UV Distributed Information Management
Summer semester 2024

Supplementary Material – Computer Science Terminology

This document provides supplementary material regarding terms that are commonly used in the context of computer science.

Figure 1 summarizes the most important parts of a computer system (CPU, caches, main memory, and SSD/HDD) for our course.

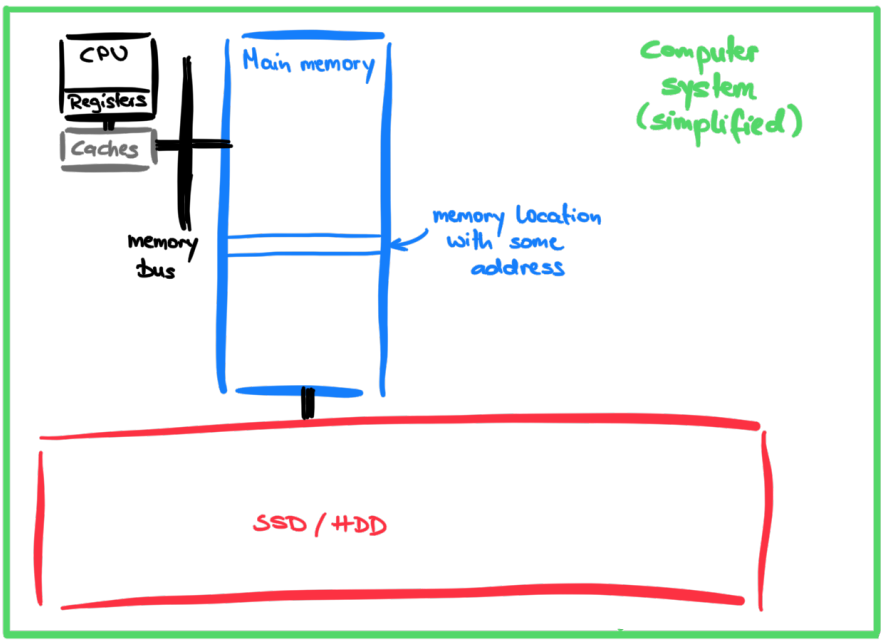

Figure 1: A computer system as geographical map (simplified).

Processor / CPU (Prozessor, in german) CPU [1] denotes the *central processing unit*, which is (as the name suggests) the main computational unit in a computer system. A CPU implements a particular instruction set architecture (ISA) [2]. The ISA defines a set of so-called *instructions* and a CPU "understands" (i.e., is able to execute) only instructions from this set (encoded in binary format [3]). An executable application is stored as a sequence of bits (each of which can be 0 or 1) and a CPU typically reads a fixed amount of bits that are translated (i.e., decoded) into the corresponding instruction. Most importantly, a CPU understands *exactly* and *only* this set of instructions and nothing else. When we switch on the power of our computer system, the CPU

[1] Central processing unit: `https://en.wikipedia.org/wiki/Central_processing_unit`
[2] Instruction set architecture: `https://en.wikipedia.org/wiki/Instruction_set_architecture`
[3] Binary format: `https://en.wikipedia.org/wiki/Binary_file`

starts to infinitely run in a so-called *cycle* [4] that naively executes one instruction after another (in multiple phases) until power is switched off again. Note that a CPU does not have any notion of intelligence. It simply (or "stupidly") executes the instructions we tell it to execute, e.g., by writing code in some programming language that is translated into a sequence of instructions. In our course, we use the terms CPU and processor synonymously.

Main memory (RAM; Hauptspeicher, in german) Main memory [5] (or RAM; random-access memory [6]) is a so-called *volatile* memory [7] in a computer system that stores code and data during execution of an application. Typical main memory sizes range from $1 - 2$GB to sometimes $>$ 100GB (nowadays; 1GB = 1Gigabyte = 2^{30}bytes [8] = about a billion bytes). Importantly, information that is stored on volatile memory is lost as soon as the power is switched off, i.e., *all* bits are set to 0. Therefore, main memory is quite fast as it does not have to care about storing the data permanently. Eeach single bit of code and data during the execution of an application is stored in main memory (at some point). When we execute an application (i.e., our code), the corresponding bits are transferred from main memory to the CPU instruction by instruction, each of which is then executed by the CPU.

HDD (Festplatte, in german) HDD [9] denotes the *hard disk drive* and denotes a *non-volatile* memory [10] that is used to store information permanently (even when the power is switched off). Typical HDD sizes range from tens of GB to sometimes tens or even hundreds of TB (nowadays; 1TB = 1Terabyte = 2^{40}bytes = about a trillion bytes). Although HDDs provide much more storage, they are about 5 orders of magnitude slower compared to main memory (cf. Figure 2). Therefore, computer systems want to minimize accesses to information that is stored on the HDD if possible (and rather retrieve it from main memory). Nonetheless, sometimes it is simply not possible to avoid the HDD access because information that must not be lost is stored there. Once our computer system is running, information (i.e., bytes) is regularly transferred from the slower to the faster memory (e.g., from HDD to main memory) in order to speed up the overall system performance.

SSD SSD [11] denotes the *solid-state drive* and denotes the de-facto successor technology to HDDs. The properties of SSDs are similar to the properties of HDDs (i.e., SSDs are non-volatile memory that store information permanently) but the underlying technology is completely different (HDDs are based on magnetic disks; SSDs are based on flash storage). SSDs are faster than HDDs but still 2-3 orders of magnitude slower than main memory. Therefore, computer systems still try to minimize access to information that is stored on SSDs (as accesses are slow compared to main memory). Typical SSD sizes range from tens of GB to sometimes a few TB (nowadays).

[4] Instruction cycle: `https://en.wikipedia.org/wiki/Instruction_cycle`
[5] Main memory: `https://en.wikipedia.org/wiki/Computer_data_storage#Primary_storage`
[6] Random-access memory: `https://en.wikipedia.org/wiki/Random-access_memory`
[7] Volatile memory: `https://en.wikipedia.org/wiki/Volatile_memory`
[8] Byte: `https://en.wikipedia.org/wiki/Byte`
[9] Hard disk drive (HDD): `https://en.wikipedia.org/wiki/Hard_disk_drive`
[10] Non-volatile memory: `https://en.wikipedia.org/wiki/Non-volatile_memory`
[11] Solid-state drive (SSD): `https://en.wikipedia.org/wiki/Solid-state_drive`

Cache (Zwischenspeicher, in german) A cache [12] is essentially a small but very fast piece of memory in a computer system, which is used to temporarily store data that has been transferred from main memory to the CPU. The objective of a cache is to speed up future accesses to the data/code that is *cached* (i.e., currently stored in the cache) and to reduce accesses to main memory. Accesses to data/code that is stored in the cache are faster than accesses to main memory due to differences in technology and the caches being physically closer to the CPU. Modern computer systems typically have multiple cache levels (e.g., L1, L2, L3): For $x < y$, $1 \leq x, y \leq 3$, Lx is smaller and faster than Ly since Lx is physically (or geographically if we consider the computer systems as a map) closer to the CPU than Ly. Typical cache sizes range from tens of KB (L1; 1KB = 1Kilobyte = 2^{10}bytes = about a thousand bytes) to tens or hundreds of MB (L3; 1MB = 1Megabyte = 2^{20}bytes = about one million bytes).

Memory hierarchy (Speicherhierarchie, in german) This term refers to the hierarchy of memory [13] units that are present in basically every (modern) computer system. We have already learned about main memory, caches, SSDs, and HDDs, but there are even more memory units like CPU registers. The memory hierarchy is a pyramid that tells us about the *size* (i.e., storage capacity), the *speed* (to access some code/data that is stored there), and the volatility (i.e., is the content of the memory unit lost or not when we switch off our system) of a particular memory unit. The smallest, fastest, volatile, and (typically) most expensive memory units are located at the top of this hierarchy, whereas the largest, slowest, non-volatile, and (typically) cheapest memory units are at the bottom. Figure 2 visualizes the hierarchy: At the top, we find the CPU registers, which are tiny memory units (only a few bytes) that are *inside* the CPU. Subsequently, we find the different levels of caches, main memory, SSDs, and HDDs (in this order). Further below, we find very large but very slow memory units like magnetic bands that are used for long-term backups (over decades). There exist more types of memory units in a modern computer system but for our purpose the memory units shown in Figure 2 suffice. With this hierarchy, we can directly see why systems typically want to minimize the accesses to HDD and even SSD: The performance gap between main memory and HDD is about 5 orders of magnitude, i.e., an access to main memory is 10^5 = about $100,000$ times faster than the equivalent access to HDD (and about 10^2 = about 100 times faster than the equivalent access to SSD). On the one hand, main memory is based on a "faster" technology than SSDs and HDDs, respectively. On the other hand, this is also due to the physical (or geographical) location *within* our computer system, i.e., memory units that are physically (or geographically) located nearby the CPU are faster because every signal in a computer system "travels" at the speed of light ($299,792,458$m/s; we cannot overcome this physical limitation) [14]. Consequently, data/code that resides in, say, the L1 cache can be accessed much faster than from memory units that are farther away from the CPU. Figure 1 illustrates this principle in a computer system. As a real-world analogy, we can consider traveling by car at a constant speed of 100km/h from Salzburg to (a) Linz (about 130km) and (b) Vienna (about 300km). Our journey to Linz will takes us about 1.3 hours, whereas our journey to Vienna will take us about 3 hours. A similar reasoning can be applied for the other levels in our hierarchy.

Physical level (Physische Ebene, in german) In a computer system, this level (or layer) refers to the physical components, e.g., main memory, the CPU, and HDDs, and denotes the lowest level where *everything* is encoded in bits and bytes (i.e., a sequence of 8 consecutive bits).

[12] Cache: `https://en.wikipedia.org/wiki/Cache_(computing)`
[13] Memory hierarchy: `https://en.wikipedia.org/wiki/Memory_hierarchy`
[14] Speed of light: `https://en.wikipedia.org/wiki/Speed_of_light`

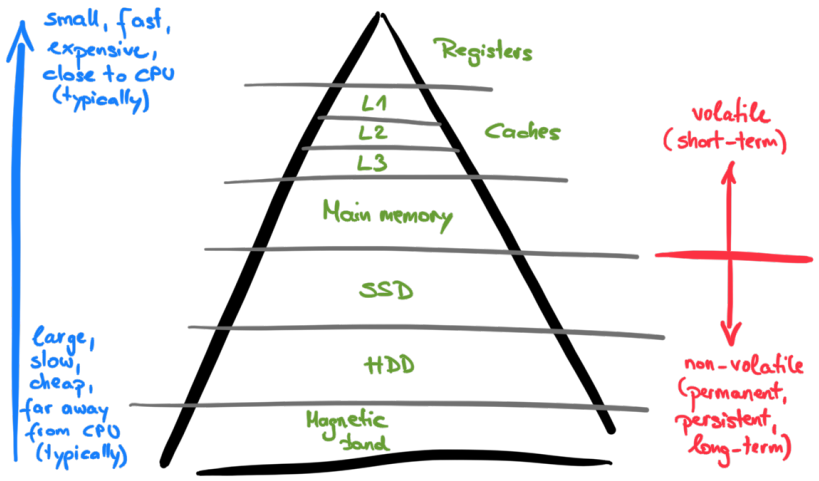

Figure 2: The portion of the memory hierarchy that is relevant for our course.

At this level, a computer system is just an electrical circuit (or many interconnected electrical circuits) that modifies the bits. The fact that we can use a computer system as we do nowadays is based on additional levels that hide the complexity of this level from us.

Instruction(s) (Instruktion(en), in german) We already know that a CPU has a defined set of so-called *instructions* that basically denotes the language it understands in binary format. An instruction [15] has a defined behavior and represents an operation *encoded* as sequence of bits (binary format). Table 1 lists selected (and simplified) example instructions.

Table 1: Selected (and simplified) example instructions.

Instruction	Description
ADD	Adds two numbers (e.g., variable values) and stores result in another variable.
LOAD	Loads the content of a memory location (address), e.g., the value of a variable.
JUMP/BRANCH	Jumps to another instruction in the code.

The JUMP and BRANCH instruction represents control-flow, i.e., it encodes `if-else` branches, loops, and function calls in high-level programming languages [16]. Every code that we write in a (high-level) programming language is (at some point) translated into a sequence of such instructions. Instructions effectively determine the behavior of our computer system. Once we press the power button, the CPU starts to execute the very first instruction until the startup process is finished. Everything we do on our machine (e.g., navigate the mouse pointer, write some document, browse through the internet, or execute our own code) results in many instructions that are execute by the CPU at a very high pace ($> 10^9$ instructions per second) in the background and we observe the result of these instructions (e.g., the mouse pointer moves, a character appears in our document, a website loads, or our code does something).

[15] Instruction(s): `https://en.wikipedia.org/wiki/Instruction_set_architecture#Instructions`
[16] High-level programming language: `https://en.wikipedia.org/wiki/High-level_programming_language`

Operation(s) (Operation(en), in german) An operation [17] is the result of an instruction that is executed by the CPU. For example, an instruction may encode addition of two numbers (i.e., the `ADD` instruction) with the result being stored in some register. The corresponding operation is "Addition" and the execution of this instruction changes the value of a register. In this sense, an operation can be seen as the behavior of our computer system when executing the corresponding instruction. Furthermore, instructions are typically small, i.e., they serve a very restricted purpose. If we want to add 5 numbers and store the result in a register, then we need to execute multiple `ADD` instructions consecutively because one `ADD` instruction can only add two numbers at a time. From this point of view, operations may also be considered to contain multiple instructions (depending on the context).

Address (Adresse, in german) An address [18] in a computer system uniquely identifies a particular location in memory. Abstractly, memory can be seen as a big box with many tiny slots each of which can, for example, store 64 bits. In order to unambiguously refer to a particular slot, the slots are numbered consecutively from 0 to $n - 1$ (for n slots), just like addresses in the real world. If an application uses a variable, then each variable is associated with one or multiple such slots in memory (depending on its size). Whenever the variable is read or modified, the computer system then automatically accesses the corresponding slot(s) in memory and retrieves or overwrites the content of the slot(s). The storage capacity of our memory is then the sum of all these tiny slots in this box.

Parallelism (Parallelismus, in german) (Hardware) Parallelism [19] refers to the *physical* ability of a computer system to perform multiple actions truly *simultaneously*. This is only possible if there exist multiple physical resources (typically CPUs) that can independently work on different tasks. As a real-world analogy, we consider k students in class (the CPUs) that can perform different tasks independently (from start to finish, without interruption). Now consider $10 \cdot k$ tasks that need to be processed during class. A single student has to work on every single task one after another. Contrarily, k students can work on k tasks *in parallel*. Therefore, to overall time to finish $10 \cdot k$ tasks is 10 times lower for k students, i.e., we get a $10\times$ speed up in our system.

Concurrency (Nebenläufigkeit, in german) This term refers to the *illusion* that multiple actions *seemingly* happen simultaneously, but they actually do *not* happen in parallel because there are not enough physical resources for parallelism. Consider a single CPU that can only execute a single instruction at a time and k tasks that are supposed to be executed simultaneously (e.g., we use multiple applications like a browser, an editor, an email client, and so on). In order to create the illusion of seemingly parallel execution of all k tasks, the instructions of each task are divided into blocks, e.g., one block may contain 10 instructions. The CPU starts to execute the first 10 instructions of Task_1. Then, the CPU *switches* to Task_2 and executes the first 10 instructions of Task_2. The same principle is applied until the first 10 instructions have been executed for every task Task_1–Task_{10}, and then the CPU starts to continue in this manner with the next block of instructions for Task_1 (i.e., instructions 11-20 of Task_1). In computer systems, a CPU can typically execute $> 10^9$ instruction per second, hence a human is not able to properly observe the fact that the tasks are not really executed in parallel but only

[17] Operation(s) in computer systems: `https://en.wikipedia.org/wiki/Operator_(computer_programming)`
[18] Memory address: `https://en.wikipedia.org/wiki/Memory_address`
[19] Parallelism: `https://en.wikipedia.org/wiki/Parallel_computing`

concurrently [20]. Reusing the real-world analogy from our discussion on parallelism, this means that we only have a single student that works on k tasks *concurrently*. Each task may be to write down a particular Wikipedia article on a sheet of paper and a block of 10 instructions means that the student can write down 10 characters from the corresponding article. The student starts writing down the first 10 characters of Article_1, then the first 10 characters of Article_2 (on a different sheet of paper), and so on. After writing down the first 10 characters of each article, the student starts over and writes down the next 10 characters of each article until every article is fully replicated. In the end, the result is identical as if the student would have written down each article completely (one article after another).

Compiler (Übersetzer, in german) A compiler [21] refers to a tool in computer science that allows us to translate code that is written in a (high-level) programming language (e.g., Java) into a representation that can be executed on our CPU. That is, a compiler translates a given code into a sequence of bits (binary format), which encodes a sequence of instructions that is semantically equivalent to our code written in some programming language. The CPU is then able to execute one instruction of this sequence of instructions after another, and this effectively reflects the behavior of our code. Typically, a portion of our code (e.g., `if-else`) is translated into multiple instructions.

Virtual machine (VM; virtuelle Maschine, in german) Sometimes, we want to execute code that is based on instructions that our native CPU does *not* understand, and it is no option to use a different hardware (i.e., CPU). For example, ARM processors [22] that are often used in smartphones are very different from Intel's x86 processors [23] (which are common in Desktop computers). In this case, a virtual machine [24] (VM) may help: We can use a so-called *hypervisor* software [25] like VirtualBox [26] to create a virtual machine that provides the functionality of a "real" physical computer system (CPU, main memory, and the likes). However, all this functionality is implemented in software (rather than in hardware). For example, we can create a virtual machine that runs ARM instructions and *host* it using, e.g., VirtualBox, on our Desktop computer that is based on the x86 architecture. VirtualBox then acts as link between our x86 and our ARM architecture. Moreover, virtual machines are also quite useful for reproducibility, i.e., to reproduce a particular behavior on a different physical machine. The latter is also the reason why we use virtual machines in this course: It is easier for the instructor to reproduce an error that you encountered when you and the instructor use the same virtual machine.

Time (Zeit, in german) This term is mostly used in combination with *complexity*, i.e., computer scientist often argue about *time complexity* [27]. It refers to the amount of *runtime* it takes on a computer system to execute an algorithm [28] that serves a specific task. The time complexity is then the overall amount of runtime to finish the algorithm. Assuming that an operation in a computer system takes a constant amount of time, the time complexity is

[20] Concurrency: `https://en.wikipedia.org/wiki/Concurrent_computing`
[21] Compiler: `https://en.wikipedia.org/wiki/Compiler`
[22] The ARM architecture family: `https://en.wikipedia.org/wiki/ARM_architecture_family`
[23] The x86 architecture: `https://en.wikipedia.org/wiki/X86`
[24] Virtual Machine: `https://en.wikipedia.org/wiki/Virtual_machine`
[25] Hypervisor: `https://en.wikipedia.org/wiki/Hypervisor`
[26] VirtualBox: https://en.wikipedia.org/wiki/VirtualBox
[27] Time complexity: `https://en.wikipedia.org/wiki/Time_complexity`
[28] Algorithm: `https://en.wikipedia.org/wiki/Algorithm`

estimated by "counting" the number of operations the system needs to execute. We will use this term in a simplified manner (no formal definition) to argue whether something runs fast or rather slow in a computer system.

Space (Raum, in german) This term is also mostly used in combination with *complexity* and is orthogonal to time complexity. However, *space complexity* [29] refers to the amount of *memory* it takes on a computer system to execute an algorithm that serves a specific task. The space complexity is then the overall amount of memory that is consumed to finish the algorithm. Similar to time complexity, we will use this term in an informal manner to argue whether something needs a large or rather small amount of memory in a computer system.

Performance (Performanz, in german) Performance [30] is a rather general term that refers to the amount of *meaningful* work in a computer system, i.e., how *effectively* a system can achieve its goal or task. In our course, performance is mostly related to runtime and memory, but also to availability and data transmission when discussing distributed systems.

Efficiency (Effizienz, in german) Efficiency [31] is often confused with performance and in some sense they are related. However, efficiency refers to how *good* a particular system or algorithm is able to achieve its goal or task in terms of resource utilization. In our course, efficiency is mostly related to runtime and memory. For example, a computer system is runtime- and memory-efficient if it achieves its goal or task while consuming only a little amount of time and memory (i.e., it is not wasteful). Moreover, modern systems are often required to be *energy-efficiency*, i.e., to consume only little energy while achieving its goal or task. Of course, a system can also achieve its goal or task by following a different, less efficient strategy but the user/company may pay the price (e.g., by paying higher energy or hardware costs).

Scalability (Skalierbarkeit, in german) This term refers to the ability of a computer system to handle larger amounts of work by possibly adding additional hardware resources such as CPUs, memory, or even full machines (in a distributed system). For example, assume our computer system is required to execute a task X_1 of size $|X_1|$, and our system is able to do this in t_1 seconds. Now, we double the amount of work, i.e., we execute a task X_2 of size $|X_2| = 2 \cdot |X_1|$. As a consequence, our system will probably not finish the task X_2 in t_1 seconds, but it will need more time $t_2 > t_1$. The factor $\frac{t_2}{t_1}$ is called *scalability factor* [32], and we want to keep this factor constant and small. Another option is to double the number of hardware resources, e.g., the number of CPUs in our system. Then, we may be able to keep our initial runtime t_1 even for task X_2, but only because we leverage additional computational power. In this case, we scale the CPUs and, again, we want a constant and low number of additional CPUs that are required to keep t_1. As a real-world analogy, we can borrow the scenario of our discussion on parallelism: A student is able to achieve 1 task in t_1 seconds and we want to finish 10 tasks. If we stick with a single student, the overall time t_{10} will be about 10 times higher. Contrarily, if we *scale out* to 10 students each of which works on a different task independently in parallel, then we may end up with all 10 tasks being finished in t_1.

[29] Space complexity: `https://en.wikipedia.org/wiki/Space_complexity`
[30] Performance: `https://en.wikipedia.org/wiki/Computer_performance`
[31] Efficiency: `https://en.wikipedia.org/wiki/Efficiency`
[32] Scalability: `https://en.wikipedia.org/wiki/Scalability`

Throughput (Durchsatz, in german) Throughput [33] is often used in the context of computer networks to refer to the number of messages that may be delivered in a defined period of time (typically in 1 second). However, it may be used analogously to assess the throughput of a computer system in terms of finished tasks per second. For example, database systems typically report their throughput in queries per second for particular workloads. The higher to throughput, the higher the performance of the computer system.

Redundancy (Redundanz, in german) In general, redundancy [34] refers to the duplication of a resource where each duplicate serves the same (or a very similar) purpose. Examples include multiple copies of the same data or multiple wires that connect a building to the internet. Depending on the specific context, redundancy may have a positive or negative connotation. On the positive side, redundancy allows us to tolerate errors in systems because we can simply use a different copy to achieve our goal or task if the original copy is unavailable (for whatever reason). For example, if we want our system to be available 24 hours, 7 days a week, 365 days a year, we may have multiple different links to connect to the internet. If our primary link is down, we simply use one of the other (redundant) links. On the negative side, redundancy in data may also cause anomalies and inconsistencies as well as performance degradation because we need to keep all the data copies up to date.

Transparency (Transparenz, in german) This term has many different meanings even in the context of computer science. In our course, we will mainly use the adjective form of this term in combination with functionality. A functionality is considered *transparent* to the user if the complexity of this functionality is hidden from the user. For example, cloud services are typically transparent to the user because the user does not need to care about how and where the data is stored without being lost. Another example is the transparent translation from website URLs into IP addresses (i.e., addresses that uniquely identify a website in the internet).

Database system (DBS; Datenbanksystem, in german) A database system [35] (DBS) is a dedicated software system that manages (typically) large amounts of data and provides functionality to users to access and modify this data transparently.

Query (Anfrage, in german) Users typically interact with database systems using so-called *queries*. A query is formulated in a so-called *query language*, which is a well-defined computer language that allows users to specify what they want the database system to do, e.g., retrieve or modify a specific portion of the data or create new entries.

Transaction (Transaktion, in german) Transactions [36] are commonly known from electronic banking systems, where transaction are used to transfer some amount of money between two users. In database systems, this concept is generalized to encapsulate multiple operations into one *transaction*. Each transaction is typically an *atomic* and *independent* unit, that is, a

[33] Network throughput: `https://en.wikipedia.org/wiki/Network_throughput`
[34] Redundancy: `https://en.wikipedia.org/wiki/Data_redundancy` and `https://en.wikipedia.org/wiki/Redundancy_(engineering)`
[35] Database system: `https://en.wikipedia.org/wiki/Database`
[36] Transaction: `https://en.wikipedia.org/wiki/Database_transaction`

transaction is executed in an all-or-nothing manner and transactions do not influence each other if they are executed concurrently. If the database system fails to execute a transaction successfully, then the transaction is reverted such that the system is in the state it has been before the transaction. Revisiting the electronic banking system, we intuitively observe that this is the desired behavior as we would not be happy if the money is lost during the transfer (i.e., neither of the two users has the money in the end).

Index An *index* [37] is a data structure in a database system that is used as shortcut to some portion of the data in order to speed up queries. There exist different types of indexes (depending on their purpose), but for our course, it suffices to view an index as a black box that provides a shortcut. A real-world analogy can be found at the end of basically every technical book: The index lists words together with the page numbers where the respective term occurs.

Imperative (Imperativ, in german) In imperative programming [38] we specify *how* the computer system achieves its goal or task. In other words, we use statements to specify the commands the computer system must execute, i.e., we describe the single steps the computer must execute one after another to finally achieve its goal or task. In the most extreme case, we need to specify every single tiny step one by one. As a real-world analogy, we can tell a person to get us some milk, and we specify every single step the person needs to do (i.e., specify the supermarket, the shelve, the brand, and so on – the person has no freedom but naively follows our instructions).

Declarative (Deklarativ, in german) In contrast to imperative programming, declarative programming [39] refers to a programming paradigm where we only specify *what* we want the computer system to achieve, but do not care about how it achieves it. In the most extreme case, we do not know anything about how the system achieves its goal or task. Reusing the real-world analogy from our discussion on imperative programming, we would only tell the person to get us some milk and nothing else (i.e., we do not care about the specific supermarket, shelve, brand, and so on – it is up to the person to decide).

Trade-off (Kompromiss, in german) This is a general term that is used in many different domains. In computer science, this term is important because computer systems often have to *trade* some property in order to enable another property. A traditional trade-off [40] in computer systems is the so-called *time-space trade-off:* If we allow the system to consume more memory, then this may (significantly) speed up the computation. Contrarily, if we allow the system to consume more time (e.g., be slower), then we may end up consuming less (or even no) memory at all. As a real-world analogy, we consider a person that wants to know the individual results of 10 calculations where the second calculation depends on the first calculation, the third depends on the second one, and so on. If we allow the person to use a sheet of paper to write down the results of previous calculations, they can simply reuse these results to calculate the remaining results. Contrarily, if we do *not* allow the person to use a sheet of paper, they must recalculate every previous result over and over again. Obviously, the usage of a sheet of paper (i.e., more

[37] Index: `https://en.wikipedia.org/wiki/Database_index`
[38] Imperative programming: `https://en.wikipedia.org/wiki/Imperative_programming`
[39] Declarative programming: `https://en.wikipedia.org/wiki/Declarative_programming`
[40] Trade-off: `https://en.wikipedia.org/wiki/Trade-off`

memory) speeds up the process. Hence, we trade space for time (and vice versa). During our course, we will learn about different trade-offs in the design of a database system.

Distributed (Verteilt, in german) This course is about *distributed* information management, where *distributed* [41] refers to the property of a software system to be physically scattered over multiple physical machines, each of which is located in different geographical locations. In order to communicate with each other, the physical machines are interconnected using a dedicated network or simply the internet. Typically, distributed systems provide their functionality to the user transparently.

Terminal (Konsole/Shell, in german) The terminal [42] is (almost) the most basic way to directly execute commands in a computer system. Depending on the operating system (Windows, MacOS, Linux, and so on), the terminal offers a defined set of commands that may be directly executed, e.g., to list all files in a particular directory: `ls -l` (Linux, MacOS) or `dir` (Windows). In general, the terminal is capable of executing any application that is compiled for the CPU architecture at hand (e.g., x86 or ARM), including applications that have been developed by us.

[41] Distributed systems: `https://en.wikipedia.org/wiki/Distributed_computing`
[42] Terminal: `https://en.wikipedia.org/wiki/Computer_terminal`