*** *which must be unique for each such entity.*

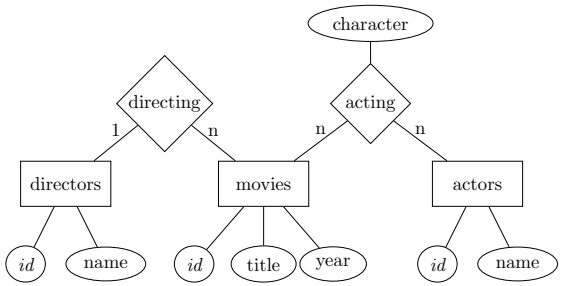

Figure 4.30: Model for a movie database. Movies and directors are in a 1-to-n relation: Each movie is directed by 1 director, and a director may work on many movies. Movies and actors are in a n-to-n relation, which has an additional attribute: An actor may appear in many movies, and each appearance is associated with a character in that movie, played by that actor.

Remarks:

- It is standard practice to assign a so-called *key attribute*, often named *id*, to every entity.
- What do ER diagrams have to do with SQL? Primarily, ER diagrams are for conceptually modeling the kind of data and relations one wishes to store. They can be translated into databases, but not in a unique way.
- A close relative of the ER diagram is the Unified Modeling Language (UML). UML is used to represent the tables of a database (or classes of object oriented software) accurately, with detailed information, e.g. fields.
- Each entity corresponds to a table with the corresponding attributes as columns. An n-to-n relation is represented by a table with columns for each attribute, and a column for the key attribute of each entity in the relation.

actor

id	name
1	Harrison Ford
2	Tom Cruise
⋮	

acting

actor_id	character	movie_id
1	Indy	2
2	Ray Ferrier	3
⋮		

Figure 4.31: The actor table and a table capturing the acting relation.

Remarks:

- The same scheme can be used for 1-to-1 and 1-to-n relations. However, one may also include the relation in the table storing the entity on the 1-side.

directors

id	name
1	Sidney Lumet
2	Steven Spielberg
3	Harold P. Warren
⋮	

movies

id	title	year	director_id
1	12 Angry Men	1957	1
2	Raiders of the Lost Ark	1981	2
3	War of the Worlds	2005	2
4	Manos: The Hands of Fate	1966	3
	⋮		

Figure 4.32: The movie and director tables using the new database layout. The director table simply maps ids to director names. Since the directing relationship is 1-to-n, it can be represented by adding a column to the movies table that stores the director for each movie.

Remarks:

- Similarly, a 1-to-1 relation can be turned into an attribute of one of the entities.
- Tables dedicated to capturing relations are often called *join* tables.

4.10 Joins

How can we access the data, which is now scattered across multiple tables?

→ notebook

```sql
SELECT movie.title, director.name AS director, movie.year
FROM movie
INNER JOIN director ON movie.director_id = director.id;
```

Listing 4.33: Example query that returns the table depicted in Figure 4.34.

SELECT ...
FROM *left-table* INNER JOIN *right-table* ON *condition*;
Returns all rows that can be formed from a row in the left-table and a row in the right-table that satisfy the specified condition.

movie.title	director	movie.year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966
	⋮	

Figure 4.34: The result returned by the query in Listing 4.33.

Remarks:

- In a query, one can create aliases for field and table names using the AS keyword, see Listing 4.33.
- The result of a JOIN clause can be ordered, fields can be aggregated and grouped, and conditions can be added using WHERE clauses.
- For example, we can combine joins and aggregations to answer our initial question of which movie has the largest cast.

```sql
SELECT movie.title, COUNT(*) AS cast_size
FROM acting INNER JOIN movie ON acting.movie_id = movie.id
GROUP BY movie.id ORDER BY cast_size DESC LIMIT 10;
```

→ notebook

Listing 4.35: Finding the 10 movies with the largest cast.

Remarks:

- The query from Listing 4.35 uses a LIMIT clause to return only the ten first entries of the sorted results.
- An INNER JOIN where the condition is TRUE returns the Cartesian product of both tables. This special case can also be obtained with the CROSS JOIN clause.
- An inner join will only return those rows of one table that have a matching row (that satisfies the condition) in the other table. For example, in Listing 4.33, a director with id 5 would not appear in the result if there are no movies which have director_id=5.
- If you want unmatched rows to appear in the result, you need to use an OUTER JOIN.

```sql
SELECT movie.title, director.name AS director, movie.year
FROM movie
RIGHT OUTER JOIN director ON movie.director_id = director.id;
```

→ notebook

Listing 4.36: Example query that returns the table depicted in Figure 4.37.

movie.title	director	movie.year
12 Angry Men	Sidney Lumet	1957
Raiders of the Lost Ark	Steven Spielberg	1981
War of the Worlds	Steven Spielberg	2005
Manos: The Hands of Fate	Harold P. Warren	1966
NULL	Jon Doe	*NULL*
	⋮	

Figure 4.37: The result returned by the query in Algorithm 4.36. The right outer join includes all rows from the inner join (see Figure 4.34) and, additionally, all entries from the directors table for which there is no matching entry in the movies table. In our example, "director" Jon Doe has not directed any movies, hence the movie title and year column are filled with *NULL* values.

SELECT ...
FROM *left-table* LEFT|RIGHT|FULL OUTER JOIN *right-table* ON *condition*;
Returns all rows from the inner join. In addition, a LEFT or RIGHT OUTER JOIN also returns all rows from the left or right table that have no

matching row on the opposite table, respectively. The fields in unmatched rows that cannot be filled from the other table are filled with NULL values. A FULL OUTER JOIN returns both of the above.

Remarks:

- A LEFT OUTER JOIN in Listing 4.36 would include the movies with no director instead of the directors who have not directed any movie.
- Queries may use more than one JOIN clause.

```
SELECT movie.title
FROM actor INNER JOIN acting
ON acting.actor_id = actor.id AND actor.name = 'Harrison Ford'
RIGHT OUTER JOIN movie ON acting.movie_id = movie.id
WHERE acting.actor_id IS NULL;
```

→ notebook

Listing 4.38: Finding all movies that Harrison Ford did not appear in.

Remarks:

- The conditions for the first join in Listing 4.38 ensure that only movies with Harrison Ford are taken into account for the second OUTER JOIN. That second join in turn delivers all movies that cannot be matched, yielding a NULL entry for the actor_id for movies without Harrison Ford.

4.11 Keys & Constraints

What is stopping us from inserting a row in the acting table that contains an actor_id or a movie_id that does not exist? Or from creating a director with a duplicate id?

Definition 4.39 (Key)**.** *In a table, a column (or set of columns) is a **unique key** if the corresponding values uniquely identify the rows within the table. The **primary key** of a table is a designated unique key. A **foreign key** is a column (or set of columns) that references the primary key of another table.*

Remarks:

- SQL databases can automatically enforce these constraints. For example, a row containing a foreign key can only be inserted if it references an existing primary key. Vice versa, a row may only be removed if its primary key is not referenced by any foreign key.

ALTER TABLE *table*
ADD CONSTRAINT UNIQUE (*field-name*,...); → notebook
Any two rows must differ in at least one of the specified fields.

ALTER TABLE *table* ADD PRIMARY KEY (*field-name*,...);
Sets the specified fields as the primary key for the table. Any two rows must differ in at least one of the specified fields. The entries in these fields must not be NULL.

ALTER TABLE *left-table* ADD FOREIGN KEY (*field-name*,...) REFERENCES right-table;
Ensures that the values in the specified fields in the left table are the primary key of a row in the right table.

Remarks:

- Constraints for new tables can also be set using CREATE TABLE.
- Other ALTER TABLE queries add different constraints (e.g., checking that an integer field contains only certain values), remove constraints, and change the name, type or default value of fields.
- To ensure that checking constraints and searching for data is fast, database systems rely on *index* data structures.

4.12 Indexing

Definition 4.40 (Index)**.** *In the database context, an **index** is a data structure that speeds up searching for rows with specific values.*

Remarks:

- Without an index data structure, rows with a specific value can only be found by scanning through the whole table.
- Earlier in the chapter you learned how hash tables can retrieve the row associated with a key quickly. Many database systems implement hash tables as one possible index data structure.

```
CREATE INDEX directorid ON director USING HASH (id);
```

Listing 4.41: Adding a hash table index to our database.

Remarks:

- The director associated with a movie is now found quickly when performing a join.
- Some database systems automatically create index data structures to speed up queries that involve frequently used fields.
- Index data structures have a name—"directorid" in Listing 4.41. This is for referencing it later, e.g., if one decides to delete the index.

- Hash tables scatter the data across the storage (volatile or persistent), and it is likely that every access incurs overhead. Many database queries require scanning through ranges of the data sequentially. For example, when searching the movies from 2000–2005. Thus, accessing supposedly closeby rows requires accessing items at many different places.
- B+ trees are a data structure designed to minimize the amount of I/O operations for both searching and scanning.

```
CREATE INDEX movieyear ON movies USING BTREE (year);
```

Listing 4.42: Adding a B+ tree index to our database.

Definition 4.43 (B+ Tree)**.** *A **B+ Tree** of order b is a rooted search tree mapping **keys** to **rows**. B+ trees are **balanced**, i.e., all leaf nodes are at the same depth.*

Every non-leaf node has between $\lfloor b/2 \rfloor$ and b children. A non-leaf node v with k children contains exactly $k-1$ keys, in sorted order. The ith key of a non-leaf node v is identical to the smallest key in the subtree rooted at v's $(i+1)$st child.

Leaf nodes contain all keys inserted into the tree, together with a pointer which points to the row associated with that key. Every leaf has pointers to at least $\lfloor (b-1)/2 \rfloor$ and at most $b-1$ table rows. Additionally, every leaf w has a pointer which points to its next sibling w'.

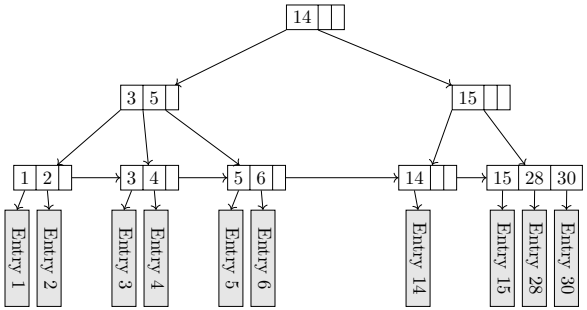

Figure 4.44: Example B+ tree of order $b = 4$.

Remarks:

- The root node is a special case—it may have as little as 2 children if it is not a leaf. If it is a leaf, it may have only one pointer to a table row.

- The order b is sometimes called *branching factor*. To reduce the number of necessary I/O operations, b is chosen so that all data necessary to store a node is the size of one page on disk.
- Finding the row for some key k in a B+ tree works similar to a binary search tree.
- When inserting a key k, we have to check if the leaf v that should contain k is already full (i.e. already contains $b-1$ keys). In that case v, and possibly predecessors of v that contain too many keys, need to be split using B+SplitUp.
- If the root node is split into two nodes v, v', then a new root r containing key k and v and v' as children is created, and the recursion stops.
- Inserting a key k is now performed by first making room using B+SplitUp if necessary, and then inserting k at the leaf.

```
def B+Insert(self, k, r):
  # Search for k to find the leaf v at which k must be inserted
  v = self.search(k)
  if len(v) == b-1:
    B+SplitUp(v, k)
    Replace child of key k with row r in node v
  else:
    Insert key k with row r into node v
```

Algorithm 4.45: B+Insert

Remarks:

- Vice versa, when deleting a key, nodes with too few keys need to be filled up or removed from the tree.
- The properties of B+ trees ensure that every node has at least one sibling. Thus, the merge operation always has two nodes to work with.
- If no keys can be "borrowed" from a sibling, the merge may propagate until the last two children of the root node are merged into one node. In that case the root node is replaced by the merged node, decreasing the height of the tree by 1.

```
def B+Delete(self, k):
  # Search for k to find the leaf v containing k
  v = self.search(k)
  Remove k from v
```

```
5   if len(v) < (b-1)/2:
6     B+MergeUp(v)
```

Algorithm 4.46: B+Delete

Remarks:

- The height of a B+ tree is changed only when inserting a new or removing an old root node. Therefore, all leaf nodes are always at the same depth, thus ensuring the *balanced* property.
- A B+ tree containing n keys has height at most $O(\log_b n)$.
- It may happen that many nodes contain as little as $b/2$ keys, wasting memory and I/O operations. B* trees ensure that nodes contain at least $\frac{2}{3}b$ keys by cleverly "trading" entries with neighboring nodes when they contain too many or too few keys.

4.13 Transactions

Definition 4.47 (Transaction)**.** *A database* ***transaction*** *is a sequence of statements that is executed atomically, i.e. the transaction appears to be instantaneous to an observer.*

Remarks:

- Why would we need transactions? Consider a bank managing customer's accounts using a database system. Alice wants to calculate the liquid assets, and Bob wants to make a money transfer:

```
-- Alice's statement:
SELECT SUM(balance) FROM accounts;
-- Bob's statements:
UPDATE accounts SET balance=balance-100 WHERE customer = 'Bob';
UPDATE accounts SET balance=balance+100 WHERE customer = 'Jim';
```

Listing 4.48: Concurrency issues in databases.

Remarks:

- Assuming that the database system uses multiple threads or processes to process queries, Alice's query may be CHF 100 short.
- To execute the queries atomically, both Alice and Bob can use transactions.

BEGIN TRANSACTION; *statement*$_1$; ...; END TRANSACTION;
Executes the statements atomically.

Remarks:

- One way to implement transactions is to keep track of all fields read from and written to (the read- and write-set, respectively). Then, before a transaction ends, the database system checks whether another transaction wrote to any value in the read-set. If the read-set is unchanged, the write-set can be applied atomically, e.g., by using a global lock.
- SQL offers different so-called isolation levels. The isolation level defines when writes of one transaction become visible to others. The above technique implements the *repeatable reads* level, ensuring that read values were committed before and are not written by another transaction.
- Consider some transaction A that selects all years between 1999 and 2004. What happens if another transaction B concurrently inserts an entry for the year 2000? In the *repeatable reads* isolation level, A may not see B's data if B's insert is scheduled *after* A read all other entries for the year 2000, and A would still be allowed to finish. *Repeatable reads* do not ensure atomicity ...
- The highest isolation level is called *serializable*. This level ensures that the transactions behave "as if they were executed in some sequential order", possibly at the cost of low concurrency.

4.14 Programming with Databases

How do you write an application that relies on a SQL database to store data? Should you construct the necessary SQL statements by manipulating strings, send them to the SQL server, and then parse the result?

Remarks:

- Writing such a SQL client is one possibility, but this is error-prone: The compiler used for the application will not be able to detect errors made in the SQL statements. Moreover, the declarative SQL most likely does not mix well with the programming language chosen for the application.
- One way to mitigate these issues in object oriented programming languages is object/relational mapping.

Definition 4.49 (Object/Relational Mapping)**.** ***Object/Relational Mapping (ORM)*** *is a design pattern used in object oriented programming to store objects in and retrieve them from relational (SQL) databases.*

→ notebook

Remarks:

- In the simplest case, an ORM simply maps a class to a table. An object then corresponds to a row, and the object's attributes correspond to the row's fields.

- The ORM takes care of storing and retrieving object in the database and performs type conversions where necessary. It provides object oriented abstractions for database queries involving WHERE and other clauses. ORMs also remove boilerplate code, i.e., setting up the SQL connection, error handling, data conversion, etc.
- Popular ORMs include SQLAlchemy for Python, ActiveRecord for Ruby, Hibernate for Java, and the Entity Framework for .NET.
- The ORM needs to know how it should translate between objects and rows. For that, many ORM implementations allow to specify the database layout using object oriented methods. Many ORM mappers also support creating the database using the object oriented specification.
- Some concepts from object oriented programming are difficult to model with database concepts, and vice versa. The problems arising from combining these two paradigms are called the *Object-relational impedance mismatch*.

Chapter Notes

Dictionaries based on search trees are useful for providing additional operations such as nearest neighbor queries or range queries, where we want to find all keys in a certain range. Binary search trees were first published by three independent groups in 1960 and 1962 (for references, see Knuth [18]). The first instance of a self-balancing search tree that guarantees logarithmic cost for insert/search/delete is the AVL-tree, named so after its inventors Adelson-Velski and Landis [1]. For multidimensional keys, e.g. geometric data or images, there are specialized tree structures such as kd-trees [3] or BK-trees [5].

Hashing has a long history and was initially used and validated based on empirical results. One of the first publications was Peterson's 1957 article [20] where he defined an idealized version of probing and empirically analyzed linear probing. Universal hashing was introduced two decades later by Carter and Wegman in 1979 [6]. Perfect static hashing was invented in 1984 by Fredman et al. [13] and is sometimes also referred to as FKS hashing after its inventors. Its dynamization by Dietzfelbinger et al. took another decade until 1994 [12]. A comprehensive study on perfect hashing by Czech et al. was compiled in 1997 [11]. Cuckoo hashing is a comparatively recent algorithm; it was introduced by Pagh and Rodler in 2001 [19].

There have been a number of other developments regarding hashing since the late 1970s; for an overview, see Knuth [18], in particular the section on History at the end of chapter 6.4. For a neat visualization of hashing with probing, see [14] online.

The power of two choices paradigm has found widespread application and analysis in load balancing scenarios. It was initially studied from the perspective of a balls-into-bins game where we want to minimize the maximum number of balls in any bin, and to do this we can pick two random bins and put the next ball into the least full of the two bins. Richa et al. [21] compiled an excellent survey on the earliest sources and numerous applications of this paradigm.

In 1970, Edgar F. Codd proposed the relational database model [9] while working at IBM research. Later in the 70s, another group at IBM developed SQL's predecessor SEQUEL (Structured English QUEry Language) [7]. After being renamed SQL due to trademark issues, it was standardized by the ISO in 1987 and later revised [15]. Other companies started developing relational database systems, and nowadays there are many SQL databases implementing different feature sets to choose from.

Around the same time, ER diagrams were conceived as a modeling tool [4, 8]. The Unified Modeling Language (UML), first standardized by the ISO in 1995 [16] and revised in 2012, also includes diagrams that model databases.

B Trees were invented in 1970 [2] for use in file systems. Many variants were studied, among them B* Trees [17], in which at most 1/3 of the memory is unused instead of 1/2 for B Trees. People soon realized that (also for file systems) scanning subsequent rows is an important operation. B+ Trees require at most one I/O operation to find the next element, cf. [17, 10].

Techniques from database systems can also be found in other areas of computer science. Transactions as a parallel programming model have been adotped for other programming languages under the term *transactional memory*. Ideas developed to ensure that database transactions appear atomic w.r.t. writing data to disk were adopted by general purpose file systems under the name *journaling*.

This chapter was written in collaboration with Georg Bachmeier and Jochen Seidel.

Bibliography

[1] M Adelson-Velskii and Evgenii Mikhailovich Landis. An Algorithm for the Organization of Information. *Doklady Akademii Nauk USSR*, 146(2):263–266, 1962.

[2] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, 1970.

[3] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[4] A. P. G. Brown. Modelling a real world system and designing a schema to represent it. In *IFIP TC-2 Special Working Conference on Data Base Description*, 1975.

[5] W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM*, 16(4):230–236, 1973.

[6] J.Lawrence Carter and Mark N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143 – 154, 1979.

[7] Donald D. Chamberlin and Raymond F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '74. ACM, 1974.

[8] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Trans. Database Syst.*, 1976.

[9] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 1970.

[10] Douglas Comer. The ubiquitous B-Tree. *ACM Comput. Surv.*, 1979.

[11] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. Perfect hashing. *Theoretical Computer Science*, 182(1 - 2):1 – 143, 1997.

[12] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert, and Robert E. Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[13] Michael L. Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0(1) worst case access time. *J. ACM*, 31(3):538–544, 1984.

[14] David Galles. Closed hashing. `https://www.cs.usfca.edu/~galles/visualization/ClosedHash.html`. Accessed: 2017-03-29.

[15] International Organization for Standardization. Information technology – Database languages – SQL – part 1: Framework (SQL/Framework), 2011. ISO/IEC 9075-1.

[16] International Organization for Standardization. Information technology – Object Management Group Unified Modeling Language (OMG UML) – Part 1: Infrastructure, 2012. ISO/IEC 19505-1.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1973.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[19] Rasmus Pagh and Flemming Friche Rodler. *Algorithms — ESA 2001: 9th Annual European Symposium Århus, Denmark, August 28–31, 2001 Proceedings*, chapter Cuckoo Hashing, pages 121–133. Springer Berlin Heidelberg, 2001.

[20] W. W. Peterson. Addressing for random-access storage. *IBM J. Res. Dev.*, 1(2):130–146, 1957.

[21] Andrea W Richa, M Mitzenmacher, and R Sitaraman. The power of two random choices: A survey of techniques and results. *Combinatorial Optimization*, 9:255–304, 2001.