Introduction to concurrency. Concurrency models

This chapter is just an introductory to concurrency. It talks about things that are usually already known from other courses like Parallel and distributed programming, Operating systems and other courses. When talking about concurrent programming, we usually say that there are three types of programming paradigms related to concurrency: sequential programming, parallel programming, concurrent programming. We can see these programming paradigms depicted in Fig. 1. In this figure, the time runs vertically, top to bottom. The first paradigm is the simplest, sequential programming. In sequential processing, we have the first task which is executed (i.e. tasks are represented as vertical or oblique line segments) and only when the first task gets completed, then the second task is executed and when the second task is completed the third task can start executing and so on. Some properties of sequential processing are: the actual code has very good clarity meaning it is usually written in an imperative programming language, there is no randomness that would impact the execution order, everything is processed sequentially one at a time. The correctness is pretty clear, so in the case of simple programs, by instantly looking at the code you can generally conclude if the program is correct or not, and in the case of more complex problems, if you run a reasonable number of tests, you get an idea whether the program is correct or not (of course, you can not know this for sure due to the possible infinite sets of input data – i.e. the data input by the user). Most importantly, consecutive runs of the same program with the same input data will all return the same results. A significant disadvantage of the sequential programming paradigm is its low performance because you can only do one thing at a time. If you imagine a web service that does something over the web, if the web server is implemented in a sequential paradigm, then the web server can only handle one client at a time.
So this is obviously not good for performance. Another programming paradigm is the (pure) parallel programming. Here we have a bunch of processes or threads or execution tasks or call them however you want, i.e. parallel entities, they get executed in parallel on different CPUs or different CPU cores. If the parallel process execute the same work as the sequential processes, of course the parallel processes complete the work a lot faster than the sequential processes. Please note that the parallel processes that are depicted in Fig. 1 refer to pure parallelism, meaning that those processes execute in parallel and they have no interaction with each other (i.e. they don't have common data that they share, they don't send messages to each other, they don't share resources, they are pure parallel). Among the advantages of pure parallel execution are: it has good clarity, good correctness properties, not as good as the sequential process, but very good, and it has high performance, higher than sequential programming. The problem with pure parallelism is that it's almost always not realistic nor practical, meaning you won't find many real life scenarios where you can implement a system using pure parallel processes. Imagine any service on the web or not on the web which deals with several clients on the same time. It's very hard to imagine a service that doesn't use any form of synchronization, it doesn't use a common database, a common data (i.e. a database or some files or even data in the memory) that is shared between those clients. So most of the parallel processing systems that we encounter in real life scenarios are actually concurrent processing systems, not pure parallel ones. Concurrent systems is the third programming paradigm and they are similar to parallel systems, but they have these interaction points where the processes interact with each other either by exchanging data or by exchanging messages or using shared data like in a database. Actually, pure parallel systems are

so uncommon and so unusable in the real world that parallel systems are actually synonyms with concurrent systems in the real world, i.e. a parallel system in the real world is actually a concurrent system – it has synchronization points. For example, a system that sells tickets on a website has a database with those tickets, there might be two clients that try to buy the same ticket, so this is a synchronization point. The concurrent processing paradigm has low clarity and correctness meaning it's quite difficult to prove the program correctness, it's also harder to debug such system because consecutive execution of the program with the same input data might not give the same results due to the randomness introduced by other concurrent processes. The concurrent programming paradigm has a relatively high performance meaning it is more efficient more performant than the sequential execution, but less than the (pure) parallel execution because the synchronization points make one process wait for other ones in order to execute correctly. A correct execution referred to in the above lines means an execution of the program according to that program's specifications. In the context of our course, correct execution generally means that the data consistency is maintained after that execution of the program.

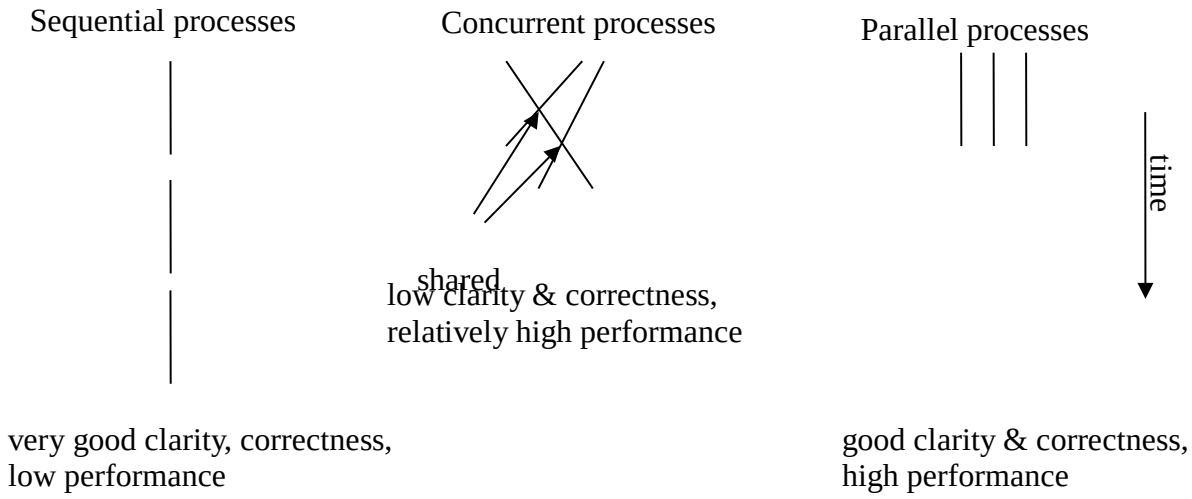


Fig. 1. Three paradigms: sequential programming, parallel programming, concurrent programming

Let's know take each programming paradigm and talk a little bit more about its characteristics. Sequential programming is the traditional programming paradigm, a serial algorithm is a serial list of instructions and each instruction is executed one at a time by one CPU even if you have several CPUs on your machine. In the case of multicore CPU machines, the sequential order dictates that instructions are executed one after the other even if they are executed on different cores which is just like executing the program using only one single CPU. The execution of a serial algorithm can have branches and can have loops/repetitions, but it still implies that you don't have two things executed or computed at the same time. It's a single thread of execution and control, it's appropriate for a Turing machine, it has stable programming methodologies: structured programming, data abstraction, object-oriented programming and design by contract, etc. There are strong formal model constructions for verifying the correctness of sequential

programs: Petri nets, deterministic automata and Labelled Transition Systems (which are similar to deterministic automata).
In parallel programming, a parallel algorithm decomposes a problem into independent tasks which can be executed in parallel. This is a natural consequence to the shift in the industry towards the multi-core architecture or multi-processor architecture. As you already know or as you have figured out the tendency or the direction in CPU development in the hardware industry is towards many more cores than towards a higher CPU frequency. In microprocessor development there is Moore' law (Gordon Moore is, together with Robert Noyce, co-founder of Intel) that says that every two years the density of transistors in a microprocessor doubles. The number of transistors on a CPU usually went until 2004 hand in hand with the clock frequency. The clock frequency of the CPU is the number of clock cycles per second and determines the computing power of the CPU and the system as a whole. If you don't know, there's a clock component in the CPU or on the computer's motherboard that generates uniform clock signal that is sent on the motherboard. This clock is used for synchronizing computer operations. Computer operations need to be synchronized, for example if the CPU reads something from the memory, it needs to send to a specific component (microchip) on the motherboard the command to read something from the memory. So, the CPU issues this command to the memory controller microchip on the motherboard and then the CPU waits on the data bus lines coming into the CPU for a specific amount of time and then it reads the data. This clock signals generated by the CPU or the motherboard are useful for this kind of operations. The CPU needs to know how much time it should wait until it starts reading data from the memory bus. If it waits too long, then it takes longer to read data from the memory which is not efficient, but if it waits less it could read only half of the data which is not good. If, for example, you want to use an integer number in a program, the integer occupies four bytes and if the CPU is very eager to read the data from the memory, it doesn't wait long enough, it only reads two bytes from the memory and it gives your application only two bytes of the memory where that 4-byte integer number is stored. Of course if you run this program, the execution will be completely flawed, you would get an incorrect execution. So everything, every single operation has fixed time bounds and is synchronized in a computer system. If this would not be so, then nothing would work on the computer system. The clock signals are generated in a computer system by several devices, one of them being the CPU and some other being the motherboard but, so different components use different clock signal. When two components communicate, eventually they use a unitary clock signal which is chosen as the one that beats slower of the clock signals of both components involved. Every component in the computer system is constructed so that every operation performed by each component will last an integer multiple of clock cycles. Now you can imagine that an addition operation takes one clock cycle and if we have two CPUs, one of them having a clock which emits one clock signal per second (equivalent to a frequency of 1 Hertz) and another CPU emitting two clock signals per second (equivalent to a frequency of 2 Hertz), of course the second CPU is twice as fast as the first one because we know the addition operation takes one clock cycle for both CPUs and on the second CPU one clock cycle lasts half a second, but on the first CPU a clock cycle lasts one second. Up to the year 2004, the driving force in the CPU industry was to increase the the CPU frequency. If you increase the CPU frequency by adding more and more transistors on the CPU by miniaturizing the transistors, the CPU would perform operations faster, but this leads to the frequency scaling problem which means that if you increase the number of transistors on the microprocessor and so increase its clock frequency, then you would use more electrical current/power; if you use more electrical current then the CPU generates more heat; if the CPU generates more heat, you need more powerful coolers in order to dissipate that extra heat; more

powerful coolers that dissipate more heat use more electrical current which generates heat at its turn… so it's a vicious circle and you get to a limit. And the microprocessor manufacturers actually got to a limit where you cannot shrink everything a lot more. This limit was hit in 2004 when Intel canceled the release of two of their CPUs and they shifted the company together with the whole industry from increasing the CPU frequency to the multicore CPU architecture and instead increasing the number of cores on a CPU. Those cores on a CPU can execute tasks in parallel. The hand in hand evolution of the CPU frequency with the number of transistors in the CPU (also Moore's law) are visible in Fig. 2.

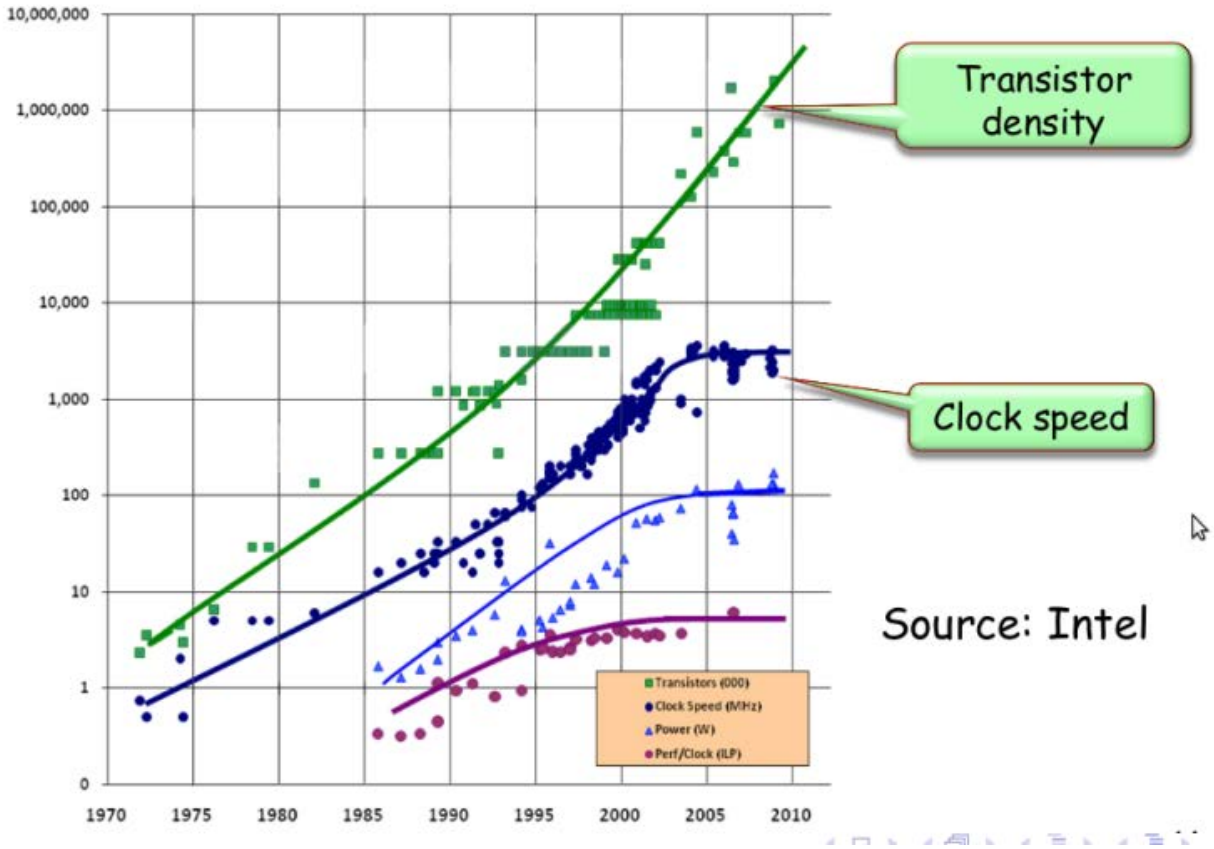


Fig. 2. evolution of the CPU frequency with the number of transistors in the CPU (also Moore's law)[1]

As far as I know the largest CPU frequency for commercial CPUs was around 4-5GHz. There are experiments with CPUs with a clock frequency of as much as 10GHz but these are not end consumer CPUs, they are cooled with liquid nitrogen. So here we conclude our side note about the frequency scaling problem in CPU design and the way the computer hardware industry shifted from a CPU architecture that kept increasing the transistor density in the CPU and the clock frequent of the CPU to an architecture that increases the number of cores on the CPU. And so parallel programming is a natural programming paradigm for this kind of CPU architecture, a multicore CPU architecture. But the problem is that there is no clear recipe for this kind of

[1] The image is from slides of B. Meyer and S. Nanz, ETH Zurich

parallel programming (actually, we are referring to concurrent programming and note pure parallel programming), as there are for sequential programming. Of course, it has increased performance with respect to sequential code, but as I have already told you, pure parallel applications are very rare in the real world, so most of the systems that exit in the real world are concurrent systems.

Concurrent systems are made of parallel processes which share data or share some state or exchange some state information). In concurrent systems there is no single thread of execution and control, they are more efficient than sequential programming, but it is harder to verify the correctness of the system and various problems / race conditions (like memory errors deadlock, live lock, etc.) can occur. I have already mentioned some important advantages of concurrent programming. Concurrency is better than sequential programming due to efficiency, i.e. it takes less time and the costs, the availability of services is higher being able to serve multiple request at the same time, and it has a high modeling power, meaning it describes systems that are inherently parallel and many systems in the real world are inherently concurrent. Some examples of systems that are inherently concurrent are booking agents (of seats, tickets) active at the same time, all sorts of selling services with a common storage. Since today's CPUs have several cores it is a natural way to write a program that is parallel from the beginning.

Now we introduce some terminology related to parallel and concurrent programming. Multiprocessing to use more than one core or more than one CPUs for processing at the same. In a multicore CPU processes are executed in parallel. But parallel execution existed even before we had multi-core CPUs or multi-processor systems. We had some form of parallel execution or multi-processing even when the computer system had only one core or one CPU. If we have a computer system with a single processing unit (core or CPU), then we still may give the illusion of having several tasks executed at the same time. For example, my current computer has an Intel core i5 CPU running at a base frequency of 3.2 GHz has four cores, but I have many more than 4 applications “running in parallel” on it: Microsoft Teams, a pdf reader, a text editor and many more processes running in the background (visible in Task Manager in Microsoft Windows and with `ps –aux` in Linux). They are all “running in parallel” on 4 CPU cores. So there can be a lot of applications running at the same time even on systems that only have one CPU. The way this is achieved is by what is called multitasking and the idea dates back to the UNIX time sharing operating system 1970 - the first time sharing operating system in history. The idea was that we have a scheduler which can be implemented either in the form of CPU interrupts or it is implemented in the system calls of the operating system, and this scheduler actually executes a bunch of instructions from one process and then the scheduler stops the execution, then moves to another process and executes a sequence of instructions from this one, and then it moves to executing another process and so on. Because this interleaving of multiple processes happens at a very fast rate (usually the execution of these instructions from one process takes less than one millisecond in modern operating systems), the scheduler of the operating system gives the illusion to the human end user that these processes are executed in parallel, but they aren't exactly executed in parallel. If we want our program to run faster on the next CPU, our program must be coded in such a way that it exploits more concurrency.

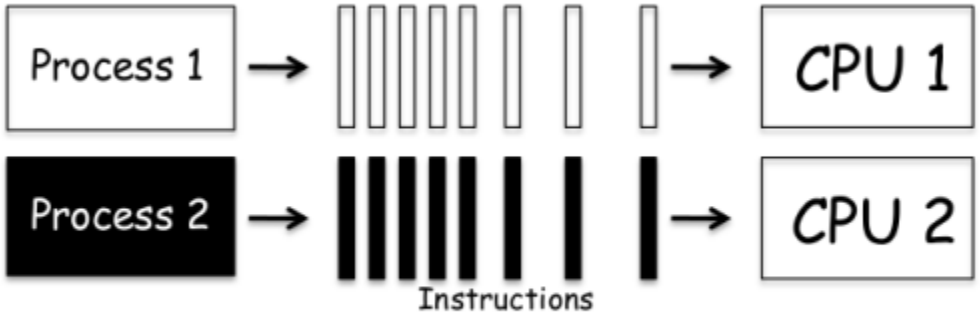


Fig. 2. Multiprocessing[2]

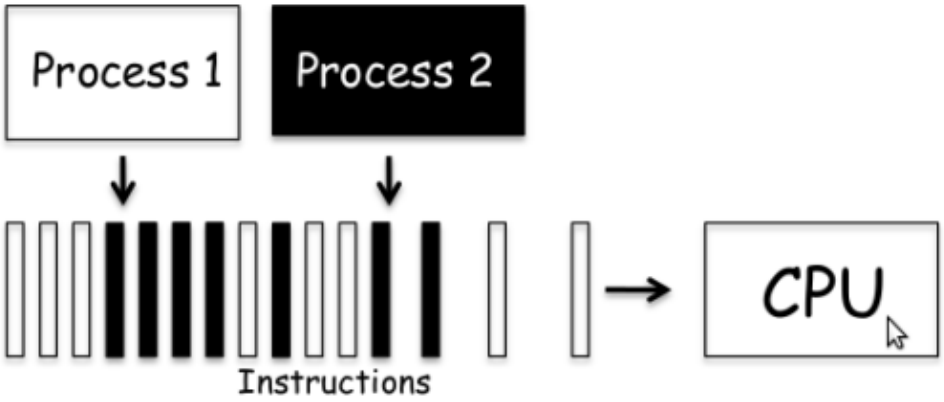


Fig. 3. Multitasking[3]

Now we might think that we have one CPU or we have a CPU with one core and we execute a program and then if we execute the same program on a different system with two cores CPU we should have double the efficiency. In other words, the program should execute twice as fast, but this is not the case. This is because there are always some parts of a program that cannot be parallelized, i.e. they cannot be executed in parallel. There is a law that defines this limit on the speedup performance. Amdahl's law defines this limit of the speedup we get by running the same program on multiple CPU systems. The speedup is defined as the performance increase of running a program on a multi-processor system with respect to the performance obtained when running the same program on a single CPU system. Mathematically, we define the speed up as

[2] The image is from slides of B. Meyer and S. Nanz, ETH Zurich
[3] The image is from slides of B. Meyer and S. Nanz, ETH Zurich

the division of the sequential execution time of the same program to the parallel execution time of the same program:

$$speedup = \frac{sequential\_execution\_time}{parallel\_execution\_time}$$

If the parallel execution time is equal to the sequential execution time then there's no benefit in executing things parallel. But if the parallel execution time is half of the sequential execution time, then we have a speed up of two because the same program executed twice as fast on a parallel architecture. Unfortunately, not everything can be parallelized so Amdahl's formula of the speedup when going from one processor to n processors is the following:

$$speedup = \frac{1}{1 - p + \frac{p}{n}}$$

where 1-p is the sequential part of the program, p is the parallel part of the program and n is the number of the CPUs or the number of cores. So let's take some numerical examples. Let's assume we want to see the speedup of the execution on 10 CPUs. If 60% of the program is parallelized and 40% can not be parallelized, the speed up according to Amdahl's law is 2.17. In other words, if we now have 10 CPUs we get an execution time that's only 2.17 times faster than the time obtained by running the same program on an architecture with only one CPU. But if a larger percent of the program can be parallelized, let's say 80%, with the 10 CPUs we get to a speedup of 3.57. If 90% of the program is parallelizable and 10% is sequential, with 10 CPUs we get a 5.26 speedup. And finally, an extreme case, if we have 99% of the code parallelizable and only 1% is sequential, then we obtain a speedup of 9.17 which close to 10.

Let's now dwell a little bit more into the details of parallel programming and talk about processor architectures for parallel computation. There is a classification of these architectures which is called Flynn's taxonomy which classifies parallel computer architectures depending on whether whether the CPU architecture is able to execute several instructions on the same time or is able to execute one instruction and apply it on several blocks of data:

- SISD – Single Instruction Single Data; if we have a CPU architecture that is able to execute a single instruction on a single data; this is just sequential execution
- SIMD – Single Instruction Multiple Data; if we have a CPU architecture that is able to execute a single instruction on multiple data (e.g. GPU programming and CPU extensions useful for graphic programming)
- MIMD – Multiple Instructions Multiple Data; if we have a CPU architecture that is able to execute multiple instructions that are applied to multiple data at the same time; some variance of MIMD is single program multiple data (SPMD) where all processors run the same program but at independent speeds and then we have multiple program multiple data (MPMD) where the we each processor runs a different program at independent speeds, usually you have some form of load distribution or job distribution policy like the manager distributes tasks and the workers return the result

We are trying to formally model concurrent processes and we will take two approaches, i.e. two

perspectives for this. The first perspective is depicted in Fig. 4 and we have several processors and there is a process that's running on each processor and all these processes share the same data. This shared data is depicted in this figure in the memory, but you can imagine that they share the same data that does not exist in the memory, but instead it exists on the hard disk drive. This common data can be stored on the filesystem in files or this data may be stored in a database which is eventually still a set of blocks of bytes that exist on the filesystem. This is the first perspective or the first approach that we will use where we have several processes that are executed in parallel and they share the common data. We get closer to the idea and the format of database systems where the database is just a bunch of data that exists in the blocks of the filesystem and this data is shared among concurrent processes which are also called transactions. So we will talk a lot about transactional systems.

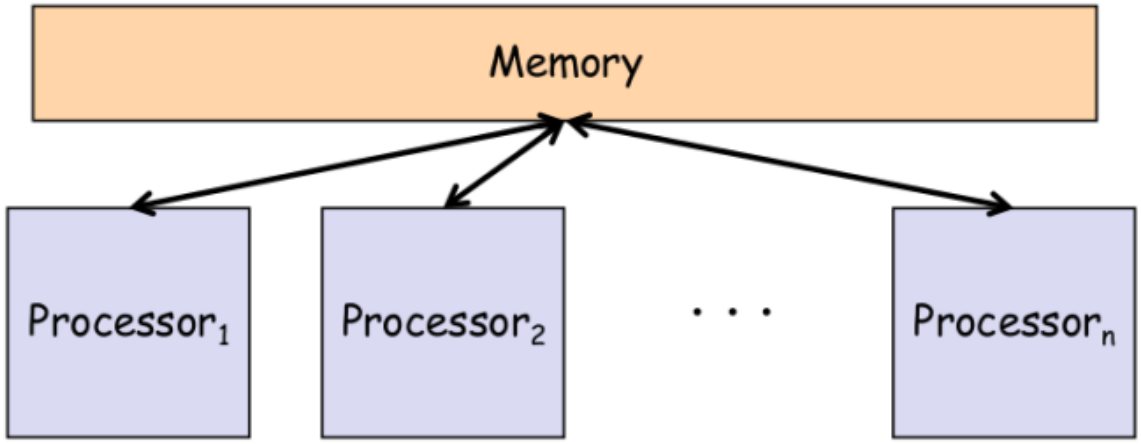


Fig. 4. The first perspective for modelling concurrent systems: multiple concurrent processes running on separate processors and sharing common data[4]

The second perspective is the one depicted in Fig. 5 where each process is running on a specific processor and has its own local memory, local data, but these processes exchange some messages, i.e. they send some messages from one to another. This is the second perspective that we will use in order to model formally concurrent systems. A specialization of this perspective is the well-known client-server computing paradigm where we have several clients and one server, each client and server have their own local memory, they run on separate processors the clients send requests (i.e. messages) to the server and they receive responses from the server. In this perspective we will introduce process algebra and related formalisms where we denote processes with symbols and we connect processes through abstract operation (similar to algebraic operations) and we try to reason about more complicated algebraic expressions resulting from this linkage. We will start in the next chapter with the formalism constructed based on this second perspective.

[4] The image is from slides of B. Meyer and S. Nanz, ETH Zurich

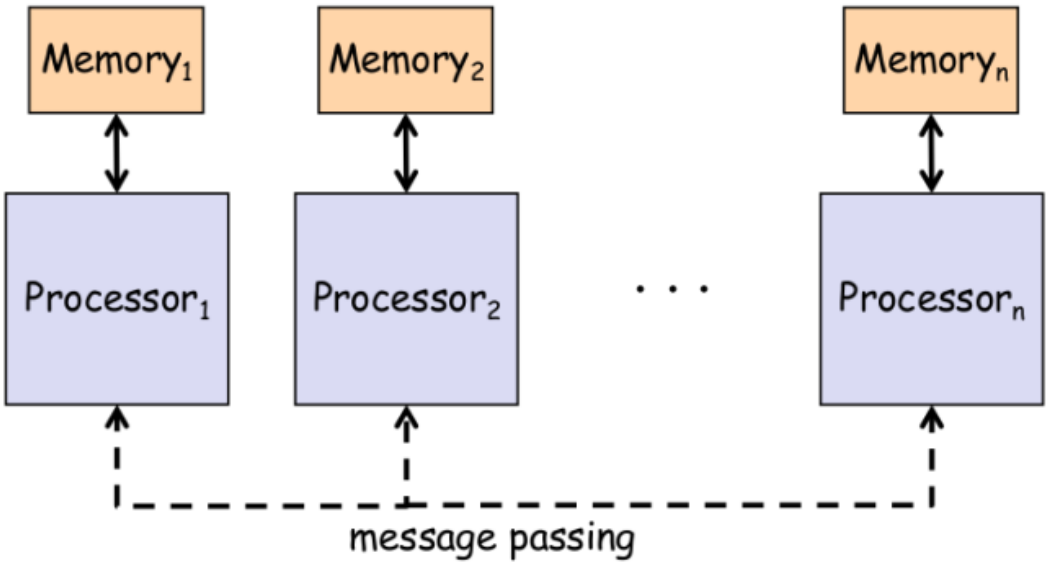


Fig. 5. The second perspective for modelling concurrent systems: multiple concurrent processes running on separate processors not sharing common data, but they exchange messages[5]

[5] The image is from slides of B. Meyer and S. Nanz, ETH Zurich

Communicating processes. Process algebra

Bibliography:
Luca Aceto, Kim G. Larsen, Anna Ingolfsdottir, Reactive Systems: Modelling, Specification and Verification, 2005.

In the previous section we presented to approaches for describing a system made from concurrent processes. The first was to consider that each concurrent process has its own memory and CPU and they do not share common data, but they communicate through messages sent from one concurrent process to the other. The second approach considered that the concurrent processes do not communicate with each other, but they share common data; this common data defines the state of the concurrent system. In this section we adopt the first approach where concurrent processes communicate through message exchange and present several formal models for describing such concurrent systems. All these formal models are united under the umbrella concept of **process algebra**.

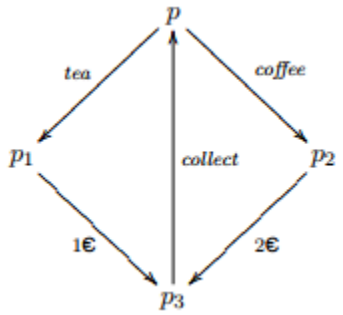
So, we should start first with the question: What is process algebra ? Well, **Algebra** is a part of Mathematics and is the study of mathematical symbols and the rules for manipulating these symbols (mostly through equations and inequations). A **Process** formally speaking is just a sequence of computations/operations. Hence, a **Process algebra** is an algebraic approach to the study of concurrent processes. The fundamental insight of process algebra formalism is Milner's observation that concurrent processes have an algebraic structure. Having 2 processes, P and Q, we can form a new process by combining P and Q sequentially or in parallel. The behavior of this new process would depends on that of P and Q and on the operation used to compose them. Formal description languages are algebraic: they consist of a collection of operations for building new process descriptions from existing ones.

Now that we have explained what is process algebra, we should ask ourselfs the next question: What is it good for ? Unlike sequential systems made from sequential processes for which we can verify the correction of the system in a more or less straight forward fashion, for parallel and concurrent systems verifying that the system is correct is a much complicated task and we need tools in order to perform this. One such tool is the concept of process algebra. Process algebra is good for formal modelling a system of concurrent processes by focusing on the messages exchanged between processes. We do this modeling of concurrent systems using process algebra, so that we can verify the correctness of the system with respect to given specifications either manually or by using automatic verification tools like the Concurrency Workbench and UPPAAL (e.g. check conditions like deadlock, livelock, starvation, reaching a final state).

Process algebra formalism examples are the following:

- Process algebra was coined by Bergstra & Klop in 1982.
- Hoare's CSP (Communicating Sequential Processes)
- Milner's CCS (Calculus of Communicating Systems)
- Bergstra & Klop's ACP (Algebra of Communicating Processes)
- π-calculus

In a process algebra, processes are considered to be independent units that evolve from one state to another. What triggers this evolution from one state to the other is the sending or the receipt of a message. In order for processes to be able to send or receive messages from/to other concurrent processes, these processes need to poses some channels which are just abstract constructs for sending and receiving messages. One way to describe the semantics of a system (i.e. the functionality of the system) is using graphical representation like Graph-based Transition systems. Two examples of such graph representations are Automata and Labeled Transition Systems (LTS).



In LTS – Labelled Transition Systems we have a graph and each vertex of the graph represents a state of the concurrent system. There are some start states and there are some final states. The life of the system is made from a sequence of transitions from one state to another. These transitions between states of the concurrent system are represented in LTS as edges and each edge has as label an action that causes that transition. Let's consider the simple example of a coffee vending machine: user inputs coins, selects tea or coffee and then collects the beverage. You can see the semantics of such a concurrent system in the figure above. The state p would be the initial/waiting/standby state. If the user chooses "tea" (by pressing a button of the machine), the system moves to a state p1 where it expects the user to input a coin (i.e. the price for the tee). Once the user input 1EUR, the system moves to state p3 where it delivers the beverage to the user and then automatically moves back to the initial state p. A similar reasoning explains the other path in the graph (the one that contains the action "coffee"). Another way of describing these transitions without using graphs is below:

$$p \xrightarrow{tea} p_1 \; and \; p \xrightarrow{coffee} p_2$$

$$p_1 \xrightarrow{1€} p_3 \; and \; p_1 \xrightarrow{2€} p_3$$

$$p_3 \xrightarrow{collect} p$$

An informal definition of LTS – Labelled Transition Systems that restates the above description would be the following:
A LTS is an labeled oriented graph consisting of a set of *states* (or *processes* or *configurations*), a set of *labels* (or *actions*), and a transition relation → describing changes in process states: if a process ***p*** can perform an action ***a*** and become a process ***p'***, we write $\boldsymbol{p \xrightarrow{a} p'}$**. A** state is considered as the start state.

A formal definition of LTS – Labelled Transition Systems is the following:

A *labelled transition system (LTS)* is a triple **(Proc, Act, {** $\overset{\mathbf{a}}{\rightarrow}$ **| a** $\in$ **Act})**, where:

- **Proc** is a set of *states*, ranged over by *s*;
- **Act** is a set of *actions*, ranged over by *a*;
- $\overset{a}{\rightarrow} \in Proc \times Proc$ is a *transition relation*, for every $a \in$ Act; we shall use the more suggestive notation $s \overset{a}{\rightarrow} s'$ instead of $(s, s') \in \overset{a}{\rightarrow}$

CCS (Calculus of Communicating Systems) is a process algebra introduced by Robin Milner in 1980, a formal language that can be used to formally describe the behavior of concurrent processes (through CCS expressions). LTSs describe the operational semantics of a CCS expression (what is the meaning of each part of a CCS expression and their interractions). We express CCS expressions into LTSs through SOS (Structural Operational Semantics):

- A CCS process will be a LTS state (vertex)
- A CCS transition will become a LTS action if it can be derived using a collection of syntax-driven rules (the SOS rules)

Calculus of Communicating Systems

If we consider a program, we can say that the program has a syntax (that describes the actions/operations) and a semantic (that describes the meaning of those operations). A programming language describes for example the syntax of the program. Labelled Transition Systems describe the semantics of a program. And CSS can be used to describe the syntax of LTS.

The basic principle of CSS is to define a few atomic processes (modelling the simplest process behaviour) and then define new composition operations (building more complex process behaviour from simpler ones). For example:
We can start with an atomic instruction: assignment (e.g. x:=2 and x:=x+2) and then introduce two new operators:

- sequential composition (*P*1; *P*2)
- parallel composition (*P*1 | *P*2)

Now for example the expression: (x:=1 | x:=2); x:=x+2; (x:=x-1 | x:=x+5) describes a new process build using the above tools.

A CCS process is a computing agent that may communicate with its environment via its nterface. The interface is a collection of communication ports/channels, together with an indication of whether they are used for input or output. We start with some basic constructs and using these, we can define what a CSS process is:

- *Nil* (or 0) process (the only atomic process)
- action prefixing (*a.P*)
- names and recursive definitions (def =)
- nondeterministic choice (+)

- parallel composition (|) - synchronous communication between two components = handshake synchronization
- restriction ($P \setminus L$)
- relabelling ($P[f\,]$)

Using the above constructs, we can define what a CSS process is.
Let A be a set of channel names (e.g. *tea*, *coffee* are channel names).
Let $L = A \cup \bar{A}$ be a set of labels where $\bar{A} = \{\bar{a} \mid a \in A\}$ (elements of A are called names and those of $\bar{A}$ are called co-names). By convention $\bar{\bar{a}} = a$.
Let $Act = L \cup \tau$ be the set of actions where τ is the internal or silent action (e.g. τ, *tea*, $\overline{coffee}$ are all actions).
Let K be a set of process names (constants).

Considering the above notations, we can define a CCS expression P as:

$P := K$ → process constants ($K \in$ K)
$\alpha.P$ → prefixing ($\alpha \in Act$)
$\sum_{i \in I} P_i$ → summation (I is an arbitrary index set)
$P_1 \mid P_2$ → parallel composition
$P \setminus$ L → restriction ($L \subseteq A$)
P[f] → relabelling ($f : Act \to Act$) such that

$$f(\tau) = \tau$$
$$f(\bar{a}) = \overline{f(a)}$$

The set of all terms generated by the abstract syntax is the set of CCS process expressions (and is denoted by P).

A CCS program is collection of defining equations of the form K def = P where $K \in K$ is a process constant and $P \in$ P is a CCS process expression.

Example of a CCS program: $R + a.$P | b.$Q \setminus L$

Transactional systems. The page computational model

Bibliography:

This text is largely based on the book
Weikum G., Vossen G., Transactional Information System: Theory, Algorithms, and Practice of Concurrency Control and Recovery, Kaufmann Morgan Publ. 2002.

We will try to describe concurrent systems using two approaches, one where we view a concurrent system as a set of concurrent processes and they all share common data or shared data and then there's the second approach where we will try to model concurrent systems where the system is formed by concurrent processes that operate independently of each other but they send messages to each other, so they communicate through messages. But right now we are in the first paradigm, i.e. a bunch of processes that run in parallel but they share common data.
The concurrency control theory presented here stands in a paradigm where we have several processes that get executed in parallel but they share some common data in the form of a database or in the form of files or in the form of the same memory data. We will give some concurrent application examples that we are trying to formally model, then we will eventually get to talk about transactions because we want the access to the data from those concurrent processes to follow some specific properties like the ACID properties (Atomicity, Consistency, Isolation and Durability) and then we will talk a little bit about the architecture of a database system, but this is only introductory talk and this is only in order to keep referring to a database for practical examples throughout the course. The whole picture of formal modeling of concurrent systems/processes is not directly linked to database systems, so we won't model only database systems, but we will generally prefer to give concrete, specific examples from the database world because database systems is just a field that we are more familiar with.
Now let's consider an example with the producer-consumer scenario depicted in the following code listing:

Producer	Consumer
	Read(x);
Read(x);	x--;
x++;	Write(x);
Write(x);	…
….	Read(x);
Read(x);	x--;
x++;	Write(x);
Write(x);	…
…	

The producer reads the data from the deposit (denoted with ‘x’) from the global memory, it increments it, so it produces something, and then it writes the deposit data to the global memory.
It does this in a loop. The consumer reads the deposit data from the global memory (i.e. ‘x’), it consumes it (i.e. decrements ‘x’) and then it writes x to the global memory location. The

consumer also consumes in a loop. The problem with executing this two programs/processes is the fact that starting from consistent data, let's say we have x equal to 1, after executing 10000 iterations for the producer and in the same time executing the consumer with 10000 iterations, we may get a final result final value of x that's different than 1 (i.e. the result we expect). We say that the final data is not consistent with respect to the initial data. In order to solve this problem we need to apply atomicity to each group of produce and consume operations. So we need these groups of three operations, (Read(x); x++; Write(x)) and (Read(x); x--; Write(x)) to happen in an atomic way meaning we this three operations group cannot be divided by some other operation. In other words, in transactional terminology, whenever these 3 operations in a group are executed, the three operations happen all of them or none of them, but in simple concurrent context we require they cannot be inter-spread with operations from the other actor (i.e. producer or consumer). You have probably seen during the Operating System class that for these simple applications, atomicity of the code would be enough to avoid consistency problems (actually, here atomicity assures consistency). But for more complicated applications, atomicity is not enough, we need more complex properties like consistency, isolation - so we need this program to run in a way that it doesn't need to know that there's another program called consumer and it also runs at the same time - and then there's also the property of durability that implies that once we finally written the result, it doesn't matter what happens after this, the data is still there. The atomicity, consistency, isolation and durability properties are the basic properties of transactions. Let's give some specific examples of applications that we will try to model with our formal theory. One example is the classical concurrent application in banking, i.e. moving funds/money from one account to another, the second one is an e-commerce application and the third one is a workflow application that plans reservations to a conference by registering to the conference, booking airplane seats and booking hotels rooms. So let's say we have a banking applications which is written in a combination of c/sql programming virtual language:

```
void main ( ) {
   int balance, accountid, amount;
   scanf (“%d %d”, &accountid, &amount);
   EXEC SQL Select Balance into :balance From Account
      Where Account_Id = :accountid;
   /* add amount (positive for debit, negative for credit) */
   balance = balance + amount;
   EXEC SQL Update Account Set Balance = :balance
      Where Account_Id = :accountid;
   EXEC SQL Commit Work;
}
```

What the above example does, is just deposit or withdraw the balance in an account from database table ‘Account’. It first reads two values from the standard input, one is the account id which and the other is the amount (the amount can also be negative for withdrawal operations). Then it selects the current balance from the database in the ‘balance’ local variable, updates the local variable ‘balance’ by adding ‘amount’ to it and then it updates the balance of the account from the database to the value of the local variable ‘balance’. In the below figure we have a possible execution of two concurrent processes executing the above functionality, they both operate on the same data, the first one withdrawals 50 money units from the account and the second one deposits 100 money units to the same account.

```
                    /* balance1=0, accountid.Balance=100, balance2=0 */
Select Balance Into :balance1          1
From Account
Where Account_Id = :accountid
                                                  Select Balance Into :balance2
                                       2          From Account
                                                  Where Account_Id = :accountid
                    /* balance1=100, accountid.Balance=100, balance2=100 */
balance1 = balance1-50                 3
                    /* balance1=50, accountid.Balance=100, balance2=100 */
                                       4          balance2 = balance2 +100
                    /* balance1=50, accountid.Balance=100, balance2=200 */
Update Account
Set Balance = :balance1                5
Where Account_Id = :accountid
                    /* balance1=50, accountid.Balance=50, balance2=200 */
                                                  Update Account
                                       6          Set Balance = :balance2
                                                  Where Account_Id = :accountid
                    /* balance1=50, accountid.Balance=200, balance2=200 */
```

In the above listing time runs vertically, is depicted by the red digits and the execution of the first process is on the left column and the execution of the second process is on the right column. At time 1 the first process selects the balance of the account into local variable balance1 and at time 2 we have the same select from process 2. So at the end of time 2 we have the value of $balance_1$ equal to 100, the actual value of the balance in the database equal to 100 and then we have the value of $balance_2$ equal to 100. At time 3 we subtract 50 units from $balance_1$, the balance in the database stays the same and the local variable $balance_2$ stays the same in process 2. At time 4 process 2 manages to add 100 money units to the value of local variable $balance_2$ that's why $balance_2$ is 200, the actual balance in the database is still 100 because we didn't do anything to the database and the value of the $balance_1$ local variable of process 1 is 50. At time 5 we actually update the database to set the balance to the value of $balance_1$ = 50 while the local variable $balance_2$ from process 2 stays 200. At time 6 the account is updated in the database by process 2 which adds 100 so we have the balance in the database equal to 200. Of course this is incorrect, this is not consistent because we started with 100 money units in the database and after subtracting 50 and adding another 100 we have a final value of 200. This is one problem that may happen in case of concurrent or parallel execution environment. Another problem happens for example if we have a process that transfers money from one account to another and let's assume that after it subtracts the amount from the first account and then there's a power surge and everything stops, preventing the process from depositing the amount of money into the second account. In order to avoid this problem, we need the atomicity property both the withdraw and deposit are executed together or none of them is executed.

Another e-commerce example is an Internet bookstore where we have clients connected to the to the bookstore server and they browse the books and they fill up shopping carts and then they buy the books. They may use a credit card and the application forwards payment to the customer bank and when payment is accepted there should be some real world actions that should happen: some

person should package the goods that were purchased and they should be sent by someone to the buyer's home address. This is still a concurrent problem, but this time it's more distributed because we have a distributed data storage, first there's the data storage of the merchant which is a list of books that it sells, then we have the data storage of the bank with the accounts from which money are withdrawn and then we have the data storage of the actual, real life books (i.e. the physical books that should exist in some deposit). So they are distributed and they all need to be synchronized across this e-commerce application.
Another example is an example of a workflow application with the following major use case: a person wants to go to a conference, first it selects the conference by checking the conference fee and it selects this conference if the fee is not larger than a specific amount (i.e. the conference fee is acceptable). A conference happens in a country in a specific city. After selecting that conference, the person then registers for this conference, but then he/she must also check the airport in order to see whether there are planes available, whether he can book a seat in a plane for that conference and then he must also check some hotels in the area in order to book a room in a hotel in that area. All these operations should happen in the scope of a single transaction, meaning that if the user has registered for the conference and has booked a flight for that conference, but then he/she is not able to book a hotel in the area, the system should rollback/cancel automatically the previously executed operations. Here we have a similar situation to the previous example where the data storage is distributed and handled by different authorities. The conference registration is maintained by a specific authority, the airplane data is maintained by a different authority and the hotel is managed by a third authority. So this is the third and final example of a concurrent application/system that we are trying to model. I have presented these applications from the business point of view, so I have presented what they actually do, just as example applications that we are trying to model. Of course, the course is about formal modeling of concurrent systems, so we won't deal with these specific applications, but they do serve only as examples.
All the above applications require the ACID properties for a transaction: they require atomicity which means all or nothing, so either all operations from transaction are executed successfully or none of them, consistency which means that if we start with consistent data, after executing our transaction (i.e. concurrent processes), we get also to consistent data, isolation which means that each process should not be aware of the fact that there are some other concurrent processes operating on the same data, and finally, durability which means that if we have committed something to the data storage, then those modifications should be persistent, i.e. they should last.

We require from the concurrent processes to access data in an ACID way meaning the processes should access the data in a way that preserves the properties Atomicity, Consistency, Isolation and Durability of the data. In this way we are very close to transactional systems in databases but we are not trying to model database systems, but instead we are trying to model transactional systems meaning we have systems with several concurrent processes and each process' data access should have the properties atomicity, consistency, isolation and durability. In this context we may also call one process a transaction, but the data on which those processes or transactions operate does not need to be the data in a database or it can be the data in the database but the database may not be a sql database. This data can also be files or memory, it can also be emails or something like that so it doesn't necessarily need to be stored in a database either relational database or non-sql database. Consequently, it doesn't really matter how the data is stored as this course is abstract, it is formal. Even if the theory in the course if general and formal, we will use

specific examples from the world of database systems so that we understand more precisely what we are talking about.
When presenting such specific, non-formal examples, we need some concepts from the world of database systems too, but otherwise we are not restricting our discussion to the world of database systems and I have given you some examples of concurrent applications/concurrent systems that we are trying to model in the previous sections. From a functional point of view, one of these specific examples was a banking application where you have accounts and you move money from one account to the other, and I have shown you some specific problems that may happen with these applications. Another example was an example of an e-commerce application, an application that sells books on the web and the final example was that of a workflow application where you have to plan a travel to a conference which implies registering to that conference, reserving the airplane ticket and booking a hotel room near the conference site. From an architectural point of view, the applications that we are trying to model are any tired or layered application, for example, we have a UI layer, then a business/middleware layer and finally a persistence layer (see Fig. 1). These layers can be part of a standalone application, or it cand be a distributed application organized using microservices (see Fig. 2).

Fig. 1. Layered architecture application

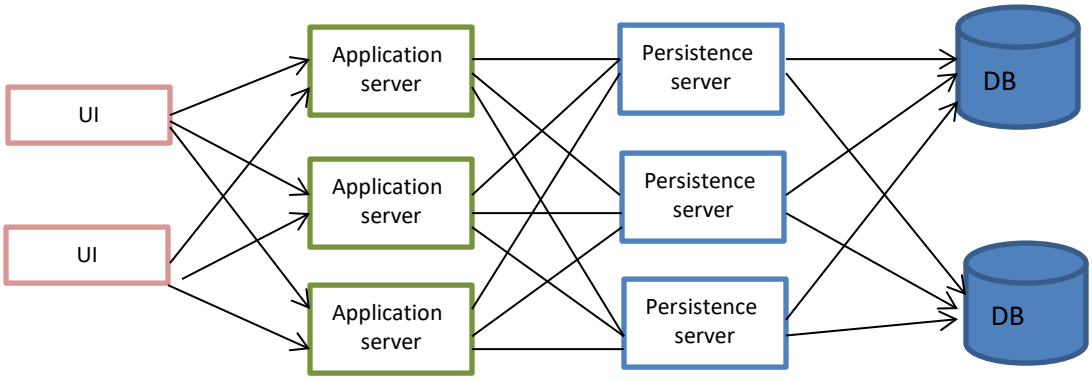


Fig. 2. Distributed application using redundant microservices for load balancing

The properties of a transaction are ACID. Atomicity means all or nothing, Consistency means if the transaction/process starts with consistent data, then after the execution of the transaction we should also have consistent data, Isolation means that a single transaction should not be aware that some other transactions execute in parallel with this one and Durability means once the data operations are executed, the data is saved persistently. In order to implement transactions in an application, we first need to somehow distinguish a transaction in the application code, i.e. we need a kind of marking of the place where the transaction starts (e.g. *start transaction*) and another mark where the transaction ends (e.g. *end transaction* or *commit*) in the application's code. All the operations in between the transaction start mark and end mark in the application's

code are considered to make part of that transaction. These start mark and end mark of transactions are programming languages dependent. Secondly, there should be something that needs to take care of the execution of transaction and that's something is the database server/system.

We will further briefly discuss about the general architecture of a database system just to make sure that we have the same common ground, i.e. we have some common terms that we will use when we show specific examples of our theory. I have previously said that our theory is not is about database systems, it's about more formal, general systems (including the field of general systems), but many times we will give specific examples of the statements and the definitions in the theory from the world of the relational database systems, because this is the application field where our theory is mostly used. Also, we are more familiar with these concepts from the world of database systems than other transactional systems and it is easier for us to understand the theory if we use examples from the relational database world.

So the database client applications send requests to the database server, usually those requests come in the form of a sequence of SQL language statements. When they arrive at the database server, the database server usually has a language and interface layer where it checks, just like a compiler, whether the requests from the client are correct (i.e. they are expressed correctly in the SQL language or some other interrogation language). Then we have the query decomposition and optimization layer which decomposes the whole transaction into separate queries. If we have a more complicated query, it decomposes it into sub queries or if we have a query that does a join maybe it decomposes the query into two queries and then performs the join. Then we have the query execution layer which executes each single query. In order to execute each single query we have the access layer which accesses the data. Actually this is just a mapping of the entities (i.e. table names, fields from tables) in the query into addresses on the hard disk drive, addresses which are then accessed using the storage layer. So the storage layer actually goes to the hard disk drive in specific blocks or pages of the hard disk drive and reads or writes the actual value in that record.

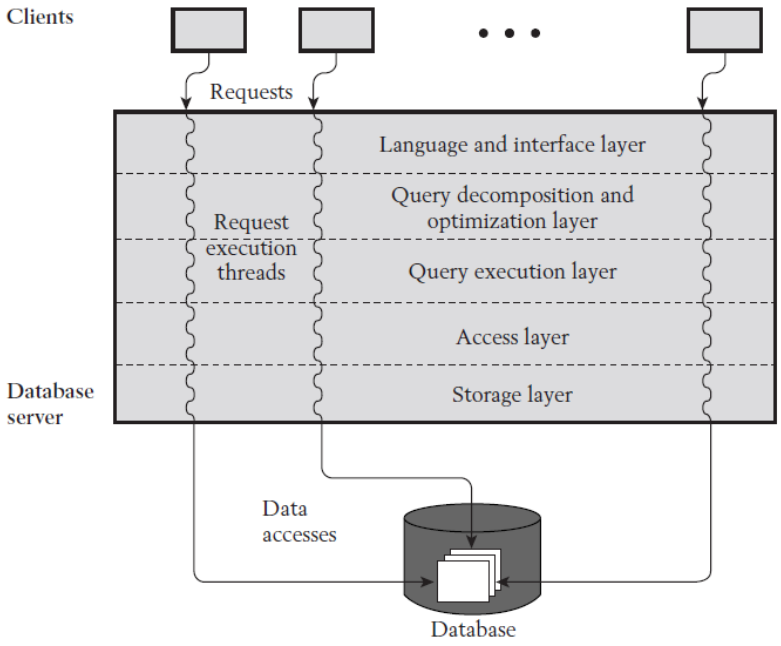


Fig. 3. The layers of a database server [1]

Now when talking about the actual storage of the data from a database system, this data is stored on blocks (or pages) of the hard disk. Maybe you remember from the operating systems course that most operating system call the transfer unit between the hard disk drive and the operating system a block or a page. This means that when you want to read one single byte one single character from a file on the hard disk drive, even if you want to read that single character, the operating system reads a block of data which is a number of bytes. This block size usually depends on the size of the partition, but it is it is larger than 1 MB of data so that 1MB of data is read into the memory and that specific character that you want to read is given to the application in the memory of the application. This is what it means that the hard disk is a block type device. Throughout this course we will also use the synonym page instead of a block to refer to a part of the hard disk drive. The data from a database system is stored in a block or in blocks on the filesystem. One database block of the filesystem usually has a structure like this: it starts with a header and then you have database records and at the end you have a slot array which specifies which slot from the current block are occupied and which ones are free and there are some pointers to the slots. You can see this in Fig. 4. Let's assume we have a record with a name an age and an address. Now this record can have a fixed length, but most of the time you would use variable length fields and in that case if the record doesn't fit in the current block then there would probably be inside the record a forward record ID which is just a pointer to the rest of the record in some other block. So this is how data from a database is usually stored on the filesystem.

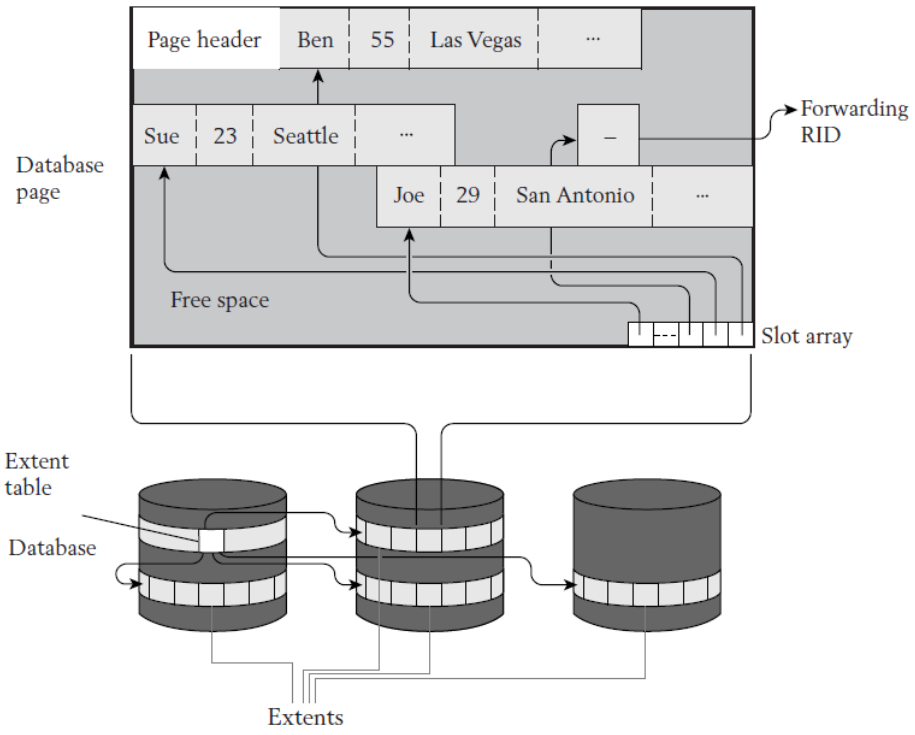


Fig. 4. A filesystem block belonging to a database [1]

When accessing the data in a database table, the database system can either do it iteratively/sequentially (i.e. start reading the blocks of a database table and then start parsing those blocks sequentially, on at a time and stop when you find the record you search for), but this is quite inefficient, so usually database systems have in place structures that allow us to search records quickly and these structures are indexes. Indexes are usually organized 2,3 trees or B+ trees (see Fig. 5). Each node of the index tree has specific values and the values from the left child/subtree of the current node has only values that are smaller (by some ordering relationship like alphabetic ordering) than all the values present in the current node. The second leftmost child/subtree of a node contains only values that are in between the first two leftmost values from the current node (e.g. in between 'Bob' and 'Eve' if the current node is the root). This rule is applied for all children/subtrees of the current node, so that the rightmost child/subtree of the current node contains only values that are strictly larger than all the values present in the current node. This rule is applied for all the nodes except the leaf nodes which do not have any children. At the leaf level, each value from a leaf node points to the actual record, i.e. is the address of the actual record which has this this value. Index structures are usually constructed for specific fields of the records. For example, in Fig. 5 you see an index on the name field of person records. So if the user query looks up the record that has the name 'Adam', the database system would go to the root of the index, then it would see that 'Adam' is smaller (alphabetically) than 'Bob', so it will follow the leftmost child of the root where it would find the 'Adam' value and it would return the actual address of the record having name equal to 'Adam' on the hard disk drive. This index structure allows us to look up a table based on a value of a specific field that is the field on which the index is computed (i.e. in this case our index is constructed based on the name field of the person record). And this index-based lookup is faster than the normal, sequential lookup.

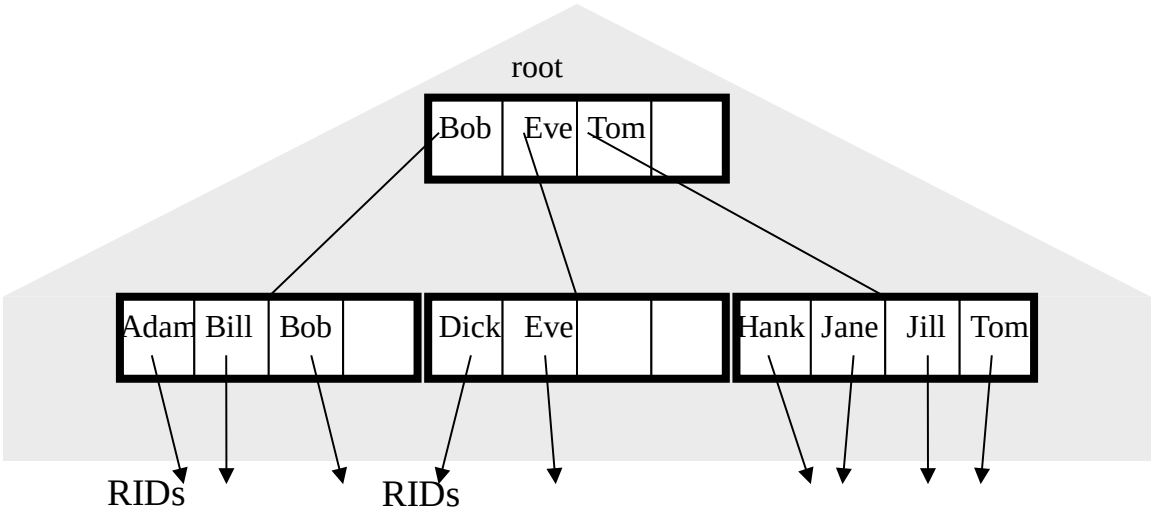


Fig. 5. A B+ tree corresponding to an index on the name field of person records [1]

The query decomposition and optimization layer of a database system takes a transaction as input and creates a query execution plan. Let's consider a query execution plan for the transaction containing only one query:

```
Select Name, City, Zipcode, Street From Person
Where Age < 30 And City = "Austin"
```

This query selects a bunch of fields like name, city, zipcode, street from the table Person, but then it has some filtering where age is smaller than 30 and city is equal to Austin. Two possible execution plans are presented in Fig. 6. One way of executing this query is the following: assuming we have an index on Age and another index on City, one can access the index on Age and then the index on City, get all the records that have the age smaller than 30 and get all the records that have the city equal to Austin and then intersect both result sets, and then use the record ID access layer in order to actually fetch the record values (because up to now we have only operated on the indexes), and finally, project the return values by selecting only the fields requested by the query (i.e. name, city, zipcode, street). A second alternative query execution plan is the following: let's assume we don't have the Age index, we only have the City index; so we use the City index in order to obtain all the records that have city equal to Austin, then we use the record ID access layer in order to actually fetch all the persons that are pointed by this index, and then we filter the persons meaning from the list of all the persons we got at this level, we filter out the persons that have an age larger or equal to 30 so we remain only with the records that have the age smaller than 30, and then we project those results on the required fields.

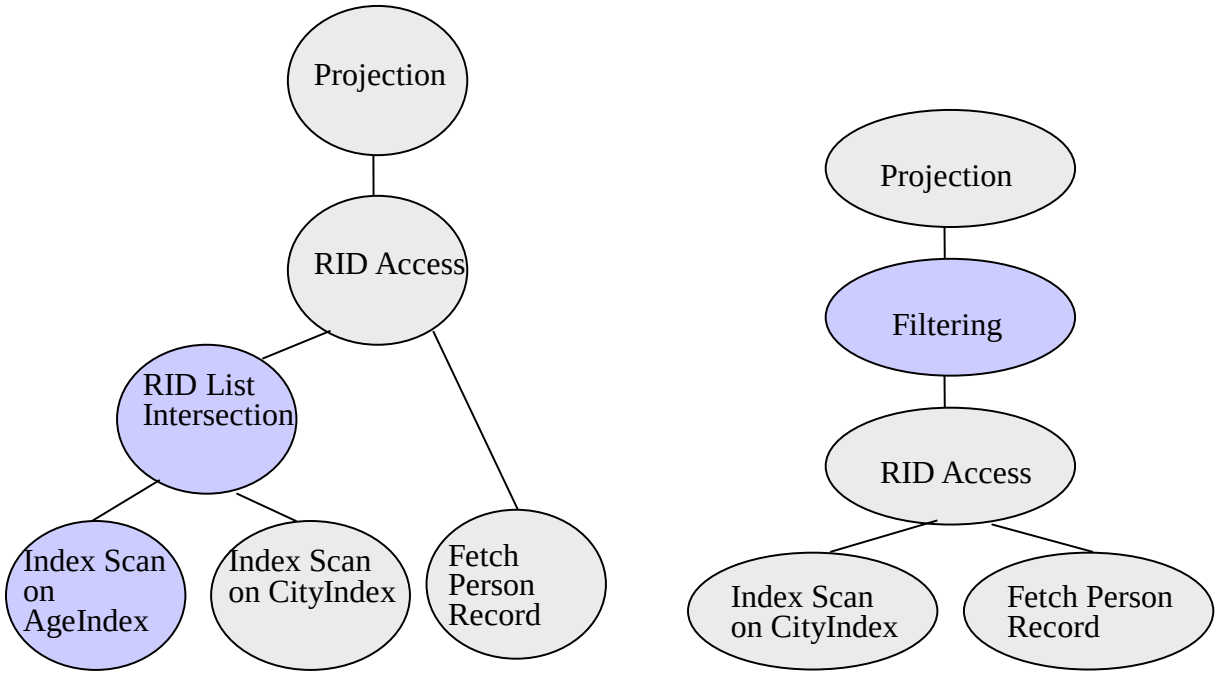


Fig. 6. Two possible query execution plans [1]

Having review all this background information, we are ready now to do the first steps in order to describe how transactional systems work. So the current paragraph can be seen as a roadmap for our quest of modelling transactional systems. A transactional system is just a component of a database system which gets at input a number of transactions and then decides when to execute each transaction, more specifically, it schedules for execution each operation from each transaction it receives as input. So this transactional system does a management of all the

transactions submitted to the transactional system. It has as input a bunch of transactions which arrive at the input, this input may be network sockets, pipes, queues or they may be read from files; usually it is sockets, so transactions arrive at the transactional system from an external source which is usually the client which runs on an external computer. So, the transactional system needs to decide when to execute each transaction, actually each operation from each transaction, and it must do this while making sure that it respects the ACID properties of each transaction. Our final goal is to describe how the transactional system works (i.e. actually, the scheduler of a transactional system) in the form of a pseudo-code like algorithm or an algorithm described in a programming language. In order to do this, we first need to consider what a transaction looks like from the perspective of a transactional system. Now if we are discussing specific examples like an e-commerce application, i.e. an application that sells books, we can say that from a transactional system point of view, transactions are just a set of operations that work on books, they either buy books (i.e. subtract values from book table records) or they load books into the store (i.e. add values to the book table records), but they also may change the accounts table and some logs table etc. If we are talking about specific applications then we know that the transaction is a bunch of operations that involves the tables used by that application like the books table or the account table, but we need to keep our discussion very generic and abstract so that we won't refer to specific applications when modelling the transactional system. One way to keep this high level of abstraction, is to consider a transaction is just a black box and we do not know what is inside that transaction, we just need to execute it. But if we go down this road and we consider that a transaction is just a black box, then the only way of executing transactions and making sure that ACID properties are provided, is to just schedule their execution sequentially, one transaction after the other. This is because if we execute the black box transactions in parallel, we have no way of assuring that the ACID properties are provided for each transaction because we don't know what happens inside the black box. This is, off course, not what we want because this kind of execution, i.e. the sequential execution, is the least performant one, so it has the least efficiency as you have seen in previous sections. Instead of sequential execution, we want to be able to somehow parallelize operations and in order to do this, we need to drop the assumption that transactions are just black boxes and we don't know what's inside the transaction. We need to somehow dwell inside transactions a little bit, but we must be careful not to talk about “books” and “accounts” and similar things, i.e. we must keep the discussion at a generic level. So we need to get into the details of a transaction, but we should strive at keeping the description quite formal, quite abstract.

We will try to describe what a transaction looks like here by modelling a transaction as being formed of a list of elementary operations which are considered atomic operations (i.e. we can not divide them further, the atomic operations are indivisible). You will see how we define those elementary operations in a couple of lines and then, after we have described what the transaction looks like using those elementary operations, then we can describe what a *schedule* looks like (i.e. a *schedule* would be the output of the scheduler of a transactional system). The input of a transactional system *scheduler* would be a list of transactions and the output would be a *schedule*. A *schedule* is just the list of operations of all transactions and an *order relationship* defined on that list of operations; the order relationship just specifies when each operation is to be executed by the scheduler with respect to the other operations from the list. The goal of the scheduler is to execute operations as many as possible in parallel because that is the most efficient, but as you will see, we cannot execute everything in parallel, so we need to somehow synchronize some operations (i.e. we need to execute a subset of those operations in sequential order) in order to provide ACID properties for all the transactions. So first we model a transaction as a list of

elementary operations, then we model the output of a transactional system scheduler (i.e. the schedule), and then we need to select out of all possible schedules that a scheduler can produce only those schedules that are correct; a *schedules is correct* if it assures the ACID properties of for all the transactions involved. In order to do this, we need to formally define what a correct schedule is and then we will describe scheduling algorithms, also called concurrency protocols, that produce correct schedules out of a list of input transactions. This is in summary the roadmap for describing how a transaction system works. So now let's move to the first task of our theory, i.e. modelling transactions as a list of elementary operations.

The page computational model

We will describe two computational models for transactions, the page model and the object model. In the page model we will define the transaction as a list of read and write operations together with an ordering of these operations.

Definition (Page Model Transaction): A **transaction** `t` is a partial order of steps (actions) of the form `r(x)` or `w(x)`, where `x ∈ D` and reads and writes as well as multiple writes applied to the same object are ordered. We write `t = (op, <)` for transaction `t` with operation set `op` and partial order `<`.

In the above definition, D is the domain of our model. The domain is a set of items D={x,y,z,…} where an item is a memory page (or a filesystem block) and it models either a record from a database table or a database table or just an entity that is stored in files, memory, database system etc. The read and write operations are also called "steps" of the model. The read operation, r(x), and the write operation, w(x), have a data symbol as parameter and they model a read operation of page x and a write operation of page x, respectively. The terms page and block are used interchangeably throughout the course. And the page or block can be a memory page or a filesystem block, because usually the memory is organized in pages in a computer system and the filesystem is usually formed from blocks of data. This is why this computational model is called the page model. So, in the page model each possible transaction is a set of partially ordered read and write operations on pages or blocks existing in the main memory or on the filesystem.
You should see that the page model is quite intuitive and quite consistent with the general understanding of transactions in an relational database. The page model is quite powerful when modelling transactions in a relational database. Usually a transaction in relational databases is formed by a sequence of SQL SELECT, INSERT, UPDATE and DELETE operations. Let's assume that our transaction contains the following SQL operation:

```
SELECT * from persons where sex = 'M'
```

This operation will select and return several records from the persons table. In order to do this, the transactional system will have to read those records' data from the filesystem from the pages/blocks where these records are stored. Therefore, it is very easy to see that an SQL Select operation at a lower level actually means a sequence of page read operations. This is also true for more complicated Select operations, maybe including JOIN clauses – they all translate to a set of filesystem pages/blocks read operations. An SQL Insert operation will similarly be equivalent to a set of page read and page write operations, because if a single record is added to the database, this means locating an empty slot from a free page (which is a read operation) and writing the record's fields in that page (i.e. a page write operation); it may also imply a couple of read

operations and then a write operation if there is an index on that table and we need to parse the B+ tree of the index (i.e. read operations) up to a leaf node which is updated (i.e. write operation). Similarly, a SQL Delete operation is equivalent to a set of page reads (for locating the pages where the records that are about to be deleted are stored on the filesystem) and write operations (to actually set free the slots from the pages where the records that will be deleted are stored). And finally, a SQL Update operation is clearly also equivalent to a set of page read and page write operations. So, as we have seen, the page computational model is pretty good at modelling SQL transactions in a relational database.

In the page model we view a transaction as a list of read or write operations, r(x) and w(x), where x is a data symbol. Although x is an abstract symbol, you can view this data symbol as a database table if you want to refer to a database systems so x can be a database table or a record from a database table or a file in a file storage or any entity in an entity system. So a transaction t is a partial order of steps or operations of the form read and write of x where read and writes as well as multiple writes applied to the same object are totally ordered. The partial order required by the definition of a transaction in the page model is very important. Mathematically, a partial order and a total order relationship is defined as:

Definition (Partial Order): Let A be an arbitrary set. A relation $R \in A \times A$ is a *partial order* on A if the following conditions hold for all elements $a, b, c \in A$:

1. $(a, a) \in R$ (reflexivity)
2. If $(a, b) \in R$ and $(b, a) \in R$ we have $a=b$ (antisymmetry)
3. If $(a, b) \in R$ and $(b, c) \in R$ we have $(a, c) \in R$ (transitivity)

A *total order relation* has the additional requirement that for any two distinct $a, b \in A$, either $(a, b) \in R$ or $(b, a) \in R$.

It is important to note that a total order relation has all the above properties of a partial order relationship, but also has an additional requirement meaning that for any distinct elements a and b either a and b are in the relation or b and a are in the relation, a requirement which is not required for a partial order relation. Usually we exemplify a total order relation with the smaller or equal relation" ($\leq$) defined on the set of natural numbers or integer numbers or real numbers. For any $x \in \mathbb{R}$, we have $x \leq x$ (i.e. reflexivity). For any $x,y \in \mathbb{R}$, if we have $x \leq y$ and $y \leq x$, then we have $x = y$ (antisymmetry). For any $x,y,z \in \mathbb{R}$, if we have $x \leq y$ and $y \leq z$, then we have $x \leq z$ (transitivity). In addition, we can always place the smaller or equal sign between two natural/integer/real numbers, i.e. either the first one is smaller or equal to the second one or the second one is smaller equal to the first one. That's why we say that this relation defined on the set of natural numbers or the set of integer numbers or real numbers is a total order relation as opposed to the same relation, but defined on sets of tuples like $\mathbb{N}x\mathbb{N}$, $\mathbb{Z}x\mathbb{Z}$, or $\mathbb{R}x\mathbb{R}$. So the same relation defined on tuples with more than one component has the properties of reflexivity, antisymmetry and transitivity, but you can not put any two tuples of the set in the relationship or its opposite relationship. For example, you can say about the following tuples defined on $\mathbb{N}x\mathbb{N}$ that $(1,2) \leq (3,4)$, but you can not pus the "smaller or equal" sign or its opposite, "greater or equal" sign between (1,2) and (2,1); we need a norm or a metric on this space which converts the tuple to a single numerical value. That is why $\leq$ is a partial order relationship defined on numbers from the sets $\mathbb{N}^n$, where $n \geq 2$.

The fact that we require partial order between operations in the transaction definition, not total order relation – actually, we will keep requiring partial orders in the following models of our theory – allows us to model also transactions that can execute operations in parallel. Take for

example a transaction that is formed entirely from 4 distinct SQL Select operations. These four Select operations can be executed in parallel for increased efficiency. In mathematical terms, this is equivalent to not placing a temporal "≤" relation between the 4 Select operations. But as, the transaction definition from above requires, whenever we have two operations on the same data symbol in the same transaction and at least one of them is a write operation, these two operations should be totally ordered in time. Imagine a transaction which has a deposit 10 money units and a withdraw 20 money units operations to the same account record (i.e. these operations are equivalent to a sequence of w(x) w(x) operations where "x" is the account record) in the Accounts database table. Also, assume that initially in that account there are 10 money units. If the deposit operation happens before the withdraw, the transaction succeeds and the balance of the account will be 0 in the end. But if the withdraw is executed first in the transaction and then the deposit, the transaction will fail because 20 money units can not be extracted from the account (i.e. the account balance is only 10 initially). So the result of the transaction will be ambiguous depending on which operation from the transaction is executed first – this is why the two operations, withdraw and deposit, needs to be totally ordered in time.

Below there are some examples of transactions:
t = r(x) w(y) w(z)
t = r(x) r(y) r(z) w(x)
t = r(x) w(x)
In the above examples, the time goes from left to right and the operations are totally ordered in each transaction. We usually use total ordered operations in our examples throughout the course just because it's easier to understand them and to represent the transactions, although the theory equally applies to partial ordered operations. Also, when we have several transactions, we will index the operation using the transaction index; for example, $r_1(x)$ is the read x operation of transaction 1; similarly, $w_2(y)$ is the write y operation of transaction 2.

In the page model we use the "r(x)" syntax in order to denote a read operation. The semantics (i.e. interpretation) of a read operation would be the assignment of the value that is read by the transaction, i.e. the x value, to a local variable of that transaction. If you think about that example where we with the accounts database, there was a select operation which selected the current balance of the account into a variable. So you usually do a select operation in a transaction because you want to read some data from the database and assign those values to some local variables (i.e. local to the transaction/application), so the semantics of a read step r(x) is just to read a value and assign that value to a local variable. In a similar way, the semantics of a write operation "w(x)" would be to compute a new value x which will be written in the database, but this new value x is equal to a function, let's call it a business function, that depends on all the local variables of this transaction (i.e. all the previous values read by this transaction from the database). So a regular SQL transaction would select a bunch of values from the database and based on these read values and some other external parameters from the user which are not included in our model, the transaction computes one or more new values which it then writes to the database.

Schedules and Histories. Correctness (serializability) criteria for schedules

Bibliography:

This text is largely based on the book

1. WeikumG., Vossen G., Transactional Information System: Theory, Algorithms, and Practice of Concurrency Control and Recovery, Kaufmann Morgan Publ. 2002.

In the previous chapter I have detailed the computational models used for modelling transactions, the page model and the object model. In the page model we describe transactions using read and write operations, where the read operation reads a memory page (or a filesystem block) and the write operation writes a memory page (or a filesystem block). And so we have described the transaction as a sequence of these read and write operation and there's also a partial order relationship between these operations in a transaction. We have also described another computational model, the object model which is like a tree of operations which has the base (leaf) layer made from read and write operations as the page model, but builds abstract layers on top of those read and write page operations in a similar way to how object-oriented programming abstracts complex types (i.e. classes) from simple ones (i.e. integer, float, string, array etc.). In other words, the object model tries to abstract a subset of read and write operations into a higher level method (or we can call it a function) and then it abstracts these methods into higher level methods (more abstract) and so on until we get to the root of the transactional tree, the highest abstraction level, which is the transaction itself. In the following chapters of this course we will mainly use the page model and forget about the object model.

Remember that the recipe we follow for describing (i.e. modelling) how transactional systems works is the following. We will first describe the input of the transactional system (well, actually we are not interested in the whole transactional system, we are only interested in the scheduling algorithm of the transactional system which is called the scheduler) and the input of the transactional system or of the scheduler is a set of transactions. Each transaction is described by the list of read or write operations of memory pages and the partial order of those operations within the transaction. So the input of a transactional scheduler which runs in a transactional system is a bunch of transactions we've already described in the previous chapter. Next, the output of a transaction system's scheduler is a schedule (i.e. in romanian it will be schedule='planificare' and scheduler='planificator'). So the scheduler receives transactions as input and produces a schedule at its output. Now we've seen how the input of a scheduler looks like, so we should describe how the output of a transaction scheduler should look like. After we do this, out of all possible schedules that a scheduler can produce, we need to select the ones that are correct (i.e. the schedules that assures the ACID properties of all transactions) so we need special correctness criteria for this. Finally, after we have done all this we can proceed to defining scheduling algorithms that produce correct schedules.

But before we do that, we should quickly run over a couple of, let's say canonical synchronization problems, that may happen in a concurrent system and when we describe/model transactional schedulers we want to make sure that the scheduler avoids this type of canonical problems. I'm

quite sure you have seen these canonical problems and talked about them in database courses that you previously followed, so these paragraphs are just a review of them.

The lost update canonical problem

In the *lost update problem* depicted in Fig. 1 we have two processes, P1 and P2, their operations are placed on the left and respectively the right column and on the central column we have the global time and the value of the data 'x' from the database. Let's assume initially the value of data 'x' in the database is equal to 100. At time t=1 P1 reads x into a local variable. Then at time t=2 P2 reads x into a local variable, at time t=3 P1 adds 100 units to its local variable and at times t=4 P2 adds 200 units to its local variable. Now at time t=5 the P1 process writes its local variable (x=200) into the database. Following, at time t=6 the second process, P2 writes its own local value (x=300) to the database so this 300 overrides the previously written 200 value of the first process. In other words the update performed at time t=5 by process P1 *is lost*. The consistent or the correct result in the database should have been 400 at the end of the execution, but instead we have 300 which is inconsistent.

P1	**Time**	**P2**
	/* x = 100 */	
r (x)	**1**	
	2	**r (x)**
x := x+100	**3**	
	4	**x := x+200**
w (x)	**5**	
	/* x = 200 */	
	6	**w (x)**
	/* x = 300 */	
update is lost		

Fig. 1. The Lost Update problem

We can describe this schedule (i.e. the intermixing of the P1's and P2's operations) like this:

$r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)$

So we have two transactions P1 is transaction 1, the red transaction and P2 is transaction 2, the blue transaction. The problem here is the interleaving of $w_1(x)$ with $w_2(x)$, because if the scheduler would have executed $r_2(x)$ right after $w_1(x)$ then there schedule will be just the serial execution, i.e. the red transaction followed by the blue one, so this schedule would have been correct. On the other hand, if $r_2(x)$ would have been executed before $r_1(x)$ and also $w_2(x)$ would have been executed before $r_1(x)$, then the execution would still have been correct with the same operations, but different interleavings.

The inconsistent read canonical problem

The *inconsistent read* problem is depicted in Fig. 2. Let's assume process P2 moves 10 money units from account *x* to account *y*. First, at times t=1,2 and 3, process P2 reads the value of x from the database, then removes 10 units from its local variable *x* and then writes the new value of *x* to the database. P2 then reads at times t=9, 10 and 11 reads the value of *y* from the database, it adds those 10 units to it and then writes the new value of *y* to the database. Process P1 at times t=4,..,8 reads the value of *x* and *y* from the database and computes their sum. Let's assume process P2 is just someone who moves money from one account to another, i.e. from account *x* to *y*, and both accounts belong to the same family. Meanwhile, process P1 can be another member of the family of P2 or it can be a banking monitoring process. Because P1 gets the balance of accounts *x* and *y* in between the money move operations it is clear that P1 will "see" a wrong/inconsistent sum. This inconsistent sis caused by the interleavings of operations of P1 and P2. We can describe the above schedule as:

$$r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)$$

If we use the same operations and the same schedule, but me we move the $r_1(y)$ operation at the end of the schedule we would not have any consistency problem. So the problem is the caused by the interleavings of the operations in time.

P1	**Time**	**P2**
	1	**r (x)**
	2	**x := x – 10**
	3	**w (x)**
s:= 0	**4**	
r (x)	**5**	
r (y)	**6**	
s:= s+x	**7**	
s:= s+ y	**8**	
	9	**r (y)**
	10	**y := y + 10**
	11	**w (y)**

Fig. 2. The inconsistent read problem

The dirty read canonical problem

The dirty read problem is depicted in Fig. 3. Here process P1 reads the value of *x*, then it adds 100 to it and then it writes *x* to the database at times t=1,2 and 3. At time t=4 P2 reads *x* from the database and subtracts 100 from it at times t=5. At time t=7, P2 writes the new *x* to the database, but meanwhile at time t=6 process P1 does something else and it doesn't work so it throws a failure and it restores the previous value of *x* that existed in the database before P1 started. In this case P2 relies on a value that is actually inconsistent. The dirty read problem can be described in the page model like this:

$$r_1(x)\ w_1(x)\ r_2(x)\ a_1\ w_2(x)$$

where a_1 is the abort operation from transaction 1.

P1	Time	P2
r (x)	1	
x := x + 100	2	
w (x)	3	
	4	r (x)
	5	x := x - 100
failure & rollback	6	
	7	w (x)

When we model the scheduler of the transactional system we want the scheduler to avoid this kind of problems. So let's assume that the input of the scheduler is a set of *n* transactions and each transaction is, of course, a list of page read and write operations ordered in a partial order relationship. The set of all possible schedules that the scheduler can produce for these *n* transactions is a subset of all the permutations of all the operations from the *n* transactions. This set is not exactly equal to the set of all possible permutations of all the operations of all transactions because there are some restrictions (implied by the ordering of operations in each transaction), but it is close to the set of all permutations of all the operations from transactions. The scheduler intermix the operations from those transactions the way it wants as long as it obeys the operation ordering of each transaction. Out of all possible schedules of a scheduler we need to filter out all the schedules that are incorrect meaning that those schedules do not respect the ACID properties of transactions. The transactional system scheduler receives operations from transactions at its input and when it receives an operation, the scheduler needs to schedule that operation so that in the end the final schedule is a correct one, meaning the schedule respects the ACID properties of the transactions involved. The problem here is that we cannot say about a schedule whether it is correct or not until the end of the execution when we have executed all the operations from all the transactions. In the end, we can say this schedule was correct meaning it respected the ACID properties of all the transactions involved or we can see that this schedule was incorrect. That is why we will talk about a history which is like a schedule produced by the scheduler looking at it in reverse order, after the schedule is completely executed. This is why we call it a history, because it already happened. So we will first look at histories and classify them as correct or incorrect. Another name for history is a complete schedule. Then we will talk about let's say partial schedules which are just prefixes of histories which are incomplete schedules, in other words a schedule that was executed, but there's still some operations from some transactions that still needs be executed/scheduled in this schedule. The formal definitions of a history and a schedule follow below.

Definition (Histories): Let $T=\{t_1, ..., t_n\}$ be a set of transactions, where each $t_i \in T$ has the form $t_i=(op_i, <_i)$ with op_i denoting the operations of t_i and $<_i$ their ordering.

A history for T is a pair $s=(op(s),<_s)$ such that the following statements are true:

(a) $op(s) \subseteq \cup_{i=1..n} op_i \cup \cup_{i=1..n} \{a_i, c_i\}$

(b) for all i, $1 \le i \le n$: $c_i \in op(s) \Leftrightarrow a_i \notin op(s)$

(c) $\cup_{i=1..n} <_i \subseteq <_s$

(d) for all i, $1 \le i \le n$, and all $p \in op_i$: $p <_s c_i$ or $p <_s a_i$

(e) for all p, q $\in$ op(s) s.t. at least one of them is a write and both access the same data item: $p <_s q$ or $q <_s p$

Definition (Schedule): A schedule is a prefix of a history.

So a history for a set of transactions is similar in definition to a transaction meaning it is a set of operations and also a partial order relation defined on those operations. We usually denote the schedule using the symbol *s* from schedule. The set of operations and the partial order relation of a schedule must comply with rules a)-e). Rule a) just specifies that the set of the operations of the schedule is included in the reunion of all the operation sets from all the transactions involved plus two special new operations for each transaction, a_i – abort of transaction t_i and c_i – commit of transaction t_i. The reason we have an inclusion in rule a) and not an equality is the fact that a transaction can execute an abort (a_i) or a commit (c_i), but not both. And this fact is emphasized in rule b) (i.e. a transaction can not abort and commit in the same schedule). So, if the transaction commits then it can not abort later in the schedule and if it aborts then the commit operation for that transaction should not be present in the schedule. Rule c) says that partial order relations of each transactions should be included in the partial order relation of the schedule. This means for example that if the partial order relation of transaction t_i specifies that in transaction t_i operation *p* comes before operation *q*, in the output of the scheduler these two operations stay in the same order, i.e. *p* happens before *q*; they cannot be reversed in the history of the scheduler because this would mean changing the transaction that the scheduler receives at its input, so the scheduler cannot change the semantics of a transaction. If a transaction does a select and then an update, then the scheduler should execute first the select and then the update, not the other way around. But the scheduler can also schedule some other operations from some other transaction in between that select and that update. The next rule, rule d) specifies that in a transaction the last operation that is scheduled is always either a commit or an abort; there can be no data operation from a transaction after the commit/abort of that transaction in the schedule. The last rule of a history, rule e), is very similar to a condition in the definition of a transaction in the page model and it says that for any two operations belonging to the schedule where at least one of them is a write and they both operate on the same data item, there should be a total order relation (either *p* before *q* or *q* before *p*). We cannot allow them to be scheduled in parallel in order to have a non-ambiguous result of the execution. All the other operations from the schedule can be in a partial order relation meaning there can be operations executed in parallel. This is similar to the rule we had in the definition of a transaction in the page model where we said that reads and writes as well as multiple writes applied to the same object are totally ordered. This pair of operations where both access the same data and at least one of them is a write is like a cornerstone of the whole theory (i.e. these operations are also called *conflicting operations*). And these are the most important operations that we need to totally order.

After we discussed about the history concept we can move to a schedule which is just a prefix of history or an *incomplete history*. Whereas a history is a *complete schedule*.

So a scheduler receives as input a set of transactions and each transaction is formed by a set of operations and a partial order relation defined on those operations of each transaction. But let's assume that all the relations are not total order relations and these order relations allow all the operations to be executed in parallel. Let's assume this simplified setup where all the transactions are formed only from read operations, i.e. they don't have write operations, and let's assume that the scheduler runs on only one CPU/core so it can only execute one operation at a time. Those read operations from each transaction can be all executed in parallel if enough CPU cores were available, because the partial order on the schedule allows it. But the scheduler can execute only one operation at the time because it has only one CPU/core, so how many possible distinct schedules can the scheduler produce ? If the total number of all operations from all transactions is *N*, then the scheduler can produce *N*! (N factorial) possible schedules (i.e. all the permutations of those *N* operations). Now of course if there are more CPUs (i.e. parallel execution is possible) then this number becomes less than *N* factorial because it is a combinatorial of *N* taken as the number of CPUs. And then if you also add some restrictions, meaning we also add to transactions some write operations and then we modify the partial order relation between the operations of a transaction so that not all operations are executed in parallel, in this case the scheduler can produce a number of possible schedules that is less than *N* factorial, but it is still a large number, i.e. there is a large number of possible schedules. Now we need to choose from all these possible schedules that can be produced by the scheduler, the ones that are correct (informally speaking we say that a schedule is correct if it assures the ACID properties of the transactions involved). Now, normally we can verify whether the schedule is correct or not at the end of the execution. After we have executed the whole schedule we can check whether the ACID properties were respected for each transaction and we can say this schedule was correct or not correct, but this is not the way a performance scheduler works in a transactional system. A performance scheduler in a transactional system cannot execute the operations and then at the end the scheduler says that ok, this execution was not correct, I should undo all the transactions and then try to execute them again in a different order. Because this is very inefficient. The way a scheduler works in real transactional system is the following. The scheduler takes each operation when it arrives, it doesn't wait until all the operations from all the transactions arrive, and for each operation it should decide whether to execute the operation right now in which case it should have enough guarantees that by executing that operation right now we obtain a correct schedule or the scheduler can decide that it cannot execute this operation right now or anywhere in the future because by executing this operation now or in the future this would break the ACID properties of a transaction involved; in this last case the scheduler of the transactional system just aborts the current operation together with the transaction to which this operation belongs. There can also be a third choice in which the scheduler can choose to postpone the execution of that operation when the operation arrives at the scheduler because the scheduler does not have enough data to decide whether executing that operation right now leads to a correct or incorrect schedule. So the scheduler postpones the execution of this operation until a later moment when it has more data and when it can make sure that by executing the operation, we get a correct schedule. In summary, a scheduler can only decide whether to execute an operation right now, to postpone its execution or to abort/cancel the execution of the operation together with the transaction to which the operation belongs.

So the scheduler needs to decide when a new operation arrives at the scheduler, after it has already executed some operations, if adding this new operation to the already produced schedule,

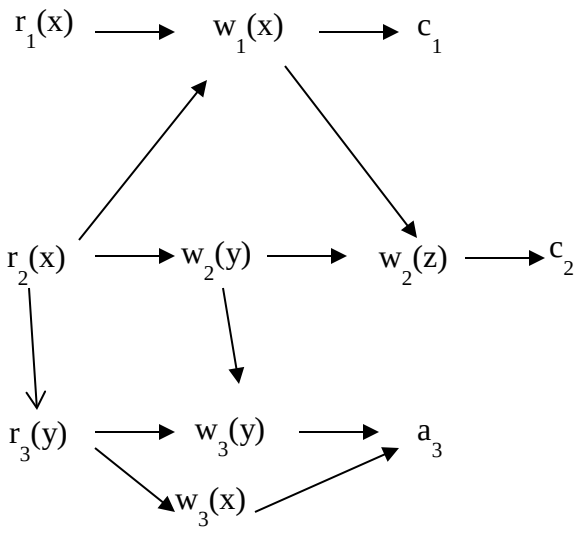
we still get a correct schedule. Informally speaking, a correct schedule is one in which the ACID properties of transactions are obeyed. Formally speaking, we will try to define the correct schedule by comparing the effects of the schedule in question to the effects of a serial schedule. A *serial schedule* is always correct, i.e. executing operations in series, one operation at the time, always respects the ACID properties of all the transactions involved. The definition of a *serial history* is the following:

Definition (Serial history):
A history *s* is *serial* if for any two transactions t_i and t_j in *s*, where i≠j, all operations from t_i are ordered in *s* before all operations from t_j or vice versa.

In other words, no operations exist in t_i and t_j that are intermixed together in a serial history. Either all the operations from t_i happen before all the operations from t_j in the history *s* or all the operations from t_j happen before all the operations from t_i in this serial schedule. We will formally define what is a correct history by comparing this history with a serial history. Actually, we will give examples of complete schedules or histories when exemplifying our theory, but the whole theory applies to incomplete schedules (i.e. prefixes of histories) also. *So a history is correct if it is equivalent with a serial history of the same transactions from the point of view of some equivalence criterion*. We will give several equivalence criteria between schedules/histories and using those equivalence criteria we can say that a random history is equivalent to a serial history. Since we know that the serial history is correct (i.e. it respects the ACID properties of the transactions involved), we can also conclude that the initial, random history is also correct because it is equivalent, according to an equivalence criterion, to this serial history. That is why it is very important to consider the serial history.

Examples of histories are presented in the next lines:
$s_1 = r_1(x)\ r_2(y)\ r_2(z)\ r_3(y)\ w_1(x)\ w_2(z)\ w_2(y)\ w_3(x)\ r_3(z)\ c_1\ c_2\ c_3$
$s_2 = r_1(x)\ r_2(y)\ r_2(z)\ r_3(y)\ w_1(x)\ w_2(z)\ w_1(y)\ a_1\ r_3(z)\ w_3(z)\ c_3\ c_2$
$s_3 = w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ a_2$

In the above examples, all histories use total order relations and the time runs horizontally, from left to right. So, in history s_1 the operation $r_1(x)$ comes before $r_2(y)$. Usually, we will use total order relations in our examples simply because they are easier to understand, but occasionally we will use histories with partial order relation between the operations like the one depicted in the following lines, but of course these are a little bit harder to read or to work with. In the following figure as you see there are parallel operations like $r_1(x)$ and $r_2(x)$. The arrows in the following drawing specify how two operations are ordered in time. So time does not run horizontally from left to right, but runs in the directions pointed by the arrows; for example, $r_2(x)$ must be executed before $w_1(x)$ and $w_2(y)$ must be executed before $w_2(z)$.



We now introduce some notations related to histories in the page model. *trans(s)* is the set of all transactions from schedule *s*. *commit(s)* is the set of all transactions that commit from schedule *s*. *abort(s)* is the set of all transactions that abort from schedule *s*. *active(s)* is the set of all transactions which are still active (not commited nor aborted) from schedule *s*. For example if we consider the schedule:

$s = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ a_3$

$trans(s) = \{t_1, t_2, t_3\}$

$commit(s) = \{t_1\}$

$abort(s) = \{t_3\}$

$active(s) = \{t_2\}$

Now, I have told you that we need to define equivalence criteria between schedules which splits the set of all possible schedules into equivalence classes. These equivalence classes should be sufficiently large meaning that the chosen equivalence criterion should say that many schedules are equivalent with a serial schedule, i.e. many schedules are correct so that it allows the scheduler to choose from those many possible correct schedules which one to execute. Thus choice will select out of all correct schedules the one that parallelizes operations the most – because it is the most efficient execution. So, we need to define when two schedules are equivalent. Let's consider an example. Let's say we have two transactions $t_1=r_1(x)$ and $t_2=r_2(x)$. We can have the following possible schedules:

$s_1 = r_1(x)\ r_2(x)$

$s_2 = r_2(x)\ r_1(x)$

$s_3 = r_1(x)$ → here $r_1(x)$ and $r_2(x)$ are executed in parallel

$r_2(x)$

From a specific point of view, all the above three executions are equivalent because the data in the database does not change at the end of each of the three schedules. In other words, all of them lead to the same result in the database.

Final-state serializability

The first equivalence criterion that we will introduce is the final state equivalence. *Two schedules are final-state equivalent if starting from the same data in the dataset (e.g. database), they get to*

the same values in the dataset at the end of their execution. So, final-state equivalence deals with the final data, the result, in the dataset – if this is the same for two schedules, if both schedules start from the same data in the dataset, then those two schedules are final-state equivalent. If we have for example, two transactions in our schedule and both transactions alter the balance of account 'a' from the database, if the initial balance of 'a' is 0 and after executing several schedules of the two transactions, the balance of account 'a' in the database is 100, then the execution of all these schedules is equivalent (according to the final-state equivalence criterion). However, will try to define this final state equality between those two schedules formally, not referring to specific numerical values or specific tables.
So, how can we formally say that two schedules are equivalent (i.e. their executions are equivalent)? One very natural definition as you have seen would be: if we have two schedules and we what to say about them that they are equivalent, we can say that if we start from the same data in the database and we run each of those two schedules and we get the same final result in the database for each schedule, we then can say that those two schedules are equivalent (because they get to the same final data in the database, starting from the same initial data). Because we require that the schedules' executions get to the same final data in the dataset, we call this criterion final-state equivalence. Together with this equivalence criterion, we will also define final-state serializability when specifying that a random schedule is final-state equivalent with a serial schedule (which means that this random schedule is final-state serializable). The above definitions were informal definitions (i.e. specified with free words). However, we need to define final-state equivalence formally using symbols and values like in algebra. Of course if we are talking about a specific transactional system that runs only transactions working on accounts and money, we can specify that very specifically, but this is not the case. We want our formal definition to be abstract and so, general. And the way we do this is by using the *Herbrand semantics*.

Herbrand semantics

The idea behind Herbrand semantics is to define how the final result (of a schedule) depends on all the operations executed by the scheduler, depends on the order in which those operations are scheduled and also on the initial symbols in the dataset. For now, let's consider that all the transactions commit and we will ignore abort operations. How can we formally define the final data (i.e. result) written in the dataset. Let's consider the following schedule:

$$s = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(x)\ r_3(y)\ r_1(y)\ w_1(y)\ w_2(z)\ w_3(z)\ c_1\ c_3\ c_2$$

This schedule works with the data *x*, *y* and *z*. The final data that will be written in the data set would be the final values written by this schedule in the data set. This means that the final value for *z* would be written by the $w_3(z)$ operation, the final value for *y* will be written by the $w_1(y)$ operation and the final value for *x* would be written by the $w_1(x)$ operation. Remember that in our examples time always runs from left to right, horizontally. In other words, the final state in the data set of a schedule would be defined by the last values written by that schedule in the dataset. In other words, it would be defined by the last write operations in that schedule for each data in the data set on which that schedule operates. In the above example, the final results would be defined by the operations $w_1(x)$, $w_1(y)$ and $w_3(z)$. So you can change the order of the operations in the above schedule as long as the last operation that writes *x* in the schedule is $w_1(x)$, the last operation that writes *y* is $w_1(y)$ and the last operation that writes *z* is $w_3(z)$. This way, you would get a different schedule that has the same final data as the initial schedule. Remember that the transactions we want to model usually read some data like the value of *x*, the value of *y* and

depending on those two values, this transaction computes a new value for *z* for example and it would write this new data to the dataset. But this final value of *z* which is written will not depend only on the values of *x* and *y* read by the transaction, it will also depend on the transactions (actually the write operations from those transactions) that wrote those *x* and *y* values to the dataset. And this is the idea of Herbrand semantics. The Herbrand semantics of read and write steps are defined below:

Definition (Herbrand semantics of read and write operations):
For a schedule s the *Herbrand semantics* H_s of steps $r_i(x)$, $w_i(x) \in op(s)$ is:

(i) $H_s[r_i(x)] := H_s[w_j(x)]$ where $w_j(x)$ is the last write on x in s before $r_i(x)$.

(ii) $H_s[w_i(x)] := f_{ix}(H_s[r_i(y_1)], ..., H_s[r_i(y_m)])$ where the $r_i(y_j)$, $1 \le j \le m$, are all read operations of t_i that occcur in s before $w_i(x)$ and f_{ix} is an uninterpreted m-ary function symbol.

You can also think about the Herbrand semantics of an operation as to "*the outcome/result of that operation*". Statement (i) above specifies that the outcome/semantics of a $r_i(x)$ operation from transaction *i* is equal to the semantics of the last write operation of *x* that was performed before this $r_i(x)$ operations. The outcome/semantics of $w_i(x)$ from transaction *i* will be equal to (i.e. will depend on) the Herbrand semantics of all read operations performed by transaction *i* before this write operation. And this will depend on all those read steps in a way that is unknown to the scheduler and is not specified by our model, but it is described (modelled) by this unknown function f_{ix}. Index *i* of the function specifies the transaction id and *x* specifies that this function is used for computing a new value for *x*. Considering the next schedule example:

$$s = r_1(x)\ r_2(z)\ r_3(x)\ w_2(x)\ w_1(y)\ r_3(y)\ r_1(z)\ w_1(x)\ w_2(z)\ w_3(z)\ c_1\ c_3\ c_2$$

the Herbrand semantics of the $w_3(z)$ operation (i.e. the actual value *z* that will be written) will depend on the values that were previously read by transaction 3 before this step, that is *x* and *y*. But we don't know the way in which this *z* value depends on the read operations, $r_3(x)$ and $r_3(y)$, so we define this dependency using an abstract notation $f_{3z}(H_s[r_3(x)], H_s[r_3(y)])$. $f_{3z}(..)$ is just an uninterpreted function symbol that is specific to the transaction and it defines the dependency relation between $w_3(z)$ and the previous two read operations performed by the same transaction, $r_3(x)$ and $r_3(y)$. The Herbrand semantics of the $r_3(y)$ operation will depend on the Herbrand semantics of $w_1(y)$ because $r_3(y)$ will actually read the value written by $w_1(y)$.
After defining the Herbrand semantics of read and write operations, we are ready to define the Herbrand semantics of the whole schedule. Not surprisingly, the Herbrand semantics of a schedule is defined by the Herbrand semantics of the final write operation from schedule *s* for each data involved in the schedule *s*. So, the Herbrand semantics of the schedule *s* for data *x* will be equal to the Herbrand semantics of the last write *x* operation from the schedule. The Herbrand semantics of the schedule *s* for data *y* will be equal to the Herbrand semantics of the last write *y* operation from the schedule.

Definition (Herbrand semantics of a schedule):
The *Herbrand semantics of a schedule* s is the mapping $H[s]: D \rightarrow HU$ defined by
$H[s](x) := H_s[w_i(x)]$, where $w_i(x)$ is the last operation from s writing x, for each $x \in D$.

In the above definition, D is the set of data (x, y, z …) and HU is the Herbrand universe which contains all the symbols introduced by Herbrand semantics for a schedule. It is formally defined below:

Definition (Herbrand Universe):
For data items D={x, y, z, ...} and transactions t_i, $1 \le i \le n$, the *Herbrand universe HU* is the smallest set of symbols such that:

(i) $f_{0x}( ) \in HU$ for each $x \in D$ where f_{0x} is a constant, and

(ii) if $w_i(x) \in op_i$ for some t_i, there are m read operations $r_i(y_1)$, ..., $r_i(y_m)$ that precede $w_i(x)$ in t_i, and v_1, ..., $v_m \in HU$, then $f_{ix}(v_1, ..., v_m) \in HU$.

In order to understand the purpose of the $f_{0x}( )$ type functions we should consider a simple history example: $s = r_1(x)\ r_2(y)\ r_1(y)\ w_2(x)\ w_1(y)\ c_1\ c_2$. If we try to compute $H_s[r_1(x)]$ which would be equal to the Herbrand semantics of the last write *x* operation before this operation, we would run into a problem because before this operation (i.e. $r_1(x)$) there is no write *x* operation (actually there is no operation at all before this one in the history *s*). This means that the above definition of Herbrand semantics of read and write operations would not be well formed. In order to correct this definition, we need to augment our schedule with let's call it a virtual transaction t_0 which just writes some initial values for the data *x* and *y* in the data set – we can say that this virtual transaction t_0 just initializes the database/data set. So the augmented history would be: $s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ r_1(y)\ w_2(x)\ w_1(y)\ c_1\ c_2$. And this allows us to properly define the Herbrand semantics of this whole history, so $H_s[r_1(x)] = f_{0x}$ and $H_s[r_2(y)] = f_{0y}$.

Let's now compute the Herbrand semantics for some example histories.
Ex.1. Let's say we have the schedule $s = r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$

We augument the previous schedule with a virtual transaction $t_0 = w_0(x)\ w_0(y)\ c_0$ which just writes some initial values for the data x and y in the dataset; the transaction t_0 initializes the dataset. So the new schedule becomes:

$s = w_0(x)\ w_0(y)\ c_0\ r_1(x)\ r_2(y)\ w_2(x)\ w_1(y)\ c_2\ c_1$

The Herbrand semantics of this schedule is:

$H[s](x) = H_s[w_2(x)] = f_{2x}(H_2[r_2(y)]) = f_{2x}(H_s[w_0(y)]) = f_{2x}(f_{0y}())$

$H[s](y) = H_s[w_1(y)] = f_{1y}(H_s[r_1(x)]) = f_{1y}(H_s[w_0(x)]) = f_{1y}(f_{0x}())$

$H[s] = \{ f_{2x}(f_{0y}()), f_{1y}(f_{0x}()) \}$

Observation: We can consider that the $t_2 = r_2(y)\ w_2(x)\ c_2$ transaction models the following transaction that buys 1 book:

```
bookid=1        // let's assume bookid 1 costs 100 lei
y = `Select balance from accounts where accountid=a`
if (y>100) {
        x = 1    // we buy 1 book
        Update books  set quantity = quantity-x where bookid=1
}
```

We can comment the above modelling of a book buy transaction in the following way. Normally a transaction functions like this: it reads a set of data from the database using a SELECT operation, then it uses some input external to the transaction for example from the user if the user

writes some input parameter in a text box on a web interface or something similar, and based on the read data and the external input it computes a new value which is written to the database. Of course, we can imagine some "dumb transactions" that do a bunch of SELECTs, they don't do anything with those values read from the database and the transaction just computes a whole new value which doesn't depend in any way from those values read from the database. But these are dumb transactions, you would not encounter this kind of transactions in the real world unless there's a bug in the software. We do not model these kind of transactions. Our transaction modelled above tries to buy a book for some specific user. A user would normally look in the application how much the book with the id 1 costs, the transaction would then select the money balance from the accounts table where the personal account of the user that is trying to buy the book is, and it checks whether the balance is larger than 100 which is the cost of this book and it updates the book table by decrementing the quantity for that specific book.

Ex. 2. Let's compute the Herbrand semantics for the following augumented history:
$s = w_0(x)\ w_0(y)\ w_0(z)\ c_0\ r_1(x)\ r_2(y)\ r_2(z)\ w_1(y)\ r_3(y)\ w_2(z)\ r_3(z)\ w_3(x)\ w_3(y)\ w_3(z)\ c_1\ c_2\ c_3$

$H[s](x) = H_s[w_3(x)] = f_{3x}(\ H_s[r_3(y)],\ H_s[r_3(z)]\ ) = f_{3x}(\ H_s[w_1(y)],\ H_s[w_2(z)]\ ) =$
$= f_{3x}\ (\ f_{1y}\ (H_s[r_1(x)],\ f_{2z}(\ H_s[r_2(y)],\ H_s[r_2(z)]\ ) =$
$= f_{3x}\ (\ f_{1y}\ (H_s[w_0(x)],\ f_{2z}(\ H_s[w_0(y)],\ H_s[w_0(z)]\ ) = f_{3x}\ (\ f_{1y}\ (f_{0x}(\ )),\ f_{2z}(\ f_{0y}(\ ),\ f_{0z}(\ ))\ )$

$H[s](y) = H_s[w_3(y)] = f_{3y}(\ H_s[r_3(y)],\ H_s[r_3(z)]\ ) = \ldots \quad = f_{3y}\ (\ f_{1y}\ (f_{0x}(\ )),\ f_{2z}(\ f_{0y}(\ ),\ f_{0z}(\ ))\ )$

$H[s](z) = H_s[w_3(z)] = f_{3z}(\ H_s[r_3(y)],\ H_s[r_3(z)]\ ) = \ldots \quad = f_{3z}\ (\ f_{1y}\ (f_{0x}(\ )),\ f_{2z}(\ f_{0y}(\ ),\ f_{0z}(\ ))\ )$

Now we are to formally state the first equivalence criterion, that is the *final state equivalence criterion*. Basically we say that two schedules s and s' are final state equivalent if the set of operations are the same in both schedules and the Herbrand semantics of both schedules (which actually expresses the final state of the schedule) is the same. If of course the set of operations from the two schedules is not the same, they cannot be equivalent.

Definition (Final-State Equivalence of schedules; definition based on Herbrand semantics):
Schedules s and s' are called *final-state equivalent* and we denote this by $s \approx_f s'$, if op(s)=op(s') and H[s]=H[s'].

Ex.1. Let's say we have the following two schedules:
$s = r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)$

$s' = r_3(z)\ w_3(z)\ r_2(y)\ r_2(x)\ w_2(z)\ r_1(x)\ w_1(y)\ w_1(x)$

We augument each schedule with the virtual transaction $t_0 = w_0(x)\ w_0(y)\ w_0(z)\ c_0$:

$s = w_0(x)\ w_0(y)\ w_0(z)\ c_0\ r_1(x)\ r_2(y)\ w_1(y)\ r_3(z)\ w_3(z)\ r_2(x)\ w_2(z)\ w_1(x)$

$s' = w_0(x)\ w_0(y)\ w_0(z)\ c_0\ r_3(z)\ w_3(z)\ r_2(y)\ r_2(x)\ w_2(z)\ r_1(x)\ w_1(y)\ w_1(x)$

We augmented our two schedules with a virtual transaction which is transaction t_0 which is just initializing the data set meaning it just writes the initial values of *x*, *y* and *z* before any operation from the transactions t_1, t_2 and t_3 start being executed.

Let's compute the Herbrand semantics for s:

$H[s](x) = H_s[w_1(x)] = f_{1x}(H_s[r_1(x)]) = f_{1x}(H_s[w_0(x)]) = f_{1x}(f_{0x}( ))$

$H[s](y) = H_s[w_1(y)] = f_{1y}(H_s[r_1(x)]) = f_{1y}(H_s[w_0(x)]) = f_{1y}(f_{0x}( ))$

$H[s](z) = H_s[w_2(z)] = f_{2z}(H_s[r_2(y)], H_s[r_2(x)]) = f_{2z}(H_s[w_0(y)], H_s[w_0(x)]) = f_{2z}(f_{0y}( ), f_{0x}( ))$

$H[s] = \{ f_{1x}(f_{0x}( )), f_{1y}(f_{0x}( )), f_{2z}(f_{0y}( ), f_{0x}( ))\}$

The Herbrand semantics of schedule *s* for data item *x*, H[s](x), is equal to the Herbrand semantics of the last write *x* operation from the schedule, $w_1(x)$. The Herbrand semantics of $w_1(x)$ would be equal, according to the definition, to an unknown function having as index the transaction number and data *x*, $f_{1x}(..)$, and having as parameters the Herbrand semantics of all the read operations from the transaction that happen before $w_1(x)$; we have only the $r_1(x)$ read operation. The Herbrand semantics of $r_1(x)$ is equal to the Herbrand semantics of the last write *x* operation that happened in the schedule before $r_1(x)$, and this is the $w_0(x)$ operation. And the Herbrand semantics of $w_0(x)$, being an operation from the virtual transaction t_0, is just the unknown function symbol $f_{0x}( )$, without any parameters (because transaction t_0 does not perform any read operations. We computed the Herbrand semantics of schedule *s* for data item *y* and *z*, H[s](y) and H[s](z), in a similar way.

Let's compute the Herbrand semantics for s':

$H[s'](x) = H_{s'}[w_1(x)] = f_{1x}(H_s[r_1(x)]) = f_{1x}(H_{s'}[w_0(x)]) = f_{1x}(f_{0x}( ))$

$H[s'](y) = H_{s'}[w_1(y)] = f_{1y}(H_s[r_1(x)]) = f_{1y}(H_{s'}[w_0(x)]) = f_{1y}(f_{0x}( ))$

$H[s'](z) = H_s[w_2(z)] = f_{2z}(H_s[r_2(y)], H_s[r_2(x)]) = f_{2z}(H_s[w_0(y)], H_s[w_0(x)]) = f_{2z}(f_{0y}( ), f_{0x}( ))$

$H[s'] = \{ f_{1x}(f_{0x}( )), f_{1y}(f_{0x}( )), f_{2z}(f_{0y}( ), f_{0x}( )) \}$

- ⇨ Because H[s] = H[s'] => $s \approx_f s'$ (i.e. s and s' are final-state equivalent)
- ⇨ Because s' is a serial schedule (i.e. $s'=t_3\ t_2\ t_1$) => s <u>is final-state serializable</u>

Ex. 2. Let's also give an example where the schedules are not final-state serializable. Let there be:

$s= r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$

$s'= r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$

The Herbrand semantics of these two schedules for the data item *y* is the following (we assume the two scheduled are augmented with the virtual transaction t_0):

$H[s](y) = H_s[w_2(y)] = f_{2y}(f_{0y}( ))$

$H[s'](y) = H_{s'}[w_2(y)] = f_{2y}(H_{s'}[r_2(y)]) = f_{2y}(H_{s'}[w_1(y)]) = f_{2y}(f_{1y}(H_{s'}[r_1(x)])) = f_{2y}(f_{1y}(f_{0x}( )))$

The Herbrand semantics of schedule *s* for data item *y*, H[s](y), is equal to the Herbrand semantics of the last write *y* operation, $w_2(y)$, and the Herbrand semantics of $w_2(y)$ is the unknown function $f_{2y}(..)$ of all the read steps performed by transaction 2 before $w_2(y)$, that is $r_2(y)$. The Herbrand semantics of $r_2(y)$ is equal to the Herbrand semantics of $w_0(y)$ which is equal to $f_{0y}( )$.
For the second schedule, the Herbrand semantics of schedule *s'* for data item *y*, H[s](y), is equal to the Herbrand semantics of the last write *y* operation, $w_2(y)$, and the Herbrand semantics of $w_2(y)$ is the unknown function $f_{2y}(..)$ of all the read steps performed by transaction 2 before $w_2(y)$, that is $r_2(y)$. The Herbrand semantics of $r_2(y)$ is equal to the Herbrand semantics of the last operation that wrote *y* before $r_2(y)$, that is $w_1(y)$. The Herbrand semantics of $w_1(y)$ is the unknown function $f_{1y}(..)$ of all the read steps performed by transaction 1 before $w_1(y)$, that is $r_1(x)$. The Herbrand semantics of $r_1(x)$ is equal to the Herbrand semantics of $w_0(x)$ which is equal to $f_{0x}( )$.

Because H[s](y) != H[s'](y) => s in not final state equivalent to s'.
If we look careful, we further see that *s'* is a serial schedule and *s* is an incorrect schedule because a lost update problem happens in *s*. So it wouldn't have been good if the schedule s would be equivalent to the serial schedule *s'* because in that case, the *s* schedule would be final state serializable, i.e. it would be a correct schedule, but we know it is not correct because the last update problem happens in schedule *s*. So in this regard, the final state equivalence criterion is a good one because it specifies that the above schedule *s* is not correct and this is what we wanted from the beginning - we wanted to avoid those problems of lost update, inconsistent read and dirty read (we want those problems not to happen in a correct schedule).

So this is final state equivalent. Now I am going to show you two additional ways of proving that two schedules are final state equivalent. The first one is harder to understand and the second one is easier.

<u>The Reads-from and Live-Reads-from relations</u>

We will define some mathematical relations that will provide us with an alternative construction to the whole Herbrand semantics theory. These relations are defined in the following lines. But before we can do this, we need to augment our schedules/histories with another virtual transaction besides t_0 and this is t_∞. We do this in order to have well defined relations. In opposition to t_0 (which contains write operations for all data items involved in the history), t_∞ contains read operations for all the data items involved in the history. For example if we have the schedule

$s = r_1(x)\ r_2(y)\ w_1(x)\ r_1(y)\ w_2(x)\ w_1(y)$

we augment this schedule with the t_0 and t_∞ transactions like this:

$s = w_0(x)\ w_0(y)\ c_0\ r1(x)\ r_2(y)\ w_1(x)\ r_1(y)\ w_2(x)\ w_1(y)\ r_\infty(x)\ r_\infty(y)\ c_\infty$

<u>Definition (Reads-from and Live-Reads-from Relation):</u>
Given a schedule s, extended with an initial and a final transaction, t_0 and t_∞, we can define the following concepts.
a) **$r_j(x)$ reads x in s from $w_i(x)$** if $w_i(x)$ is the last write on x such that $w_i(x) <_s r_j(x)$.
This statement just specifies that the read *x* operation from transaction j reads the value *x* that was previously written by transaction i.

b) The **reads-from relation** of s is defined as the set:

RF(s) := $\{(t_i, x, t_j) \mid$ an $r_j(x)$ reads x from a $w_i(x)\}$.

The Read-from relation is a set formed by tuples of two transactions and a data item and the property that the t_j transaction has a read $_x$ operation that reads the value of *x* written by a write operation from transaction t_i

c) Step p is **directly useful** for step q, denoted $p \rightarrow q$, if q reads from p, or p is a read step and q is a subsequent write step of the same transaction. $\rightarrow^*$ is the "**useful**" **relation** and it denotes the reflexive and transitive closure of $\rightarrow$.

This "directly useful" property is just a reformulation of the dependencies defined by the Herbrand semantics. Step p is **directly useful** for step q ($p \rightarrow q$) if either:

- q reads from p => this is just like saying that $H_s[r_j(x)] = H_s[w_i(x)]$ where $q=r_j(x)$ and $p=w_i(x)$
- or p is a read step and q is a subsequent write step of the same transaction => this is just like saying that $H_s[w_i(x)] = f_{ix}(H_s[r_i(x)], H_s[r_i(y)], \ldots)$ where $r_i(x)$ and $r_i(y)$ are read steps of transaction I performed before $w_i(x)$.

If we apply the "$\rightarrow$" relation recursively (in order to obtain the reflexive and transitive closure), we get to this long chains of dependencies between operations (as you will see below in the examples).

d) Step p is **alive** in s if it is useful for some step from t_∞, and **dead** otherwise.

e) The **live-reads-from relation** of s is defined as the set:

LRF(s) := $\{(t_i, x, t_j) \mid$ an alive $r_j(x)$ reads x from $w_i(x)\}$

The LRF set is just the RF set defined above, but we filter out from the RF set the tuples that contain *dead read* operations (a *dead read* operation is just an operation that does not sit in a dependency chain that ends with an operation from the t_∞ transaction.

Ex. 1. Let there be two schedules:

$s= r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)$

$s'= r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)$

We augument these 2 schedules with transactions t_0 and t_∞:

$t_0 = w_0(x)\ w_0(y)\ c_0$

$t_\infty = r_\infty(x)\ r_\infty(y)\ c_\infty$

$s= w_0(x)\ w_0(y)\ c_0\ \ r_1(x)\ r_2(y)\ w_1(y)\ w_2(y)\ \ r_\infty(x)\ r_\infty(y)\ c_\infty$

$s'=w_0(x)\ w_0(y)\ c_0\ \ r_1(x)\ w_1(y)\ r_2(y)\ w_2(y)\ \ r_\infty(x)\ r_\infty(y)\ c_\infty$

Let's define the read-from sets for these schedules. In order to compute RF(s) we need to find read operations in schedule s. The first read operation is $r_1(x)$. So transaction t_1 reads *x* from transaction t_0 (from operation $w_0(x)$). Then we go to the next read operation in the schedule, $r_2(y)$, and so transaction t_2 reads *y* from transaction t_0 (from operation $w_0(y)$). The next one would be $r_\infty(x)$ which reads the data x from transaction t_0, from the $w_0(x)$ operation. Finally, the last read operation in the schedule, $r_\infty(y)$, reads *y* from transaction t_2, from the operation $w_2(y)$. And this is how we compute RF(s). The RF(s') set is computed similarly for the s' schedule.

$RF(s) = \{(t_0, x, t_1), (t_0, y, t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

$RF(s') = \{(t_0, x, t_1), (t_1, y, t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

We have according to the $p \rightarrow q$ definition if:

- $q=r_i(x)$ and $p=w_j(x)$ and q reads x from p
- or $p=r_i(x)$ and $q=w_i(y)$ where p is scheduled before q (and belong to the same transaction, t_i)

Let's compute the useful relations for schedule s. We start parsing the schedule from left to right, in order and the first operation is $w_0(x)$. $w_0(x)$ is directly useful to a subsequent read operation that reads *x* from this $w_0(x)$. This is $r_1(x)$. $r_1(x)$ is directly useful to a subsequent write operation from the same transaction. So $r_1(x)$ is directly useful to $w_1(y)$. $w_1(y)$ is directly useful to a subsequent read operation that reads *y* from this operation, but there is no such operation because $r_\infty(y)$ reads *y* from $w_2(y)$. So, this directly useful chain ends with $w_1(y)$ because we cannot add any other operation in it, so this means that all the steps that we have in this chain are either dead, but there may be the case that some of them still appear on some other directly useful chain of alive steps, in which case they would be alive steps. Next we go again from left to right through the schedule and select the first operation that was not already placed on a directly useful chain and this operation is w0(y). And we construct a second useful chain. An so on..

$w_0(x) \rightarrow r_1(x) \rightarrow w_1(y)$
$w_0(y) \rightarrow r_2(y) \rightarrow w_2(y) \rightarrow r_\infty(y)$ => this means that all the steps from this chain are <u>alive steps</u>
$w_0(x) \rightarrow r_\infty(x)$: these steps are alive
So, the steps $r_1(x)$, $w_1(y)$ are <u>dead</u>.

Let's compute the useful relations for schedule s'. We apply the same reasoning as above.
$w_0(x) \rightarrow r_1(x) \rightarrow w_1(y) \rightarrow r_2(y) \rightarrow w_2(y) \rightarrow r_\infty(y)$ => this means that all the steps from this chain are <u>alive steps</u>
$w_0(y)$ => this step may be alive if it appears on an alive chain
$r_\infty(x)$: this step is alive
So step $w_0(y)$ is <u>dead</u>.

So, the live-reads-from sets are the read-from sets, RF(s) and RF(s'), from which we filter out tuples that imply a dead read operation. For example, in RF(s), only the tuple (t_0, x, t_1) contains a dead read step, i.e. $r_1(x)$, so we eliminate this tuple from RF(s). The Live-Read-from sets for both schedules are:
$LRF(s) = \{(t_0,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$
$LRF(s') = \{(t_0,x,t_1), (t_1,y,t_2), (t_0,x,t_\infty), (t_2,y,t_\infty)\}$

So after we did this exercise were we computed the RF and LRF sets for schedules, we can state another definition of final-state equivalence based on LRF sets. This definition follows.

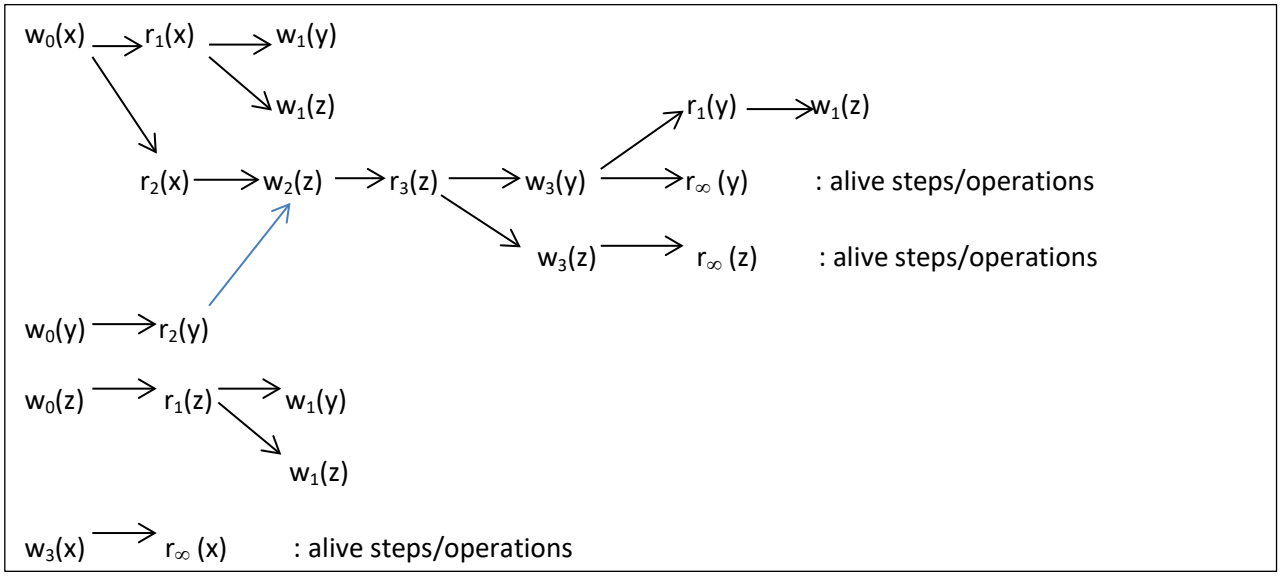
<u>Definition (Final-state equivalence of schedules; definition based on Live-Read-from relations):</u>
Having schedules s and s', if the set of operations from both schedules are identical (i.e. op(s)=op(s')) and LRF(s)=LRF(s'), then s is *final-state equivalent* to s' and we denote it by $s \approx_f s'$.

Using this new definition, we can see for example, that the two schedules from the example we did before (Ex.1) are not final-state equivalent because LRF(s) ≠ LRF(s'). Now let's give other example with more complicated directly useful chains.

Ex. 2. Being given the schedule s = $r_1(x)$ $r_2(y)$ $r_2(x)$ $r_1(z)$ $w_1(y)$ $w_2(z)$ $w_3(x)$ $r_3(z)$ $w_3(y)$ $r_1(y)$ $w_1(z)$ $w_3(z)$. Compute RF(s) and LRF(s).

Firsly, we augument the schedule with virtual transactions t_0 and t_∞:
s = $w_0(x)$ $w_0(y)$ $w_0(z)$ c_0 $r_1(x)$ $r_2(y)$ $r_2(x)$ $r_1(z)$ $w_1(y)$ $w_2(z)$ $w_3(x)$ $r_3(z)$ $w_3(y)$ $r_1(y)$ $w_1(z)$ $w_3(z)$ $r_\infty(x)$ $r_\infty(y)$ $r_\infty(z)$ c_∞

RF(s) = { (t_0, x, t_1), (t_0, y, t_2), (t_0, x, t_2), (t_0, z, t_1), (t_2, z, t_3), (t_3, y, t_1), (t_3, x, t_∞) , (t_3, y, t_∞) , (t_3, z, t_∞) }
We compute the useful and directly useful relations for schedule s:



In the above picture we can see more complicated directly useful (dependency) chains. Remember that these chains just encode Herbrand semantics in a different way. We can see here for example a chain bifurcation for $w_0(x)$ because $w_0(x)$ is directly useful for two operations, $r_1(x)$ and $r_2(x)$, because both read *x* from $w_0(x)$. We can also see another type of chain bifurcation at $r_1(x)$ that is directly useful for $w_1(y)$ and $w_1(z)$, both operations being subsequent write operation from the same transaction as $r_1(x)$. But we can also see another junction in $w_2(z)$, once it is derived from $r_2(x)$ (i.e. $r_2(x)$ is directly useful for $w_2(z)$) and then, it is also derived from $r_2(y)$ (i.e. $r_2(y)$ is directly useful to $w_2(z)$). We can also see in the above figure $w_1(z)$ being present in 3 chains.

Remember, a step/operation is alive if it is present at least on one alive chain (i.e. a chain that ends with a step from t_∞) and is dead otherwise. Hence, the next operations are dead: $r_1(x)$, $w_1(y)$, $w_1(z)$, $r_1(y)$, $w_0(z)$, $r_1(z)$.
The rest of the operations are alive.

Hence the Live-Read-from set is:

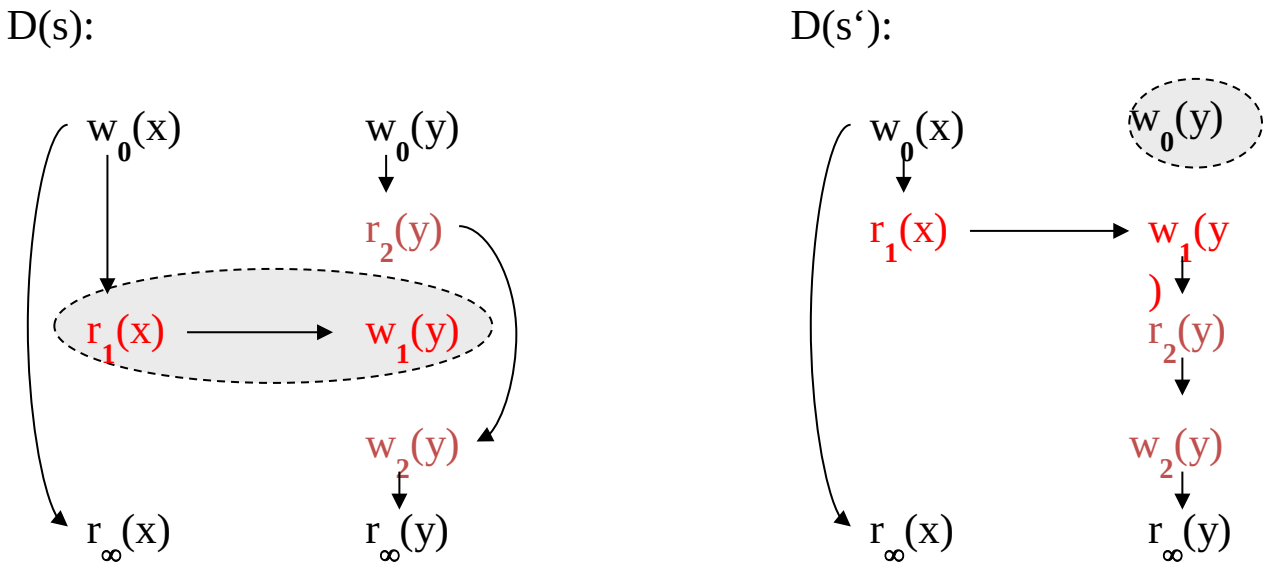
LRF(s) = { (t_0, y, t_2), (t_0, x, t_2), (t_2, z, t_3), (t_3, x, t_∞) , (t_3, y, t_∞) , (t_3, z, t_∞) }

So, we can compute that two schedules are final-state equivalent either by computing their Herbrand semantics and checking whether the Herbrand semantics for both of them is the same or we can compute the Live-Read-from set of both schedules and if the two sets are the same for both schedules then the schedules are final-state equivalent. So this is just another way of determining whether two schedules are final-state equivalent and then there is also a third way which involves computing this step graph or graph steps/operations which is the directed graph where the vertices are operations from the schedule and edges are of this form: we have an edge from *p* to *q* if *p* is directly useful to *q*. Out of this step graph we can derive a so-called reduced step graph which is the same graph, but from which we remove all vertices that correspond to dead steps and then we get this reduced step graph. If the Live-Read-from set of both schedules is the same, then also the reduced step graphs of both schedules is the same and the schedules are final-state serializable. The reverse is also true and everything is states in the following definition and theorem.

Definition (Final-state equivalence of schedules; definition based on reduced step graphs):
Having schedule s, we can build its *step graph* D(s)=(V,E) as a directed graph with vertices V:=op(s) and edges E:={(p,q) | p→q}, and then, we can derive from it the *reduced step graph* D_1(s) by removing all vertices that correspond to dead steps from D(s).
Having two schedules, if the set of operations from both schedules are identical (i.e. op(s)=op(s‘)) and their reduced step graphs are the same (i.e. $D_1(s)=D_1(s')$) , then s is *final-state equivalent* to s’ and we denote it by $s \approx_f s'$.

Let’s take the previous example and compute the reduced step graphs for both schedules:
s= $r_1(x)$ $r_2(y)$ $w_1(y)$ $w_2(y)$
s‘= $r_1(x)$ $w_1(y)$ $r_2(y)$ $w_2(y)$

After we augment the schedules with the t_0 and t_∞ and then we draw the step graph. The vertices are the operations from the schedule and the edges are linking directly useful operations. With the background gray we have the operations that are dead. So, if we remove the gray operations, we get the reduced step graphs. We can see that the reduced step graph of schedule s is not equal to the reduced step graph of schedule s’, for example because in D(s’) we have a edge from $w_0(x)$ to $r_1(x)$ and we don't have that in D(s) because here $r_1(x)$ is dead. So s is not final state equivalent to s’.



Example from [1]

Theorem. Having two schedules, s and s', with op(s) = op(s'), the following statements are all equivalent:

a) s is *final-state equivalent* to s', $s \approx_f s'$
b) H[s] = H[s'] (their Herbrand semantics is the same)
c) LRF(s) = LRF(s') (their Live-Read-from sets are equal)
d) $D_1(s) = D_1(s')$ (their reduced step graphs are equal)

Hence, the above theorem gives us three ways of proving that two schedules are final-state equivalent. Now we can finally state what final-state serializability means.

Definition (Final State Serializability):
A schedule s is *final state serializable* if there is a serial schedule s' such that $s \approx_f s'$. FSR denotes the class of all final-state serializable histories.

So we now have three ways of determining whether two schedules are final state equivalent. And we can determine this, i.e. whether two schedules are final state equivalent in polynomial time with respect to the length (i.e. the number of operations) from those two schedules. In other words, we can do this efficiently, we can decide whether two schedules are a final state equivalent in an efficient, and this is important because when the scheduler receives a new operation and the scheduler already produced a schedule so far, it needs to decide very fast whether by adding this new operation (i.e. by executing this new operation) to the already produced schedule, we still have a correct schedule (i.e. a serializable schedule). So this evaluation whether two schedules are final state equivalent should be done relatively fast. So we denote with FSR the class of all the histories that are final state serializable, meaning that there exists there one or more serial histories which are final state equivalent to all the histories from FSR.

Ex. 3. Let's consider the schedule s = $r_1(x)\ r_2(x)\ w_1(x)\ w_2(x)\ c_1\ c_2$. Decide whether s is final-state serializable (i.e. s belongs to FSR).

Let's use the LRF technique. We need to compute the read from set of the schedule. We first augment with transaction t_0 and transaction t_∞. Now we are able to compute the read from set and then the directly useful chains. The read from set is computed by taking all the read operations in the schedule, in order, from left to right, and seeing which operation writes the value that is read by a read operation. The read from set of schedule s is :

$RF(s) = \{ (t_0,x,t_1), (t_0,x,t_2), (t_2, x, t_\infty) \}$
Let's compute the "useful" relationship:
$w_0(x) \rightarrow r1(x) \rightarrow w_1(x)$: dead steps (except $w_0(x)$)
$w_0(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_\infty(x)$: alive steps
⇨ $LRF(s) = \{ (t_0,x,t_2), (t_2, x, t_\infty) \}$

You can notice above that that $w_0(x)$ is an alive operation because it appears on an alive chain, even if it appears on a dead chain also. In order to compute the LRF(s) set we need to remove tuples which contain dead steps from the RF(s), that is the tuple (t_0,x,t_1) because it contains the dead $r_1(x)$ step.
Now in order to decide whether schedule *s* is final state serializable or not, we need to compute the set of all possible serial schedules with the two transactions, t_1 and t_2. So let's consider first the serial schedule $t_1\ t_2$ which is formed by transaction t_1 followed by transaction t_2. We compute below for this schedule, the RF set, the directly useful chains and finally the LRF set.

a) Let's consider the serial schedule $s_{t1t2} = r_1(x)\ w_1(x)\ c_1\ \ r_2(x)\ w_2(x)\ c_2$
$RF(s_{t1t2}) = \{ (t_0,x,t_1), (t_1,x,t_2), (t_2,x,t_\infty)\}$
Let's compute the "useful" relationship for this schedule:
$w_0(x) \rightarrow r1(x) \rightarrow w_1(x) \rightarrow r_2(x) \rightarrow w_2(x) \rightarrow r_\infty(x)$: alive steps
So, we have $LRF(s_{t1t2}) = RF(s_{t1t2}) = \{ (t_0,x,t_1), (t_1,x,t_2), (t_2,x,t_\infty)\}$.
The live read from set of this schedule is equal to the read from set of the schedule because all the steps are alive (i.e. they are useful to a step from the infinity transaction).

Now let's consider the serial schedule $t_2\ t_1$ which is formed by transaction t_2 followed by transaction t_1. We compute below for this schedule, the RF set, the directly useful chains and finally the LRF set. The live read from set of this schedule will be equal to the read from set of the schedule because all the steps are alive (i.e. they are useful to a step from the infinity transaction).

b) Let's consider the serial schedule $s_{t2t1} = r_2(x)\ w_2(x)\ c_2\ \ r_1(x)\ w_1(x)\ c_1$
$RF(s_{t2t1}) = \{ (t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\}$
Let's compute the "useful" relationship for this schedule:
$w_0(x) \rightarrow r2(x) \rightarrow w_2(x) \rightarrow r_1(x) \rightarrow w_1(x) \rightarrow r_\infty(x)$: alive steps
So, we have $LRF(s_{t2t1}) = RF(s_{t2t1}) = \{ (t_0,x,t_2), (t_2,x,t_1), (t_1,x,t_\infty)\}$.

And now we should look at the Live Read From set of the schedule s and compare it to the Live Read From set of the first serial schedule we considered, t_1t_2 or compare it to the Live Read From

set of the second serial schedule, t_2t_1. If LRF(s) is equal to one of them, then it means the schedule s is final state serializable.

But $LRF(s) \neq LRF(s_{t1t2})$ and $LRF(s) \neq LRF(s_{t1t2})$. So schedule s is not final-state serializable.

This is of course good/correct and it happens because schedule s exemplifies the lost update problem (i.e. $w_2(x)$ overwrites the value of *x* written by $w_1(x)$). Which is a very good example why final-state serializability is a good correctness criterion because we wanted in the correctness criteria that we define to avoid the occurrence of those canonical problems, one of them being the lost update problem. So the fact that the schedule *s* is not final state serializable, it is just an argument that final state serializability is a good serializability criterion. But then we have the inconsistent read anomaly, if you remember another canonical problem we mentioned and this can be exemplified by two transactions, one of them moves some money from account *x* to account *y* and let's say this transaction is executed by one member of a family, this family has two accounts, *x* and *y*, and then we have a second transaction executed by another member of the same family or the bank itself which just checks the whole balance of both family accounts. If transaction 2 executed in between operations of transaction 1 like you see in the example below, the total account balance read by transaction 2 is incorrect because that money that was withdrawn from account *x* is not yet added to account *y* when transaction 2 get executed. So this is the inconsistent read anomaly.
$s = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$

If we compute the read-from set of the above history, we would get to (augmentation of the above history with the t_0 and t_∞ virtual transactions is omitted here, but it is, of course, assumed):
$RF(s) = \{(t_0,x,t_2), (t_2,x,t_1), (t_0,y,t_1), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$

Since $r_1(x)$ and $r_1(y)$ are dead steps (i.e. they do not lay on an alive dependency chain), we eliminate the (t_2,x,t_1) and (t_0,y,t_1) tuples from RF(s) and we get:
$LRF(s) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$

Then, we consider all possible serial combinations with transactions t_1 and t_2 and compute first the $RF(s_{t1\ t2})$ and $RF(s_{t2\ t1})$ for the serial histories $s_{t1\ t2}$ and $s_{t2\ t1}$. Afterwards, we eliminate from these read-from sets all the dead steps and we obtain the Live-Read-From sets for both serial histories:
$LRF(s_{t1\ t2}) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
$LRF(s_{t2\ t1}) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$

We can see from above that $LRF(s) = LRF(s_{t1\ t2}) = LRF(s_{t2\ t1})$, so this would mean that the schedule s is final-state equivalent to $s_{t1\ t2}$ and $s_{t2\ t1}$, hence the schedule s is final-state serializable (i.e. a correct schedule). But we already know that schedule s emphasizes the inconsistent read problem, so, obviously, this schedule can not be correct. Consequently, the final-state serializability criterion is not a good serializability criterion (because it permits the inconsistent read problem to happen). From the previous example, we can see that besides the final state of a history/schedule, something else like the semantics of the read operations should also matter in order for the equivalence/serializability criterion to be correct/good. And actually, this is what we will explore in other, subsequent equivalence criteria for schedules/histories. But besides the fact

that the final-state serializability criterion allows the occurrence of the inconsistent read problem, there is another argument why final-state serializability is not a good serializability criterion. And this is because evaluating whether a random history is final-state serializable or not is not computationally efficient, meaning you can not computed in a time that is polynomial with respect to the number of operations from the history. It is true that, as we said before, we *can compute whether two histories are final state equivalent or not in a polynomial time with respect to the number of steps in the histories*, in other words, we can do this in an efficient way (by using either Herbrans semantics or LRF or reduced step graphs tools), but *in order to verify that a history is final state serializable we need to verify this schedule against all possible serial histories of those transactions, in worst case scenario, and the number of all possible serial histories of those transactions is equal to the number of permutations of those transactions, so it is equal to the factorial of the number of transactions* – and this can not be evaluated in polynomial time. Consequently, these two reasons, determine us to head towards other serializability/equivalence criteria.

View serializability

It is not enough to consider the Herbrand semantics only of the last write operations, we also need to consider the Herbrand semantics of all the read steps and we do this by introducing a new equivalence criterion which is just final state equivalence plus an additional restriction. This is called *view equivalence* (and *view serializability* – if a schedule is view equivalent to a serial schedule then the initial schedule is view serializable). The definition of the view equivalence relation follows.

Definition (View Equivalence):
Being given two schedules, s and s‘, these schedules are *view equivalent*, denoted $s \approx_v s‘$, if the following statements are true:
a) op(s)=op(s‘)
b) H[s] = H[s‘]
c) $H_s[p] = H_{s‘}[p]$ for all (read or write) steps

The first two requirements of the definition are just the requirements from final state equivalence and for view equivalency we have added a third one which says that the Herbrand semantics of all the steps read or write should be the same in both schedules, s and s’. So view equivalency is the same as final state equivalency plus an additional requirement, so it is more restrictive than final state equivalence. This is required in order to remove the inconsistent read problem. Then we have some results similar to final state serializability which are stated in the following lines. The next theorem offers us alternative tools to prove that two schedules are view equivalent.

Theorem. Having two schedules, s and s’, with op(s) = op(s’), the following statements are all equivalent:
a) s is *view equivalent* to s’, $s \approx_v s‘$
b) RF(s)=RF(s‘) (read-from sets equality)
c) D(s)=D(s‘) (step graph equality)

Remember the read from sets, RF(s) is the one where we don't deal with the dead and alive steps. For final state equivalency we use Live-Read-From sets (LRF(s)) and for view equivalency we use only the Read-From set. We also don't need the directly useful relation and alive and dead steps for view equivalence. Hence, we can either prove that two schedules are view equivalent by following the definition with Herbrand semantics or we compare their read-from sets (RF(.)) or we can compute the step graphs of both schedules and check if they are the same. Remember that for final-state equivalence we computed the step graph with all the operations from all the transactions, but then we removed the dead steps and then we obtained the reduced step graphs. Then we compared the reduced step graphs for both schedules. Now we don't need to remove the dead steps, we just leave them in place and we just compare the step graphs of the first schedule to the step graph of the second schedule and if they are the same, then the two schedules are view equivalent. Hence view equivalency is just a further restriction of a final state equivalency.

As with the final state equivalence, view equivalence of two schedules can be efficiently decided in polynomial time with respect to the length of both schedules (by using the definition or the above theorem). We also call a schedule *view serializable* just similar to final state serializability if there exists a serial schedule to which this initial schedule is view equivalent.

Definition (View Serializability). A schedule s is *view serializable* if there exists a serial schedule s‘ such that $s \approx_v s'$. VSR denotes the class of all view-serializable histories.

Now the inconsistent read problem is rejected by the view serializable criterion because now we just need to compare the read-from sets as demonstrated by the following example.

Ex.1 We consider the previous mentioned inconsistent read history:
$s = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$

We compute the Read-From sets of s and of serial schedules s_{t1t2} and s_{t2t1}.
$RF(s) = \{(t_0,x,t_2), (t_2,x,t_1), (t_0,y,t_1), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
$RF(s_{t1\ t2}) = \{(t_0,x,t_1), (t_0,y,t_1), (t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$
$RF(s_{t2\ t1}) = \{(t_0,x,t_2), (t_0,y,t_2), (t_2,x,t_1), (t_2,y,t_1), (t_2,x,t_\infty), (t_2,y,t_\infty)\}$

We can notice that the read-from set of history s is different than the read-from set of serial schedule s_{t1t2} and it is different than the read-from set of serial schedule $s_{t2\ t1}$ which means that the history *s* is not VSR (i.e. it is not view serializable) - which is good because all along we wanted to reject this inconsistent read problem.

So view serializability rejects the inconsistent read problem which is good, but there is still a problem with view serializability, a problem that is also true for final-state serializability and that problem is deciding whether a given schedule belongs to VSR (or respectively to FSR). This problem is still NP complete, meaning we cannot compute this in polynomial time (with respect to the number of steps from the schedule), because although we can check whether a schedule is view equivalent to another schedule in a polynomial time, in order to check whether a schedule is view serializable, we need to compare this schedule, in worst case scenario, to all the permutations of serial schedules of the same transactions (i.e. N factorial serial schedules where

N is the number of transactions scheduled). Factorial is not polynomial time, it is above polynomial time. Hence, it is not efficient to check that a schedule is view serializable or is final state serializable. Because of this, view serializability is still not a good serializability criterion.

Some other facts about VSR and FSR are that VSR is included in FSR because by the definition, VSR means the requirements for FSR, plus a third restriction. Well actually, view equivalency implies an additional requirement with respect to final-state equivalence, so it is a more strict equivalence criterion, so that is why the VSR set is included into FSR. Also, if we have a history that doesn't contain dead steps, this means that the reduced step graph is equal to the step graph and the Live-Read-From set is equal to the Read-From set of that history, so this history belongs to VSR if and only if it belongs to FSR. This means that under these conditions, view serializability is the same as final-state serializability. We introduce now the definition of *monotony*.

Definition (Monotony of classes of histories). Let s be a schedule and $T \subseteq trans(s)$. $\Pi_T(s)$ denotes the projection of s onto T. A class E of histories is called *monotone* if the following holds:

if *s* is in E, then $\Pi_T(s)$ is in E for each $T \subseteq trans(s)$.

VSR is not monotone.

$\Pi_T(s)$, the projection of *s* onto *T* is similar to the projection definition in the world of databases. You have talked about projection when you have talked about SQL and relational algebra at database courses, so the projection of *s* onto a subset of transactions like $\{t_1, t_2\}$ would be just the schedule *s* from which we eliminate all the operations which do not belong to transactions t_1 and t_2. The "E" from a class E of histories from the above definition can be FSR or VSR or CSR, as we will learn later, or some other equivalence class of histories. The definition says that a class of histories is monotone if it has the property that if *s* is a schedule from this class, then the projection of *s* onto a subset of transactions is also in this class, and this is true for each subset of transactions. We can see that VSR is not a monotone class in the following example.

Ex. 1. We prove that VSR is not monotone through a counter-example.

Let's consider the schedule $s = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1\ w_3(x)\ w_3(y)\ c_3$.
$RF(s) = \{ (t_3,x, t_\infty), (t_3,y, t_\infty) \}$
Normally, we should verify if the schedule *s* is view equivalent to the serial histories: $t_1t_2t_3$, $t_1t_3t_2$, $t_2t_1t_3$, $t_2t_3t_1$, $t_3t_1t_2$, $t_3t_2t_1$. We just choose one history from this list and hope the Read-From set of that history is equal to RF(s). But if we look more carefully at the RF(s) set above we see that the operations that write the final value for *x* and *y* in the dataset are operations from the t_3 transaction. Hence, we just need t_3 in the end of the schedule, so as long as the last write operations of *x* and *y* in the schedule are from t_3, it doesn't matter the order of the other operations.

Let's consider the history $t_1t_2t_3 = w_1(x)w_1(y)c_1\ w_2(x)w_2(y)c_2\ w_3(x)w_3(y)c_3$.
$RF(t_1t_2t_3) = \{ (t_3,x, t_\infty), (t_3,y, t_\infty) \}$. So s is view serializable because $RF(s) = RF(t_1t_2t_3)$.

We now consider the projection of schedule s onto the set $\{t_1, t_2\}$, $\Pi_{\{t1, t2\}}(s) = w_1(x)\ w_2(x)\ w_2(y)\ c_2\ w_1(y)\ c_1$. We have

$RF(\Pi_{\{t1, t2\}}(s)) = \{ (t_2,x, t_\infty), (t_1,y, t_\infty) \}$.
Now we consider the serial histories $t_1t_2 = w_1(x)w_1(y)c_1\ w_2(x)w_2(y)c_2$ and
$t_2t_1 = w_2(x)w_2(y)c_2w_1(x)w_1(y)c_1$.
We have $RF(t_1t_2) = \{ (t_2,x, t_\infty), (t_2,y, t_\infty) \}$ si $RF(t_2t_1) = \{ (t_1,x, t_\infty), (t_1,y, t_\infty) \}$.
We have $RF(\Pi_{\{t1, t2\}}(s)) \neq RF(t_1t_2)$ si $RF(\Pi_{\{t1, t2\}}(s)) \neq RF(t_2t_1)$.
This means that $\Pi_{\{t1, t2\}}(s)$ is not in VSR, so VSR is not monotone. So because we have found a schedule that is VSR, but a projection of that schedule is not VSR, we conclude that VSR is not monotone.

Conflict serializability

Now we move to another equivalence criteria which is conflict serializability which is a good serializability criterion (meaning it avoids canonical problems like dirty read, inconsistent read and lost update) and also it is easier to compute/verify. You may start wondering why did we study Herbrand semantics and final state serializability and view serializability because they are not so good criteria and why didn't we just skip to conflict serializability? Well, it is important to see that there are also possible good criteria besides conflict serializability, but they have some small defects and they may be used in context where more relaxed requirements are asked for transactions. We start with the definition of conflict operations.

Definition (Conflict operations and Conflict relations):
Let's assume s is a schedule and t, t' $\in$ trans(s), t $\neq$ t'.
a) Two data operations p $\in$ t and q $\in$ t' *are in conflict* in s, if they access the same data item and at least one of them is a write.
b) The set of tuples conf(s) = {(p, q)} | p, q are in conflict and $p <_s q$} is the *conflict relation* of s.

If you have read carefully the previous sections of this course you may notice that the above requirement for conflict operations about two operations that they both access the same data item and at least one of them is a write is kind of a corner stone of all the transaction modelling done so far. You have seen this requirement in various formats before; you have seen this statement about two operations that they both access the same data item and at least one of them is the write when we have defined the concept of history or schedule. In that definition, we have said that for all *p* and *q* at least one of them being a write operation and they both access the same data item, we then require that they are totally ordered (not partially ordered), either *p* before *q* or *q* before *p*. You have also seen the same kind of statement in the definition of transaction, where we said that the steps of the transaction are in partial order, but every two steps that access the same data item and at least one of them is a write, they need to be totally ordered. So this property which we now call a *conflict property* between two operations that both access the same data item and at least one of them is a write, is very, very important for scheduling the execution of operations from transactions. These two operations from the above definition that both act on the same data item and at least one of them is a write are called a conflict pair and the set of all conflict pairs in a schedule is called the conflict relation on that schedule. A schedule is defined by its conflict relation and as you may have guessed we will say that two schedules s and s' are conflict equivalent if the conflict sets of both schedules are equal.

Definition (Conflict Equivalence):

Schedules s and s‘ are *conflict equivalent*, denoted $s \approx_c s'$, if the set of operations is the same for both schedule, op(s) = op(s‘), and their conflict sets are the same, conf(s) = conf(s‘).

Definition (Conflict Serializability):
Schedule s is *conflict serializable* if there is a serial schedule s‘ such that $s \approx_c s'$. CSR denotes the class of all conflict serializable schedules.

Let’s consider some examples now.
Ex. 1. Let’s prove that the following schedule is conflict serializable: $s = r_1(x)\ r_2(x)\ r_1(z)\ w_1(x)\ w_2(y)\ r_3(z)\ w_3(y)\ c_1\ c_2\ w_3(z)\ c_3$.

We don't need now transactions t_0 and t_∞ anymore. We only need to compute the conflict set of schedules. We start with computing conf(s) and then try to match this with the conflict set of a serial schedule that includes the three transactions. In order to compute conf(s), we take each operation from the schedule, from left to right, and for each operation we seek other possible operations following it that might be in conflict with this one. The first operation from the schedule is $r_1(x)$, so we seek an operation from another transaction that writes *x*, but there is no such operation in schedule *s*; there is $w_1(x)$ but it belongs to the same transaction. Let’s take the second operation, $r_2(x)$. $r_2(x)$ can be in conflict with a subsequent write *x* operation from another transaction and this operation is $w_1(x)$; so, $r_1(x)$ is in conflict with $w_1(x)$. Next, $r_1(z)$ can be in conflict with a subsequent write *z* operation from another transaction, and this operation is $w_3(z)$. $w_1(x)$ can be in conflict with a subsequent read *x* or write *x* operation from a different transaction. There is no such operation. And so on..

We have : $conf(s) = \{ (r_2(x), w_1(x)), (r_1(z), w_3(z)), (w_2(y), w_3(y))\}$
Let’s take all possible serial schedules of t_1, t_2 and t_3 and compute their conflict sets.
$conf(s) = \{ (r_2(x), w_1(x)), (r_1(z), w_3(z)), (w_2(y), w_3(y))\}$
Let’s consider the schedule $s_{t1t2t3} = r_1(x)\ r_1(z)\ w_1(x)\ c_1\ r_2(x)\ w_2(y)c_2\ r_3(z)\ w_3(y)\ w_3(z)\ c_3$
$conf(s_{t1t2t3}) = \{(r_1(z),w_3(z)), (w_1(x), r_2(x)), (w_2(y), w_3(y))\}$
Let’s consider the schedule $s_{t1t3t2} = r_1(x)\ r_1(z)\ w_1(x)\ c_1\ r_3(z)\ w_3(y)\ w_3(z)\ c_3\ r_2(x)\ w_2(y)c_2$
$conf(s_{t1t3t2}) = \{(r_1(z),w_3(z)), (w_1(x), r_2(x)), (w_3(y), w_2(y))\}$
Let’s consider the schedule $s_{t2t1t3} = r_2(x)\ w_2(y)c_2\ r_1(x)\ r_1(z)\ w_1(x)\ c_1\ r_3(z)\ w_3(y)\ w_3(z)\ c_3$
$conf(s_{t2t1t3}) = \{(r_1(z),w_3(z)), (r_2(x),w_1(x)), (w_2(y), w_3(y))\}$
….

We don’t compute the rest of serial schedules because $conf(s) = conf(s_{t2t1t3})$, which means that s is conflict serializable.
Anyway, we can also take a smarter approach, instead of computing the conflict sets of all possible serial schedules with the three transactions, t_1, t_2 and t_3, and then check to see which one is equal to the conflict set of the original schedule *s*. We can see that the conflict set of schedule *s* is made out from conflicting pairs $(r_2(x), w_1(x))$, $(r_1(z), w_3(z))$, $(w_2(y), w_3(y))$ and a conflict pair says something about the operations in the pair – that these two operations are in a *conflict relation* -, but it also say something about *the necessary order of these two operations in a schedule so that the schedule is serializable*. It says that $r_2(x)$ should come before $w_1(x)$ in a schedule, $r_1(z)$ should come before $w_3(z)$ in a schedule, and $w_2(y)$ should come before $w_3(y)$ in a schedule. These three orders can be rephrased as:

- transaction t_2 comes before transaction t_1 ($r_2(x)$, $w_1(x)$)
- transaction t_1 comes before transaction t_3 ($r_1(z)$, $w_3(z)$)
- and transaction t_2 comes before transaction t_3 ($w_2(y)$, $w_3(y)$)

An if we intersect all the above three requirements, we really are left with one possible serial ordering of the three trasactions: $t_2t_1t_3$ and this is the serial schedule that is conflict equivalent with our original schedule *s* (which is what you have already seen above).

Ex. 2 Let's prove that the following schedule is not conflict serializable: $s = r_2(x)\ w_2(x)\ r_1(x)\ r_1(y)\ r_2(y)\ w_2(y)\ c_1\ c_2$.

We first compute the conflict set of schedule s:
$conf(s) = \{(w_2(x), r_1(x)), (r_1(y),w_2(y))\}$

Let's take all possible serial schedules of t_1 and t_2 and then compute their conflict sets:
$s_{t1t2} = r_1(x)\ r_1(y)\ c_1\ r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2$
$conf(s_{t1t2}) = \{(r_1(x),w_2(x)), (r_1(y),w_2(y))\}$
$s_{t2t1} = r_2(x)\ w_2(x)\ r_2(y)\ w_2(y)\ c_2\ r_1(x)\ r_1(y)\ c_1$
$conf(s_{t2t1}) = \{(w_2(x),r_1(x)), (w_2(y),r_1(y))\}$

⇨ $conf(s) \neq conf(s_{t1t2})$ and $conf(s) \neq conf(s_{t2t1})$, so s is not conflict serializable (i.e. is not CSR)

We can also check that a schedule is not conflict serializable by checking that its conflict graph is acyclic (details will follow in the next paragraphs).
The conflict graph of s is (if we have a cycle in the conflict graph, s is not CSR):

As you see in the picture above there is a cycle in the conflict graph of schedule s, so the schedule can not be conflict serializable.

Scheduling algorithms that that produce conflict serializable schedules (the next section is devoted to schedulling algorithms that produce conflict serializable schedules) including the one that you know it more or less, i.e. the schedule based on locking and locks, all they do is just reorder conflicting operations - where by conflicting operations we understand exactly what we specified above - and leave all the other operations that are not conflicting to be executed whenever it's possible. So everything that the scheduling algorithm does is just re-order conflict operations, the rest operations don't matter, just execute them when it can, in parallel. Those operations that are important are pairs of conflicting operations.
So one way to verify that a schedule is conflict serializable is to compare the conflict set of the current schedule to the conflict sets of all possible serial schedules, but that's just the way we did it for VSR and FSR, and we have already said that's not efficient (i.e. that's not possible in polynomial time because we need to compare with all possible permutations of transactions).

There is a better way to do this if we notice that in serial schedules, the order of transactions in conflicting pairs of operations is maintained, meaning either all the operations from transaction t_i, for example, come before all conflicting operations from transaction t_j, for example, or the reverse happens. You would never have in the conflict set of a serial schedule conflicting pairs of operations from two transactions where in some of them the operation from transaction t_i comes before the operation from transaction t_j and in other conflicting pairs the operation from transaction t_j comes before the operation from transaction t_i. We can actually compute whether a schedule is conflict serializable (CSR) by building the so-called *conflict graph* and checking if there are cycles in this graph (a graph cycle in an oriented graph would mean that there are order changes in the schedule contradicting what we have said before about maintaining the order of transactions in serial schedules). A conflict graph is an oriented graph with the transactions involved as vertexes and for each pair of conflicting operations from transactions t_i and t_j, we add an edge from transaction t_i to transaction t_j (the edge points from the first transaction involved in the respective conflicting pair to the second transaction involved in the respective conflicting pair). The baisic idea is that if we have a cycle in this conflict graph like you have seen in the previous example, the schedule depicted in that conflict graph is not conflict serializable. And this (i.e. computing if an oriented graph has a cycle or not) is something that can be done in polynomial time. In conclusion, the more efficient way of computing whether a schedule is CSR is to build the conflict graph and check whether the conflict graph has a cycle. The conflict graph and using the conflict graph to prove that a schedule is conflict serializable are formally described in the following statements.

Definition (Conflict Graph): Let s be a schedule. The **conflict graph** of this schedule G(s) = (V, E) is a directed graph with vertices V := commit(s) and edges $E := \{(t_i, t_j) \mid i \neq j$ and there are steps $p \in t_i$, $q \in t_j$ with $(p, q) \in conf(s)\}$.

Theorem: If the conflict graph of a schedule does not have cycles, then this schedule is conflict serializable. Inverse, if a schedule belongs to CSR, then its conflict graph does not present any cycles.

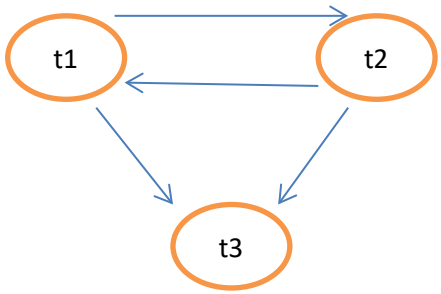
The above theorm can be proven like this. This proof is taken from [1]:
a) Let s be a schedule in CSR. So there is a serial schedule s‘ with conf(s) = conf(s‘).
Now assume that G(s) has a cycle $t_1 \rightarrow t_2 \rightarrow ... \rightarrow t_k \rightarrow t_1$. This implies that there are pairs (p_1, q_2), (p_2, q_3), ... , (p_k, q_1) with $p_i \in t_i$, $q_i \in t_i$, $p_i <_s q_{(i+1)}$, and p_i in conflict with $q_{(i+1)}$. Because s‘ $\approx_c$ s, it also implies that $p_i <_{s‘} q_{(i+1)}$. Because s‘ is serial, we obtain $t_i <_{s‘} t_{(i+1)}$ for i=1, ..., k-1, and $t_k <_{s‘} t_1$. By transitivity we infer $t_1 <_{s‘} t_2$ and $t_2 <_{s‘} t_1$, which is impossible.
This contradiction shows that the initial assumption is wrong. So G(s) is acyclic.
b) Now to prove the inverse. Let G(s) be an acyclic conflict graph. So it must have at least one source node. The following topological sort produces a total order < of transactions:

1. start with a source node (i.e., a node without incoming edges),
2. remove this node and all its outgoing edges,
3. iterate a) and b) until all nodes have been added to the sorted list.

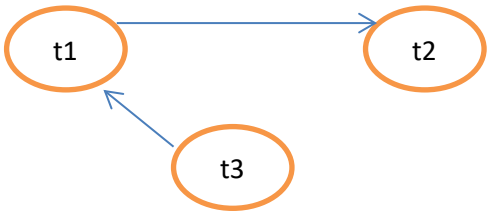
The total transaction ordering order < preserves the edges in G(s); therefore it yields a serial schedule s‘ for which s‘$\approx_c$ s. QED.

Computing if a schedule is conflict serializable can be done in a time polynomial in the number of transactions from the schedule (because we just need to determine if there is a cycle in the conflict graph of that schedule); actually, this can be computed in a time polynomial with respect to the number of transactions multiplied with the total number of all operations from transactions (i.e. the number of operations influences the time needed to build the conflic graph and the number of transactions influences the time to compute if there is a cycle in the graph).
Let's give some other examples of using conflict graphs.

Ex. 4. Show that the following schedule is not CSR: s = $w_1(x)$ $w_2(x)$ $w_2(y)$ c_2 $w_1(y)$ c_1 $w_3(x)$ $w_3(y)$ c_3



The conflict graph has cycles, so s is not in CSR.
Ex. 5. Let's consider the following schedule: s = $w_1(x)$ $r_2(x)$ c_2 $w_3(y)$ c_3 $w_1(y)$ c_1



The conflict graph does not have cycles. So the schedule *s* is conflict equivalent to the serial schedule $t_3t_1t_2$. But in the *s* schedule transaction t_2 completely preceeds transaction t_3 which is not true for the serial schedule $t_3t_1t_2$. So, *s* is not OCSR.

Theorem: CSR $\subset$ VSR.

The theorem states that CSR is included in VSR meaning it's more restrictive than VSR and if you remember I have already said that VSR is included into FSR because VSR is just FSR plus an additional restriction, so it's more restrictive. But CSR is even more restrictive than both of them and we can give an example of a schedule that belongs to VSR, but does not belong to CSR. This example also means that there's no equality between these two sets, CSR and VSR.

Ex.6. Let's consider the schedule from Example 4, that is s = $w_1(x)$ $w_2(x)$ $w_2(y)$ c_2 $w_1(y)$ c_1 $w_3(x)$ $w_3(y)$ c_3.

You have already seen from Example 4 that this schedule is not CSR. In order to check whether the schedule belongs to VSR, we should compute the read from set of schedule s. This is:
$RF(s) = \{(t_3, x, t_\infty), (t_3, y, t_\infty)\}$
And this read-from set is equal to $RF(s_{t1t2t3})$ and to $RF(s_{t2t1t3})$. Which means that schedule s belongs toVSR (view serializable).

In the previous section we had a theorem which said that VSR is not monotone. Remember that an equivalence class (FSR or VSR or CSR) is monotone if for any schedule that belongs to that class, all projections of this schedule on a subset of the transactions involved in this schedule are also part of the same equivalence class. VSR was not monotone, but CSR is, in fact CSR is the largest monotone subset of VSR. This is established formally by the following theorem.

Theorem:
a) CSR is monotone.
b) $s \in CSR$ if and only if $\Pi_T(s) \in VSR$ for all $T \subseteq trans(s)$ (i.e., CSR is the largest monotone subset of VSR).

Commutativity-based equivalence

So up to this point, we have seed that we can prove a schedule is conflict serializable either by comparing its conflict set to the conflict set of all possible serial schedules containing those transactions or by checking that its conflict graph is acyclic. Now we add a third way. The third way is applying some commutativity rules which basically allow us to exchange the order of neighboring operations as long as those operations are not conflicting. And the purpose is to exchange commutative operations until we get to a serial schedule. Actually this is another equivalence criterion, but it is ultimatelly equal to the conflict equivalence criterion. The commutativity rules are the following:

Commutativity rules:

C1: $r_i(x)\ r_j(y) \rightarrow r_j(y)\ r_i(x)$ if $i \neq j$
C2: $r_i(x)\ w_j(y) \rightarrow w_j(y)\ r_i(x)$ if $i \neq j$ and $x \neq y$
C3: $w_i(x)\ w_j(y) \rightarrow w_j(y)\ w_i(x)$ if $i \neq j$ and $x \neq y$
C4: $o_i(x)$, $p_j(y)$ unordered $\rightarrow o_i(x)\ p_j(y)$ if $x \neq y$ or both o and p are reads

The rule C1 says that a schedule can interchange the order of two read operations from different transactions, because they are never in conflict. The second rule, C2 says that the order of a read and a write operations from different transactions can be interchanged as long as they are not in conflict (i.e. they work on different data). Rule C3 says that two write operations from different transactions can be interchanged as long as they are not in conflict (i.e. they work on different data). And finally, rule C4 is the one that specifies that two unordered (i.e. parallel) operations can be placed in any order if they are non-conflicting operations. For the scheduling algorithm the operations that are in conflict are important; for the other operations that are not in conflict it doesn't matter when are scheduled. This is the intuition behind this commutativity rules. Below we have an example of applying this these rules in order to verify that a schedule is conflict serializable. The idea is to start with the schedule and then keep applying recursively the

commutativity rules until we get to a schedule that is serial. In that case, we can say that the initial schedule is a conflict serializable one.

Ex.7. Prove that the schedule s = $w_1(x)$ $r_2(x)$ $w_1(y)$ $w_1(z)$ $r_3(z)$ $w_2(y)$ $w_3(y)$ $w_3(z)$ is conflict serializable.

We take the initial schedule and we see that the $r_2(x)$ $w_1(y)$ operations are commutative so we exchange their order (apply C2 rule) so we get to the new schedule:

$w_1(x)$ $w_1(y)$ $r_2(x)$ $w_1(z)$ $w_2(y)$ $r_3(z)$ $w_3(y)$ $w_3(z)$

This new schedule is conflict equivalent to the initial schedule (i.e. their conflict sets are the same). Then we notice that in this new schedule the neighboring operations $r_2(x)$ $w_1(z)$ are commutative so we apply the C2 rule and switch them so we get to the following conflictequivalent schedule:

$w_1(x)$ $w_1(y)$ $w_1(z)$ $r_2(x)$ $w_2(y)$ $r_3(z)$ $w_3(y)$ $w_3(z)$

In this new schedule, we see that it's actually a serial schedule, that is t_1 t_2 t_3, so we conclude that the original schedule *s* is conflict equivalent to the serial schedule, so it is conflict serializable.

This is an alternative formulation of the conflict serializability criterion and it's called commutativity based reducible criteria. We formulate this new criterion in the following statements.

Definition (Commutativity Based Equivalence): Schedules s and s‘ such that op(s)=op(s‘) are **commutativity based equivalent**, denoted s ~* s‘, if s can be transformed into s‘ by applying rules C1, C2, C3, C4 finitely many times.

Theorem**:** Let s and s‘ be schedules such that op(s)=op(s‘). Then $s \approx_c s'$ if and only if s ~* s‘.

Definition (Commutativity Based Reducibility): Schedule s is **commutativity-based reducible** if there is a serial schedule s‘ so that s ~* s‘.

The commutativity-based reducible criterion is just a synonim for conflict serializability criterion. So if s is conflict equivalent to s', then s is commutativity based equivalent to s'. And the reverse, if s is commutativity based equivalent to s', then s is conflict equivalent to s'.

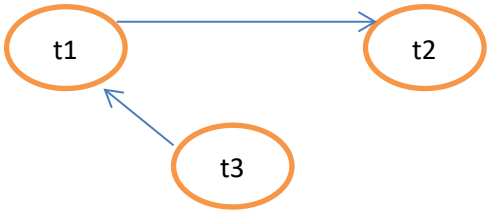
Order preserving conflict serializability

Now we will present some specializations of CSR. So the following serializability criteria are just conflict serializability plus additional particularities, additional restrictions. The order preserving conflict serializability is defined below.

Definition (Order Preservation Conflict Serializability): Schedule s is **order preserving conflict serializable** if it is conflict equivalent to a serial schedule s‘ and in addition it has the following property: for all t, t‘ $\in$ trans(s), if t completely precedes t‘ in s, then the same holds in s‘.
OCSR denotes the class of all schedules with this property.

Because OCSR is just CSR plus an additional restriction it is natural to say that OCSR is included in CSR. But OCSR is not equal to CSR. We have below an example of a schedule that is CSR, but not OCSR.

Ex.1. Let's consider the following schedule: $s = w_1(x)\ r_2(x)\ c_2\ w_3(y)\ c_3\ w_1(y)\ c_1$



The conflict graph does not have cycles. So the schedule *s* is conflict equivalent to the serial schedule $t_3t_1t_2$. But in the *s* schedule transaction t_2 completely preceeds transaction t_3 which is not true for the serial schedule $t_3t_1t_2$. So, *s* is not OCSR.

Commit order preserving conflict serializability

The commit order preserving conflict serializable it is another CSR-derived serializability criterion which implies the conflict serializability criterion plus an additional property which says that the order of conflicting operations also dictates the order of commits of the respective transactions. The criterion is formalized in the following definition.

Definition (Commit Order Preserving Conflict Serializability): Schedule s is **commit order preserving conflict serializable** if s is conflict serializable and in addition it has the property: for all $t_i, t_j \in trans(s)$: if there are $p \in t_i$, $q \in t_j$ with $(p,q) \in conf(s)$ then $c_i <_s c_j$.
COCSR denotes the class of all schedules with this property.

In the definition we have two transactions and they have conflicting operations. The idea is that the order of the conflicting operations *p* and *q* specifies the order of commits of the respective transactions. Notice that once we have $(p,q) \in conf(s)$, $p \in t_i$, $q \in t_j$, we can not have another pair of operations $u \in t_i$, $v \in t_j$ where $(v,u) \in conf(s)$, because if we did have this, then there would be a cycle in the conflict graph in which case, schedule s would no longer be conflict serializable. Once the scheduler added an edge $t_i \rightarrow t_j$ into the conflict graph due to two conflicting operations, the scheduler must to everything to keep this order between transactions t_i and t_j (this order is only important for conflicting operations from t_i and t_j, for non-conflicting operation, it does not matter) . This is just conflict serializability. But the above definition says that in addition to this, the scheduler should also commit t_i before t_j.
So the scheduler should execute the non-conflicting operations whenever it wants, but for the conflicting operations, it should preserve the same order. If transaction t_1 had a conflicting operation that was executed before the conflicting operation from transaction t_2, then for all conflicting operations from transaction t_1 and t_2, the operation from transaction t_1 should be executed first, before the respective conflicting operation from transaction t_2. This preserving of the same conflicting order between transactions is realized by schedulers this using locks. When

one transaction obtains a lock on a specific data x, other conflicting transactions working on the same data x can not obtain this lock on x. So it must wait and the operation from that transaction will be executed later. So this waiting just means order preserving.

Theorem: Schedule s is in COCSR if and only if there is a serial schedule s‘ such that $s \approx_c s'$ and for all $t_i, t_j \in trans(s)$: $t_i <_{s'} t_j \Leftrightarrow c_i <_s c_j$.

Theorem: $COCSR \subset OCSR \subset CSR$.

The schedule $s = w_3(y)\ c_3\ w_1(x)\ r_2(x)\ c_2\ w_1(y)\ c_1$ is an example of schedule that belongs to OCSR, but does not belong to COCSR.

Commit Serializability

Up to this point we considered that all transactions in a schedule commit, we payed no attention to transactions that abort. And we demonstrated various equivalence and serializability properies for random, incomplete schedules or histories. These are all good, but in the end a scheduler needs to be able to determine at the end of a complete schedule’s execution if the execution was correct (i.e. it respected the ACID properties of transactions involved) or not. And it can only hope the execution was correct, because if the schedule execution was incorrect and some transactions are already commited, it can not do anything. The mathematical machinery presented in this small section comes to help. What this theory says is that if the scheduler starts with an empty schedule and it keeps adding operations to this schedule (as a new operation arrives and as the scheduler schedules that operation for execution) and before adding a new operation to the schedule, the scheduler verifies that by adding this new operation to the schedule, the schedule remains correct (i.e. conflict serializable; respects the ACID properties of the transactions involved), in the end the complete schedule (history) produced by the scheduler in this way remains correct (i.e. conflict serializable; respects the ACID properties of the transactions involved). So the already presumed recipe is correct: condier adding a new operation to an already produce schedule after verifying that the new schedule is still conflict-serializable.

Definition (Closure Properties of Schedule Classes): Let E be a class of schedules.
For schedule s let CP(s) denote the projection $\Pi_{commit(s)}(s)$.
E is **prefix-closed** if the following holds: $s \in E \Leftrightarrow p \in E$ for each prefix of s.
E is **commit-closed** if the following holds: $s \in E \Rightarrow CP(s) \in E$.

Theorem:
CSR is prefix-commit-closed, i.e., prefix-closed and commit-closed.

In the above definition, CP(s) denotes the projection of schedule s on the commit set of s, meaning that we take all the operations from transactions that committ in s and we remove the operations from the rest of transactions (i.e. the transactions that aborted). So if we have a schedule s which has three transactions, t_1, t_2 and t_3, and transactions t_1 and t_2 committ in the schedule s, but transaction t_3 abort, then we remove all the operations from transaction t_3 and the remaining schedule is the projection on the commit set of s (i.e. CP(s)). A prefix-closed class (like CSR) is a class that includes in it all prefixes of a schedule that is already included in the

class. Similarly, a commit-closed class is a class that includes CP(s) for any schedule s previously included in the class. CSR has both these qualities, prefix-closed and commit-closed.

Definition (Commit Serializability):
Schedule s is **commit-C-serializable** if CP(p) is C-serializable for each prefix p of s, where C is a serializability class like FSR, VSR, or CSR. The resulting classes of commit-C-serializable schedules are denoted CMFSR, CMVSR, and CMCSR.

Theorem:
a) CMFSR, CMVSR, CMCSR are prefix-commit-closed.
b) CMCSR $\subset$ CMVSR $\subset$ CMFSR

Generalization of conflict serializability. Indivisible units.

Let's consider as example that the scheduler knows all the transactions that it will execute in advance and it also knows their semantics. Let's consider our scheduler needs to execute the following 3 types of transactions:

- transfer transactions on checking accounts a and b and savings account c:
  $t_1 = r_1(a)\ w_1(a)\ r_1(c)\ w_1(c)$
  $t_2 = r_2(b)\ w_2(b)\ r_2(c)\ w_2(c)$
- balance transaction:
  $t_3 = r_3(a)\ r_3(b)\ r_3(c)$
- audit transaction:
  $t_4 = r_4(a)\ r_4(b)\ r_4(c)\ w_4(z)$

The first two transactions are executed by two different persons, let's assume they are part of the same family and they just transfer money from one account to the other, all three accounts being family accounts. Then there's the balance transaction t_3 which is executed by another member of the same family. This transaction just selects all the balances of all the three family accounts a, b, and c. The third type of transaction selects all the balances of all three family accounts and then it writes a new value to an audit table or a log table of the bank. So transaction t_4 is the transaction of the bank which gets executed from time to time. The following are some possible interleavings of operations.

Schedule examples:

$r_{11}(a)\ w_1(a)\ r_2(b)\ w_2(b)\ r_2(c)\ w_2(c)\ r_1(c)\ w_1(c)$	$\in$ CSR	application-tolerable interleavings
$r_1(a)\ w_1(a)\ r_3(a)\ r_3(b)\ r_3(c)\ r_1(c)\ w_1(c)$	$\notin$ CSR	
$r_1(a)\ w_1(a)\ r_2(b)\ w_2(b)\ r_1(c)\ r_2(c)\ w_2(c)\ w_1(c)$	$\notin$ CSR	non-tolerable interleavings
$r_1(a)\ w_1(a)\ r_4(a)\ r_4(b)\ r_4(c)\ w_4(z)\ r_1(c)\ w_1(c)$	$\notin$ CSR	

The first schedule is CSR so it is a corect schedule. The second schedule does not belong to CSR, there's a cycle in the conflict graph, and it is just a realization of the inconsistent read problem. But assuming the transaction that performs the inconsistent read, t_3, can tolerate a small inconsistency over a very small time interval, this schedule may also be considered suitable by the application. So we may assume that it's not such a big problem if at some point he member of

the family t_3 selects inconsistent values from the database as long as in a subsequent select read operation, it selects the correct values. The third and fourth schedules are non CSR and the third schedule is the lost update problem which leads to inconsistent data in the dataset, so it is not tolerable by the system. Similar, in the fourth schedule the bank (t_4) needs to have at each point the correct view on the money values from the database, so this is not tolerable. In conclusion, even if the schedule is not CSR, some applications may find the schedule tolerable. And we will introduce a serializability criterion that is a generalization of conflict serializability. This criterion takes into account some input from the application which specifies the interleavings that it tolerates. And if the application tolerates an interleaving the scheduler will permit its execution even if it is not CSR. But this works only if we know these allowable interleavings about all the transactions in advance. We can formalize what we have said before using the concept of *indivisible units.*

Definition (Indivisible Units): Let $T=\{t_1, ..., t_n\}$ be a set of transactions. For $t_i, t_j \in T$, $t_i \neq t_j$, an **indivisible unit of t_i relative to t_j** is a sequence of consecutive steps of t_i such that no operations of t_j are allowed to interleave with this sequence. **$IU(t_i, t_j)$** denotes the ordered sequence of indivisible units of t_i relative to t_j. $IU_k(t_i, t_j)$ denotes the k^{th} element of $IU(t_i, t_j)$.

We will give an example so that you will understand it better.

Ex.1. Let's assume we have the following schedules.
$t_1 = r_1(x)\ w_1(x)\ w_1(z)\ r_1(y)$
$t_2 = r_2(y)\ w_2(y)\ r_2(x)$
$t_3 = w_3(x)\ w_3(y)\ w_3(z)$

and then we have the following indivisible units specified by the application:
$IU(t_1, t_2) = < [r_1(x)\ w_1(x)],\ [w_1(z)\ r_1(y)] >$
$IU(t_1, t_3) = < [r_1(x)\ w_1(x)],\ [w_1(z)],\ [r_1(y)] >$
$IU(t_2, t_1) = < [r_2(y)],\ [w_2(y)\ r_2(x)] >$
$IU(t_2, t_3) = < [r_2(y)\ w_2(y)],\ [r_2(x)] >$
$IU(t_3, t_1) = < [w_3(x)\ w_3(y)],\ [w_3(z)] >$
$IU(t_3, t_2) = < [w_3(x)\ w_3(y)],\ [w_3(z)] >$

The schedules:
$s_1 = r_2(y)\ r_1(x)\ w_1(x)\ w_2(y)\ r_2(x)\ w_1(z)\ w_3(x)\ w_3(y)\ r_1(y)\ w_3(z)$ → respects all indivisible units
$s_2 = r_1(x)\ r_2(y)\ w_2(y)\ w_1(x)\ r_2(x)\ w_1(z)\ r_1(y)$ → violates $IU_1(t_1, t_2)$ and $IU_2(t_2, t_1)$

Conflict serializable schedulers

Bibliography:

This text is largely based on the book

1. WeikumG., Vossen G., Transactional Information System: Theory, Algorithms, and Practice of Concurrency Control and Recovery, Kaufmann Morgan Publ. 2002.

Now we are at a point in our theory where we can focus on schedulers, on the scheduling algorithm itself. During the time the scheduler executes a transaction, this transaction can have several states. It has the *begin state*, then it may move to the *running state* then at some point it can be *blocked* which means it's suspended (i.e. it waits for a specific event), then it can go back to running as the transaction resumes. These states, blocked and active, are the so-called *active states*. The transaction can move from the active states to the *commit state* when it's done executing or it can move to the *aborted state* if it has an operation that causes a cycle in the conflict graph and then it may move to the begin status again so it can be restarted by the scheduler; or it can be left aborted and just an error message is shown to the user.

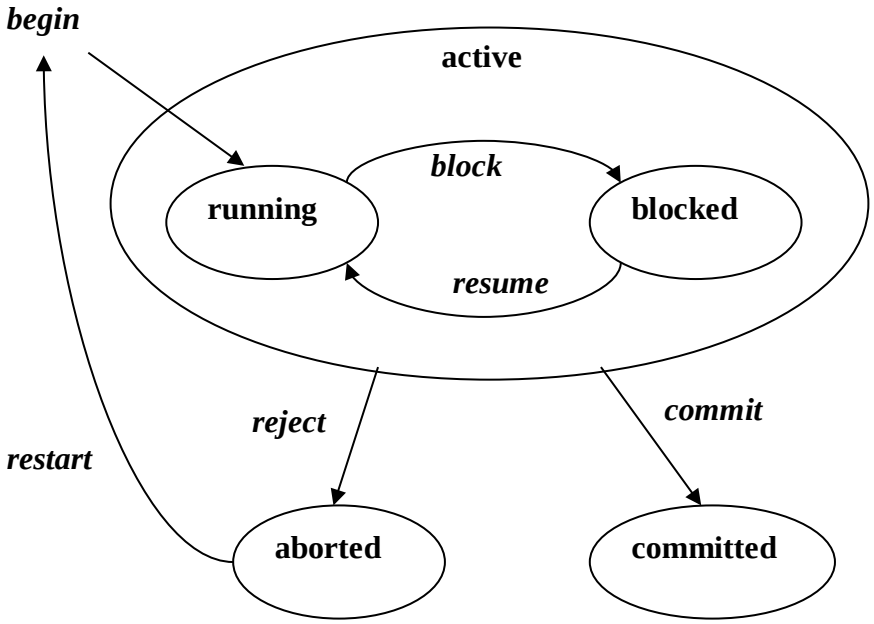


Fig. 1. Executing states of a transaction in a scheduler [1]

Now we are interested only in schedulers that produce only CSR schedules. The concurrency control algorithms (we will also call a scheduling algorithm a concurrency control algorithm) we will talk about in this section all produce CSR schedules. So, in this section, the terms scheduling algorithm, scheduler, concurrency control protocol or concurrency control algorithm are all synonyms.

Remember I said that a scheduler when it receives a new operation can do only three things:

- the scheduler can execute this operation when it arrives because it has enough data to conclude that by executing this operation it will produce a CSR schedule
- the schedule can refuse to execute this operation and abort the transaction to which this operation belongs because the scheduler decides that by executing this operation now or in the future, the obtained schedule will be non-CSR
- the scheduler can choose the third option which means just to defer the execution of this operation until later on when the scheduler has enough data to conclude that by executing the operation later it will get a CSR schedule.

There are concurrency control algorithms or scheduling algorithms that would rather choose the last alternative, meaning they don't execute a new operation when they receive it - these are pessimistic concurrency control algorithms, they usually block the execution and they make the operation wait until a specific condition is met in the future. There are the so-called optimistic schedulers which assume that conflicts between operations are don't happen very often in the schedule and they don't make any operation wait (i.e. they choose the first alternative), they just keep executing operations and if later on the scheduler decides that the schedule obtained so far is non-CSR, they just abort a transaction and maybe restarts it. There are also the hybrid schedulers which combine features from pessimistic and optimistic concurrency control. The landscape of scheduler algorithms that we will discuss in this section are outlined below.

- Pessimistic concurrency control:
  - Locking based:
    - Two-phase locking:
      - 2PL and customizations: Conservative 2PL (C2PL), Static 2PL (S2PL) and Strong 2PL (SS2PL)
      - Generalizations of 2PL: Altruistic Locking (AL), Ordered 2 Phase Locking (O2PL)
    - Non two-phase locking: Write-only Tree Locking (WTL), Read-Write Tree Locking (RWTL)
  - Non-locking based:
    - Timestamp ordering (BTO)
    - Serialization Graph Testing (SGT)
- Optimistic concurrency control:
  - Backward Oriented Concurrency Control (BOCC)
  - Forward Oriented Concurrency Control (FOCC)
- Hybrid concurrency control

Locking based Pesimistic Concurrency Control

When describing locking based scheduling algorithms, we still have our computational model, the page model, with read and write operations and commit and abort. We will augment our computational model with four additional operations: $rl_i(x)$ – read lock on x, $wr_i(x)$ – write lock on x, $ru_i(x)$ – read unlock on x, $wu_i(x)$ – write unlock on x. The scheduler requests a lock on behalf of each operation from a transaction. We have two types of locks, we have *read locks* and

write locks. The read locks are also called *shared locks* and the write locks are also called *exclusive locks*. Prior to using a data item, a transaction must request an appropriate lock on that data item (read lock or write lock). If the data item is not yet locked in an **incompatible mode** the lock is granted; otherwise there is a **lock conflict** and the transaction becomes **blocked** (suffers a **lock wait**) until the current lock holder **releases the lock**. We have a compatibility matrix between these two types of locks depicted in the next figure.

		$rl_j(x)$	$wl_j(x)$	Lock requestor
Lock holder	$rl_i(x)$	+	-	
	$wl_i(x)$	-	-	

Fig. 2. Locks compatibility matrix

If transaction j requests a read lock on x and transaction i already has the read lock on x, because these two are compatible, transaction j will be granted the lock on x. If transaction i has the read lock on x and transaction j requests the write lock on x, the write lock on x is not granted because these are incompatible. If transaction i has the write lock on x and transaction j requests the read lock on x, it is not granted because they are incompatible and the same happens if transaction I has the write lock on x and transaction j requests the write lock on x. The rl(x) and wl(x) operations imply that after they are executed, the corresponding lock is granted. We add some additional rules in order to simplify our computational model:

LR1: Each data operation $o_i(x)$ must be preceded by $ol_i(x)$ and followed by $ou_i(x)$.
LR2: For each x and t_i there is at most one $ol_i(x)$ and at most one $ou_i(x)$.
LR3: No $ol_i(x)$ or $ou_i(x)$ is redundant.
LR4: If x is locked by both t_i and t_j, then these locks are compatible.

LR1 says that each data operation, read or write, must be preceded by a corresponding lock operation and followed by an unlock operation. LR2 says that for each data, a transaction obtains a lock only one time and then unlocks only once. This simplifies our model - we don't want to lock a data x for transaction t_i and then unlock it and lock it again and so. There is only one lock operation for data x for a transaction t_i and the same only one unlock operation – LR3. LR4 specifies that if x is locked by both t_i and t_j then these locks should be compatible; otherwise we cannot have two transactions having the lock on the same data at the same time.

Two-phase locking (2PL) scheduling algorithm

Two-phase locking (2PL) means is that each transaction requests and releases locks in two episodes, in the first episode, it requests locks and in the second episode, it only releases previously acquired locks. In other words, once a transaction released its first lock, it cannot request new locks. The formal definition of 2PL follows.

Definition (Two-phase Locking scheduling algorithm): A locking protocol is **two-phase (2PL)** if for every output schedule s and every transaction $t_i \in trans(s)$ no ql_i step follows the first ou_i step ($q, o \in \{r, w\}$).

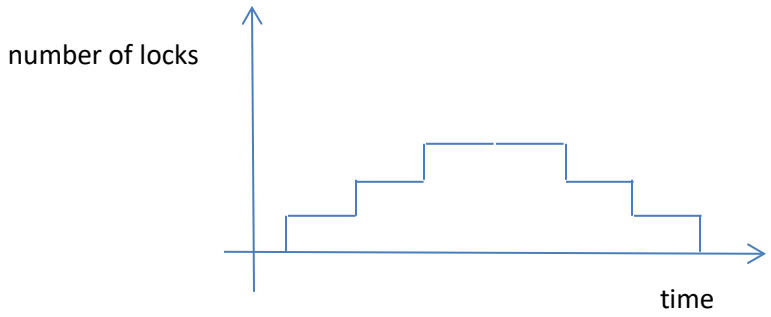
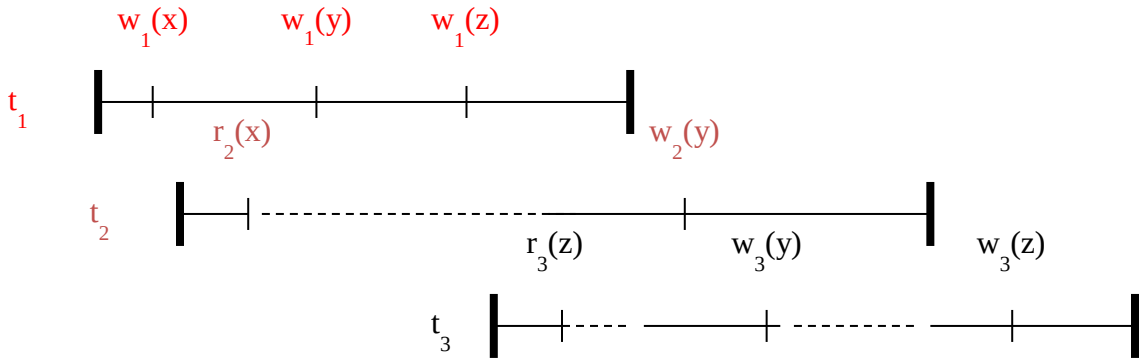


Fig. 3. Number of locks evolution in time for a transaction in 2PL

We can visualize two-phase locking (2PL) in the previous figure. The number of locks a transaction acquires is depicted across time. You can see that the number of locks held by a transaction increases up to a plateau point where it starts to decrease until it gets to zero. The plateau is the place where the transaction starts unlocking resources and after the first unlock, it can not request new locks. Let's take an example now:

Ex.1. Let's consider that the 2PL scheduler receives at its input the following operation sequence $w_1(x)$ $r_2(x)$ $w_1(y)$ $w_1(z)$ $r_3(z)$ c_1 $w_2(y)$ $w_3(y)$ c_2 $w_3(z)$ c_3 and let's depict the output, the schedule produced by the 2PL scheduler from this operation sequence [1]. The output schedule is displayed in the following figure where time runs from left to right and also described in the following total-order sequence (where time still goes from left to right).



Output schedule = $wl_1(x)$ $w_1(x)$ $wl_1(y)$ $w_1(y)$ $wl_1(z)$ $w_1(z)$ $wu_1(x)$ $rl_2(x)$ $r_2(x)$ $wu_1(y)$ $wu_1(z)$ c_1 $rl_3(z)$ $r_3(z)$ $wl_2(y)$ $w_2(y)$ $wu_2(y)$ $ru_2(x)$ c_2 $wl_3(y)$ $w_3(y)$ $wl_3(z)$ $w_3(z)$ $wu_3(z)$ $wu_3(y)$ c_3

The first operation that arrives at the scheduler is $w_1(x)$, so transaction t_1 obtains the write lock on x because no other transaction has the write lock on x and then it executes $w_1(x)$. The next operation in the sequence of operations is $r_2(x)$. Transaction t_2 tries to request the read lock on x, but it cannot be granted by the scheduler because transaction t_1 still has the write lock on x and this read lock is incompatible with the write lock of t_1. Hence $r_2(x)$ is put on hold and the transaction t_2 is put on hold (i.e. it is blocked). This is what I meant in a previous section when I said that a locking algorithm only reorders the conflicting operations. So this $r_2(x)$ is conflicting with $w_1(x)$ and that is why its execution is shifted later on. With $r_2(x)$ is put on hold, the next

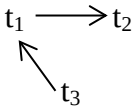
operation is w1(y), transaction t_1 obtains the write lock on y because no other transaction has any lock on y, and then it executes $w_1(y)$. Next is $w_1(z)$, transaction t_1 tries to get and obtains the lock on z and then executes $w_1(z)$. Remember that $r_2(x)$ is still on hold here. Next operation $r_3(z)$ tries to get the read lock on z, but it cannot because transaction t_1 has the write lock on z, so t_1 is put on hold. Transaction t_1 does not have any operation so it releases the lock on x and transaction t_2 can finally get the read lock on x for $r_2(x)$ which was on hold and then it executes $r_2(x)$. Transaction t_1 releases the lock on y and z and commits. Transaction t_3 wakes up and obtains the read lock on z because transaction t_1 released the lock on z. Transaction t_3 executes $r_3(z)$ and then the next operation that arrives is w2(y), it obtains the write lock on y and then it executes $w_2(y)$. $w_3(y)$ arrives at the scheduler and it is put on hold because transaction t_2 has the write lock on y. Then t_2 releases the lock on x and y, t_3 wakes up and obtains the write lock on y, executes $w_3(y)$ and finally it obtains the write lock on z and executes $w_3(z)$. Then t_3 releases all the acquired locks and commits.

Theorem: A two-phase locking scheduling algorithm (2PL) always produces CSR schedules.

So, 2PL scheduling algorithms are always safe because the set of schedule they produce is included in CSR. But it's equally important to notice that there are CSR schedules that are not (can not be) produced by 2PL scheduling algorithms. In the following example, we will present a schedule that is CSR (actually, it is OCSR), but we show that it can not be produced by a 2PL algorithm.

Ex.2 Let's consider the following schedule: $s = w_1(x)\ r_2(x)\ r_3(y)\ c_3\ w_1(y)\ c_1\ w_2(z)\ c_2$. We will try to show that schedule s is COCSR, but schedule *s* can not be produced by a 2PL algorithm.

The conflict graph of s is:



The conflict graph does not have any cycles => schedule *s* belongs to CSR.
Because there is no transaction in schedule s that completely preceeds other transaction, we can say that the schedule s is OCSR. Because t_3 commits before t_1 and t_1 commits before t_2 (this being the order of conflicting operations from the three transactions), we can also say that schedule *s* is COCSR.

In order for schedule *s* to be produced by a 2PL scheduling algorithm, we must have the following:

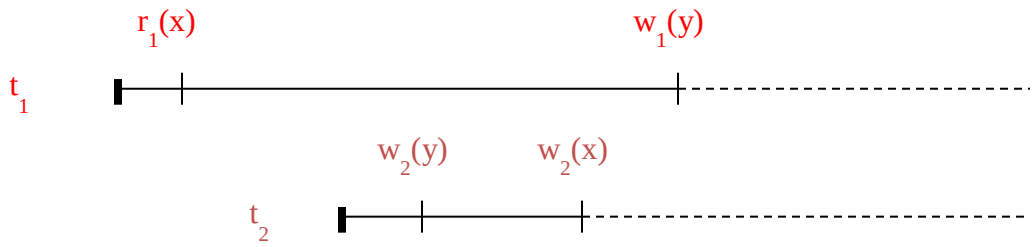
- Transaction t_2 must obtain the $rl_2(x)$ lock immediately after $w_1(x)$ got executed
- Transaction t_2 can not obtain the $rl_2(x)$ lock immediately after $w_1(x)$ got executed because transaction t_1 can not issue $wu_1(x)$ as it still has to acquire a couple of locks before (i.e. the 2 Phase Locking property).

Therefore, schedule *s* can not be produced by a 2PL algorithm.

Theorem: A two-phase locking scheduling algorithm (2PL) always produces OCSR schedules.

Deadlock handling

In all scheduling algorithms based on locks, deadlock may happen. Deadlock happened when we have a cycle in the wait-for graph. The wait-for graph is just a graph with transactions as vertices and there is an edge from t_i to t_j if t_j waits for a lock held by t_i. An example of deadlock is depicted in the following figure:



The Wait-For table for the above example would be (the Wait-For graph cycle is clearly visible in the Wait-For table):

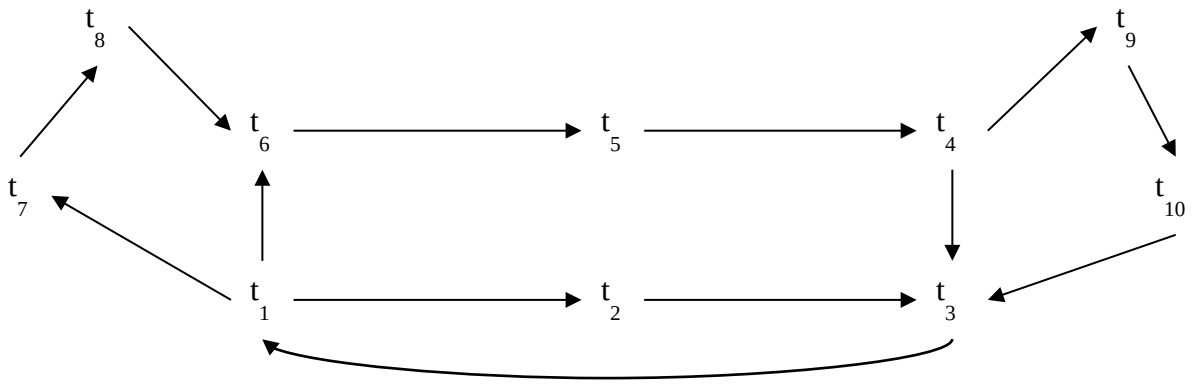
Transaction which waits for lock	**Lock data**	**Lock type**	**Transaction which owns the lock**
t_2 (operation $w_2(x)$)	x	write lock	t_1
t_1 (operation $w_1(y)$)	y	write lock	t_2

We have two transactions, t_1 and t_2, time as always runs from left to right horizontally. Transaction t_1 executes $r_1(x)$ first, so it first obtains the read lock on x and then executes $r_1(x)$ and then along comes operation $w_2(y)$ from transaction t_2. Transaction t_2 obtains the lock on y and then writes y and then transaction t_2 tries to execute $w_2(x)$ which means that it has to acquire the write lock on x, but it cannot get this lock because transaction t_1 has the read lock on x which is incompatible with the write lock on x requested by transaction t_2. Transaction t_1 cannot release its lock because of the two-phase locking rule (as t1 still has to acquire a new lock for $w_1(y)$). The next operation is $w_1(y)$ which demands a write lock on y which does not happen because transaction t_2 has the write lock on y. So transaction t_2 awaits for transaction t_1 to release the lock on x and transaction t_1 awaits for transaction t_2 to release the lock on y and no transaction moves forward and is able to do some useful work. So t_1 and t_2 would wait indefinitely and no one would solve this situation. So there must be an external intervention, there should be the scheduler who steps in and resolves this deadlock. The way it resolves the deadlock is by aborting one of those two transactions and releasing the locks owned by the aborted transaction. Of course, the aborted transaction can also be automatically restarted by the scheduler or the application that tries to execute that transaction can be informed that the transaction failed and maybe the application wants to resend the transaction again. But these are just details. So the idea is that the transaction manager or the concurrency control algorithm (i.e. scheduler) maintains a Wait-For graph which has transactions as nodes and there's an edge from transaction t_i to t_j if t_j waits for a lock held by t_i. From time to time, the scheduler needs to check whether there is a cycle in this graph. If there is a cycle in the Wait-For graph, we have a deadlock. In the above

example, I have just shown a very simple cycle, where transaction t_1 waits for t_2 and transaction t_2 waits for t_1, but we can have more complicated cycles like one where t_1 waits for t_2, t_2 waits for t_3 on another resource, t_3 then waits for t_4 on another resource and finally, t_4 waits for t_1. So this is a cycle with more than two nodes. The concurrency control algorithm (i.e. scheduler) can test whether there is a cycle in the Wait-For graph continuously upon each lock request operation (actually, you would test for a cycle only when there's a new lock request that is put into the wait state because other transaction owns that lock) or the scheduler can test this periodically, from time to time. There are advantages and disadvantages for each of these two alternatives. If this test is performed continuously after each lock wait, then whenever the deadlock happens, it is instantly detected and resolved by aborting one transaction from the cycle. But there is a significant cost with this because there is additional work that must be performed or executed by the CPU in order to keep checking the Wait-For graph for cycles after each lock wait. This can be made more efficient by performing these checks periodically, so from time to time, not on each lock wait. but of course there is the drawback that the deadlock may appear and the scheduler detects the deadlock only after some time has passed, leaving those transactions to just wait in a blocked state. There are various strategies on which transaction victim to choose for abortion in a wait-for graph cycle, and these are:

a) the transaction that added the last edge in the Wait-For graph cycle
b) a random transaction from the cycle in the Wait-For graph
c) the youngest transaction (i.e. most recently created) out of all transactions in the Wait-For graph cycle
d) the transaction that owns the minimum amount of locks from the cycle
e) the transaction that performed the minimum amount of work in a Wait-For graph cycle
f) the transaction that participates to most cycles in the Wait-For graph
g) the transaction that has the most edges in the Wait-For graph and is on at least one cycle

For example, in the following Wait-for graph there are 5 cycles [1].



One cycle is formed by the transactions t_1, t_2, t_3, another cycle would be t_1, t_7, t_8, t_6, t_5, t_4,t_3, another one would be t_1, t_7, t_8, t_6, t_5, t_4, t_9, t_{10}, t_3, the fourth cycle is t_1, t_6, t_5, t_4, t_9, t_{10}, t_3 and the fifth one is t_1, t_6, t_5, t_4, t_3. Most deadlock resolving strategies would select t_1 or t_3 to abort because they participate to all of the cycles; this would break all the cycles.

There are some strategies that try to prevent locks, most of them rely on aborting a transaction if there are signs that it will cause a cycle in the Wait-For graph. One strategy is the Wait-die strategy where if transaction t_i tries to get a lock that is owned by t_j, if t_i started before t_j then t_i should wait, otherwise t_i is aborted. A second strategy is Wound-wait in which if transaction t_i tries to get a lock that is owned by t_j, if t_i started before t_j then abort the t_j transaction, otherwise wait for the lock release. Another strategy is Immediate restart in which if transaction t_i tries to get a lock that is owned by t_j, t_i transactions gets aborted. Another strategy is Running priority in which if transaction t_i tries to get a lock that is owned by t_j, if t_j is itself blocked then t_j is aborted, else t_i waits for the lock release from t_j. The fifth strategy is Timeout in which if a transaction waits for a lock release, it waits for a limited amount of time (i.e. timeout) and after this time interval has passed, the transaction is aborted.

Variants of 2PL

We present in this section three customizations of two-phase locking (2PL) which are scheduling algorithms that are 2PL, but also add restrictions to 2PL. They are defined in the following lines.

Definition (Conservative 2PL): **Static** or **conservative 2PL (C2PL)** is a form of two-phase locking where each transaction acquires all its locks before the first data operation (preclaiming).

Previously, in Fig. 3 we depicted the evolution in time of the number of locks owned by a transaction in general two-phase locking (2PL). The evolution of number of locks in time for a transaction in C2PL is depicted in the following figure.

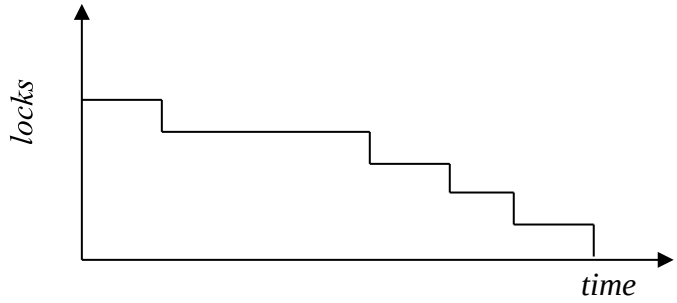


Fig. 4. Evolution of number of locks owned by a transaction in time for C2PL [1]

Definition (Strict 2PL): **Strict 2PL (S2PL)** is a form of two-phase locking where each transaction holds all its write locks until the transaction terminates.

The evolution of number of locks in time for a transaction in S2PL is depicted in the following figure. In this figure only the write locks (not the read locks) are depicted.

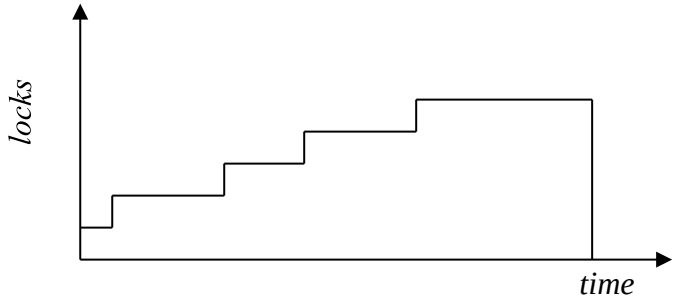


Fig. 5. Evolution of number of locks owned by a transaction in time for S2PL [1]

Definition (Strong 2PL): **Strong 2PL (SS2PL)** is a form of two-phase locking where each transaction holds all its locks (i.e., both read and write locks) until the transaction terminates.

The evolution of the number of locks in time for a transaction in SS2PL is the same as the one depicted in the above figure, Fig. 5, only that in case of SS2PL the graphic will depict both read and write locks owned by the transaction.

Theorem: The set of schedules produced by SS2PL is included in the set of all schedules produced by S2PL which is included in the set of all schedules produced by 2PL. Actually, all the schedules produced by SS2PL are included in COCSR.

Ordered Sharing of Locks (O2PL)

We now present some generalizations of two-phase locking (2PL). These are scheduling algorithms based on 2PL, but add some relaxations to the rules of 2PL. We will start with Ordered Sharing of Locks (O2PL). Let's consider an example first. Let's take the following schedule:

$s = w_1(x)\ r_2(x)\ r_3(y)\ c_3\ w_1(y)\ c_1\ w_2(z)\ c_2$

What is interesting about this schedule is that it is COCSR, but it can not be produced by a 2PL scheduling algorithm. This schedule is COCSR, because the conflict graph does not have cycles and the order of conflicting operations dictate the order of the commit operations of the transactions. However 2PL can never generate this schedule because in order to have the sequence $w_1(x)\ r_2(x)$ scheduled for execution, this would require transaction t_2 to own the read lock on x, but this can not happen in 2PL because transaction t_1 already has the write lock on x because $w_1(x)$ has already been executed and transaction t_1 cannot release the write lock on x here because it still has an additional operation to execute which requires a new lock acquisition (w1(y)). In a 2PL scheduling $r_2(x)$ would be executed only after $w_1(y)$. So this schedule can never be produced by a 2PL algorithm, but nevertheless this schedule is correct (i.e. COCSR), it respects the ACID properties of the transactions involved. We are trying to relax the rules of 2PL so that it allows CSR schedules like the one presented in the above example.

Out intuition in this process is that main goal of locks is *to order conflicting operations*. And the purpose of the 2PL principle is to *properly order conflicting operations* where "properly" means that it keep the same order between conflicting operations of two transactions; i.e. if the order of a conflicting operations pair places t_i before t_j, the 2PL rule makes sure that for all the following conflicting operations pair from t_i and t_j, the operation from t_i comes before the conflicting operation of t_j; this prevents cycles in the conflict graph, thus assuring conflict-serializability. So if after we encounter the first conflicting pair between transactions t_i and t_j we can maintain the conflicting ordering between operations from t_i and t_j without the 2PL rule, we do not need the 2PL rule anymore. In the same way, the above schedule would have been feasible if locks could be shared between conflicting transactions such that the order of lock acquisitions dictates the order of data operations (because actually this is exactly the purpose of locks – to order data operations).

In O2PL *we allow two conflicting operations, $p_i(x)$ from transaction i and $q_j(x)$ from transaction j, to share the lock (meaning transaction j can obtain the lock on x even if transaction i has the p*

lock on x) and we write it like this: **$pl_i(x) \rightarrow ql_j(x)$**, *as long as the order of the lock request operation dictates the order of the corresponding data operations protected by those locks* (meaning that $p_i(x)$ should be scheduled for execution before $q_j(x)$). If we reconsider the above example using this O2PL rule of shared locks, we get to this schedule:
$wl_1(x)\ w_1(x)\ rl_2(x)\ r_2(x)\ rl_3(y)\ r_3(y)\ ru_3(y)\ c_3\ wl_1(y)\ w_1(y)\ wu_1(x)\ wu_1(y)\ c_1\ wl_2(z)\ w_2(z)\ ru_2(x)\ wu_2(z)\ c_2$

We have here transaction 1 which acquires the write lock on x then it executes $w_1(x)$, then transaction 2 acquires the read lock on x thus sharing the lock on x with transaction 1 and then t_2 executes $r_2(x)$; t_2 was able to share the lock on x with t_1 because the order of the lock requests (t_1 before t_2) is the same as the order of the data operations (i.e. $w_1(x)$ happens before $r_2(x)$). The remaining operations are executed according to 2PL. $r_3(y)$ is executed after transaction t_3 obtains the read lock on y, then t_3 commits after releasing the locks. Following, $w_1(y)$ gets executed after transaction 1 obtains the write lock on y, t_1 unlocks x and y and then commits. Finally, t_2 obtains the write lock on z, executes $w_2(z)$, releases all its locks and then commits.

Here we have the original locks compatibility table for locking scheduling algorithms (‘+’ means compatible, ‘-‘ means incompatible).

LT1	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	-
$wl_i(x)$	-	-

And the idea is to swap to this incompatibility table with other compatibility tables where the incompatibility sign ‘-‘ is replaced with order sharing of locks sign ‘$\rightarrow$‘. So we obtain the following 7 lock compatibility tables.

LT2	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	$\rightarrow$
$wl_i(x)$	-	-

LT3	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	-
$wl_i(x)$	$\rightarrow$	-

LT4	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	-
$wl_i(x)$	-	$\rightarrow$

LT5	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	$\rightarrow$
$wl_i(x)$	$\rightarrow$	-

LT6	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	-
$wl_i(x)$	$\rightarrow$	$\rightarrow$

LT7	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	$\rightarrow$
$wl_i(x)$	-	$\rightarrow$

LT8	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	$\rightarrow$
$wl_i(x)$	$\rightarrow$	$\rightarrow$

The most general one is LT8 which says that two read locks from different transactions are fully compatible and these other combinations of write lock with a read lock and write lock with a write lock are allowed in the order sharing of locks (O2PL).
The lock acquisition rule of O2PL is:
OSLA (lock acquisition): $pl_i(x) \rightarrow ql_j(x)$ is permitted as long as if $pl_i(x) <_s ql_j(x)$ then $p_i(x) <_s q_j(x)$.

Then if we consider the following schedule:
$s = wl_1(x)\ w_1(x)\ wl_2(x)\ w_2(x)\ wl_2(y)\ w_2(y)\ wu_2(x)\ wu_2(y)\ c_2\ wl_1(y)\ w_1(y)\ wu_1(x)\ wu_1(y)\ c_1$

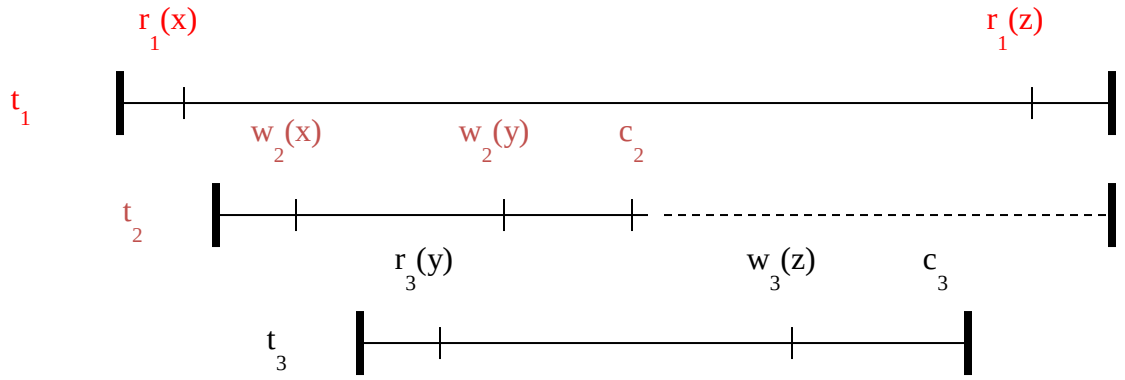
we can see that it satisfies the OS1 lock acquisition rule and also the LR1-LR4 rules of locking algorithms and also the 2PL rule (i.e. no new lock request after the first lock release), but it's not CSR. This example shows us that we also need another rule for lock release.

OSLR (lock release): If $pl_i(x) \rightarrow ql_j(x)$ and t_i has not yet released any lock, then t_j is **order-dependent** on t_i. If such t_i exists, then t_j is **on hold**. While a transaction is on hold, it must not release any locks.

Definition (Ordered sharing of Locks algorithm): An Ordered Sharing of Locks (O2PL) algorithm is a locking based scheduling algorithm that obeys the general locking rules LR1-LR4, the 2PL rule and rules OSLA and OSLR, and uses a lock compatibility table from LT2-LT8.

Let us show the previous example with the above rules applied and see what happens. The operations arrive in the following order at the scheduler:
operations order = $r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$

Remember, time always runs from left to right horizontally. This sequence of operations would be scheduled by an O2PL scheduler like this [1]:



$s = rl_1(x)\ r_1(x)\ wl_2(x)\ w_2(x)\ rl_3(y)\ r_3(y)\ wl_2(y)\ w_2(y)\ wl_3(z)\ w_3(z)\ ru_3(y)\ wu_3(z)\ c_3\ rl_1(z)\ r_1(z)\ ru_1(x)\ ru_1(z)\ wu_2(x)\ wu_2(y)\ c_2\ c_1$

Transaction 1 obtains the read lock on x, executes $r_1(x)$, then transaction 2 obtains the write lock on x (i.e. t_1 shares the lock on x with t_1, so t_2 cannot release any locks until transaction t_1 released its first lock), executes $w_2(x)$, then t_3 obtains a read lock on y and executes $r_3(y)$, then t_2 obtains the write lock on y (it shares the write lock on y with transaction t_3, so it must not release any locks until transaction t_3 releases its first lock) and executes $w_2(y)$. Next transaction t_2 cannot commit because transaction t_2 awaits for transaction t_1 and t_3 to release locks and only after they release their first locks, t_2 can release its locks and commit. Next t_3 acquires the lock for $w_3(z)$ and executes this operation and then it releases its locks and commits. Next t_1 acquires the lock for $r_1(z)$, executes this operation and then it releases locks, so that t_2 can now release locks and commit.

Theorem: All the O2PL algorithms with the compatibility tables LT1-LT8 produce only CSR schedules. The set of schedules produced by O2PL with the LT8 compatibility table is equal to the set OCSR.

Altruistic locking (AL)

Another variant of generalized two-phase locking is Altruistic Locking (AL). Let's assume we have these three transactions, a long transaction that writes a b c and so on and two other smaller transactions which just read two data.
$t_1 = w_1(a)\ w_1(b)\ w_1(c)\ w_1(d)\ w_1(e)\ w_1(f)\ w_1(g)$
$t_2 = r_2(a)\ r_2(b)$
$t_3 = r_3(c)\ r_3(e)$

If transaction t_1 obtains first a write lock on a, after it executes $w_1(a)$, although it doesn't use a anymore, it cannot release the write lock on a because it still has to obtain the write lock on all the next data, b, c, d, e, f, g. Even though t_1 doesn't deal with *a* anymore, it wrote the value of a and it doesn't care about *a* anymore, but two-phase locking requires transaction t_1 to hold the lock until all these other operations from t_1 obtain their corresponding locks. So, all this time, transactions t_2 and t_3 have to wait until t_1 finishes. The relaxation proposed by Altruistic Locking is for transaction t_1 to donate the write lock on a to some other transaction, because t_1 doesn't use it anymore. So we augment our model with a new operation:
$d_i(x)$ - transaction i donates the lock on x to other transactions

Using this concept of donating locks, a schedule for the above three transactions can be:
$wl_1(a)\ w_1(a)\ d_1(a)\ rl_2(a)\ r_2(a)\ wl_1(b)\ w_1(b)\ d_1(b)\ rl_2(b)\ r_2(b)\ wl_1(c)\ w_1(c)\ d_1(c)\ rl_3(c)\ r_3(c)\ wl_1(d)\ w_1(d)\ wl_1(e)\ w_1(e)\ d_1(e)\ rl_3(e)\ r_3(e)\ wl_1(f)\ w_1(f)\ wl_1(g)\ w_1(g)\ ru_2(a)\ ru_2(b)\ ru_3(c)\ ru_3(e)\ wu_1(a)\ wu_1(b)\ wu_1(c)\ wu_1(d)\ wu_1(e)\ wu_1(f)\ wu_1(g)$

Transaction t_1 after obtained the write lock on a then executed write a, donated the lock on a so that t_2 can acquire the lock on a. Then transaction t_1 donates the lock on b for transaction t_2 and so on. After showing this example, we can now formally specify the additional rules Altruistic Locking.

AL1: Once t_i has donated a lock on x, it can no longer access x.
AL2: After t_i has donated a lock on x, t_i still has to unlock x sometimes.
AL3: t_i and t_j can simultaneously hold conflicting locks only if t_i has donated its lock on x.

We need some additional rules in order to avoid problems and these are a little bit trickier.
We say that an operation $p_j(x)$ *is in the wake* of t_i ($i \neq j$) in s if $d_i(x) <_s p_j(x) <_s ou_i(x)$. We say that t_j *is in the wake* of t_i if some operation of t_j is in the wake of t_i. Transaction t_j *is completely in the wake* of t_i if all its operations are in the wake of t_i (meaning that all its operations should be should be between the donate operation and the unlock operation from this other transaction). Transaction t_j *is indebted to* t_i in s if there are steps $o_i(x)$, $d_i(x)$, $p_j(x)$ such that $p_j(x)$ is in the wake of t_i and ($p_j(x)$ and $o_i(x)$ are in conflict or there is $q_k(x)$ conflicting with both $p_j(x)$ and $o_i(x)$ and $o_i(x) <_s q_k(x) <_s p_j(x)$).

AL4: When t_j is indebted to t_i, t_j must remain completely in the wake of t_i.

Definition: Altruistic Locking (AL) is the locking based scheduling algorithm defined by general locking rules LR1-LR4, the two-phase locking property and the AL1-AL4 rules.

Let's consider the following schedule example:
$s = rl_1(a)\ r_1(a)\ d_1(a)\ wl_3(a)\ w_3(a)\ wu_3(a)\ c_3\ rl_2(a)\ r_2(a)\ wl_2(b)\ ru_2(a)\ w_2(b)\ wu_2(b)\ c_2\ rl_1(b)\ r_1(b)\ ru_1(a)\ ru_1(b)\ c_1$

This schedule does not obey the AL4 rule and is not even CSR. But if we correct this schedule such that it obeys AL4 we obtain a CSR schedule produced by AL.

$s' = rl_1(a)\ r_1(a)\ d_1(a)\ wl_3(a)\ w_3(a)\ wu_3(a)\ c_3\ rl_2(a)\ r_2(a)\ rl_1(b)\ r_1(b)\ ru_1(a)\ ru_1(b)\ c_1\ wl_2(b)\ ru_2(a)\ w_2(b)\ wu_2(b)\ c_2$

In the above schedule, t_2 stays completely in the wake of t_1.

The set of schedules produced by the AL algorithm is included in CSR, but it is a generalization of 2PL so the set of schedules produced by 2PL is included in the set of all schedules produced by AL.

Non-Locking based Pesimistic Concurrency Control

Basic Timestamp Ordering (BTO)

Let's consider an example. Let's say that we have two transactions:
$t_1 = p_1\ q_1$
$t_2 = p_2\ q_2$
And let's assume that p_1 is conflicting with q_1 and p_2 is conflicting with q_2, respectively. And also let's assume that $p_1\ p_2$ were already scheduled in this order (so we have the conflicting pair (p_1, p_2)). The scheduler knows about these two conflicting operations, it already executed them, p_1 first and then p_2, so the scheduler must make sure that for all subsequent conflicting operations

between transaction t_1 and t_2 the conflicting operation from t_1 should happen before the respective conflicting operation from transaction t_2. If after a while q_2 arrives at the scheduler, the scheduler detects that there's no other conflicting operation with q_2 (because q_1 didn't arrived yet at the scheduler, so the scheduler does not know about it), it executes q_2 normally. So far, our schedule looks like this:

$s = p_1 \ldots p_2 \ldots. q_2$

But later on, the operation q_1 arrives at the scheduler which is conflicting to q_2 and which should normally be executed before q_2 in order to maintain the order between transactions t_1 and t_2. But the scheduler has no possibility of executing this q_1 operation in the past, before q_2. So what happens in this case is that the scheduler aborts transaction t_1 because it's obviously that if the scheduler executes q_1 now or somewhere in the future, it will get a non-CSR schedule. This problematic situation is avoided by the two-phase locking property because a 2PL algorithm would not have been produced a schedule like $s = p_1 \ldots p_2 \ldots. q_2$ because when p_1 arrives at the schedule, transaction t_1 would acquire the lock on the data p_1 operates and when p_2 arrives later at the scheduler, p_2 is conflicting with p_1, so p_2 can not be granted any lock because the lock is still owned by t_1 for operation p_1 (since p_1 is conflicting with p_2, we know they operate on the same data item). Transaction t_1 can not release the lock on the data p_1 operates, because it still needs to acquire a new lock for operation q_1 which did not arrive at the scheduler yet and 2PL property says that once you release a lock, you can never acquire a new lock.

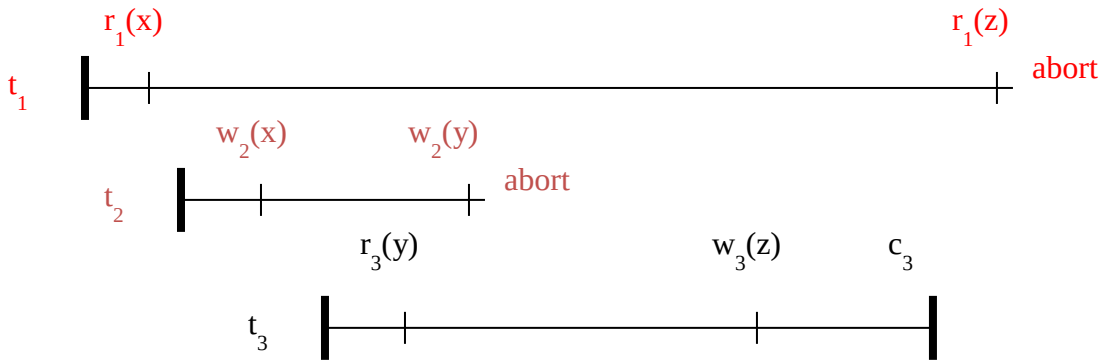
The idea behind Basic Timestamp Ordering is that when the first pair of conflicting operations of transactions t_i and t_j arrives at the scheduler, BTO decides to order the conflicting operations from this pair, first the operation from the older transaction (i.e. the one that started first) and then the operation from the younger transaction and then BTO does its best to keep this order between conflicting operations from transactions t_i and t_j. And the way BTO implements this is using timestamps for each transaction, whenever the first operation of a transaction arrives at the scheduler, the scheduler assigns a timestamp to that transaction and to all the operations from that transaction. And whenever there are two operations in conflict $p_i(x)$ and $q_j(x)$, the scheduler enforces the ordering of the older transaction followed by the younger one. Please note that this is not the only way to produce CSR schedules meaning that a perfectly good order would be to execute the younger transaction (i.e. the conflicting operations from the younger transaction) first and then the older one (i.e. the conflicting operations from the older transaction). But it's more probable that by executing the operation from the younger transaction and then the operation from the older transaction, we get to an abort. One example where executing the conflicting operation from the younger transaction first and then the conflicting operation from the older transaction is the following: $s = r_1(a)\ w_2(x)\ r_1(x)\ w_1(y)\ w_2(z)$.

We can now formally define Basic Timestamp Ordering (BTO) rules.

- Each transaction t_i is assigned a unique timestamp $ts(t_i)$ (e.g., the time of t_i's beginning)
- If $p_i(x)$ and $q_j(x)$ are in conflict, then the following must hold: $p_i(x) <_s q_j(x)$ if and only if $ts(t_i) < ts(t_j)$ for every schedule s.
- For each data item x maintain two timestamps:
  max-r (x) = max{$ts(t_j)$ | $r_j(x)$ has been scheduled}
  and max-w (x) = max{$ts(t_j)$ | $w_j(x)$ has been scheduled}.
- When a new operation $p_i(x)$ arrives the timestamp of transaction t_i is compared to max-q (x) for each conflicting q:

- if $ts(t_i) < max\text{-}q(x)$ for some q then abort t_i
- else schedule $p_i(x)$ for execution and set $max\text{-}p(x)$ to $ts(t_i)$

Let's also consider an example.
Ex.1. Operation order: $r_1(x)\ w_2(x)\ r_3(y)\ w_2(y)\ c_2\ w_3(z)\ c_3\ r_1(z)\ c_1$
The schedule produced by BTO is:



and in serial notation: $s = r_1(x)\ w_2(x)\ r_3(y)\ a_2\ w_3(z)\ c_3\ a_1$

In the above picture, time goes from left to right horizontally, so the timestamp of t_1 is smaller than the time of t_2 which is smaller than timestamp of t_3 (i.e. $ts(t_1) < ts(t_2) < ts(t_3)$). t_1 executes first $r_1(x)$, updates $max\text{-}r(x) = ts(t_1)$. Then $w_2(x)$ arrives, so $ts(t_2)$ is compared with $max\text{-}r(x) = ts(t_1)$ and $max\text{-}w(x)=-1$ (uninitialized) and because ts(t2) is larger than both, $w_2(x)$ gets executed and $max\text{-}w(x) = ts(t_2)$. Following, $r_3(y)$ arrives, $max\text{-}w(y) = -1$ (uninitialized), so $ts(t_3) > max\text{-}w(y)$ and $r_3(y)$ is accepted for execution and $max\text{-}r(y)$ is set to $ts(t_3)$. When $w_2(y)$ arrives, $ts(t_2)$ is compared with $max\text{-}r(y) = ts(t_3)$ and $max\text{-}w(y) = -1$, but because $ts(t_2) < max\text{-}r(y)$, the scheduler concluded that the operation $w_2(y)$ arrived to late and it aborts transaction t_2. Next $w_3(z)$ arrives and $ts(t_3)$ is compared with $max\text{-}r(z) = -1$ and $max\text{-}w(z) = -1$, so $w_3(z)$ is accepted for execution and $max\text{-}w(z)$ is set to $ts(t_3)$. Transaction t_3 commits. The last operation that arrives at the scheduler is $r_1(z)$, but because $ts(t_1) < max\text{-}w(z) = ts(t_3)$, transaction t_1 is aborted.

Theorem: The set of schedules produced by BTO is included in CSR.

Serialization Graph Testing (SGT)

Serialization graph testing (SGT) is a quite nice easy to understand. Whenever the scheduler executes a new conflicting operation, it just updates the conflict graph and after each operation it checks whether there is a cycle in the conflict graph. If there is a cycle, then just abort the transaction that caused the cycle because the schedule is not CSR anymore. The formal definition of SGT rules follow.

- For $p_i(x)$ create a new node in the graph if it is the first operation of t_i
- Insert edges (t_j, t_i) for each $q_j(x) <_s p_i(x)$ that is in conflict with $p_i(x)$ $(i \neq j)$.

- If the graph has become cyclic then abort t_i (and remove it from the graph) else schedule $p_i(x)$ for execution.
- A node t_i in the graph (and its incident edges) can be removed when t_i is terminated and is a source node (i.e., has no incoming edges).

Theorem: The set of schedules produced by SGT is equal to CSR.
Example: In the schedule $s = r_1(x)\ w_2(x)\ w_2(y)\ c_2\ r_1(y)\ c_1$, removing node t_2 at the time of c_2 would make it impossible to detect the cycle.

Optimistic concurrency control

We have two optimistic concurrency control algorithms and the idea is that all these protocols have three phases: a read phase, a validation phase and a write phase. Usually the validation phase is executed together with the write phase. In the read phase, each transaction just reads values from the global memory space or the global data set. And if the transaction wants to write something, it doesn't write in the global data set, but it writes values in a private workspace like a cache or a local memory. The other transactions don't see these values written by the current transaction in the private space. Then after the transaction is done, so no other operation should be executed by this transaction, it goes to the validation phase where it checks whether the values that were written in the private workspace of this transaction can be written into the global data set without breaking ACID properties of other transactions. If writing the private workspace data into the global dataset breaks the ACID properties of any transaction, then the current transaction is aborted. Otherwise, the current transaction is commited and the data stored in the private workspace is written into the global data set; this happens in the final, write phase. Of course, no other operation should happen while a transaction validates or writes; i.e. there should be no other operation that accesses the the global data set while a transaction validates and writes. So whenever a transaction validates and writes all its private data, it should lock the whole database/dataset. But of course, it is locked only for a finite, small amount of time. In summary, optimistic scheduling looks like this:

Each transaction t has three phases:

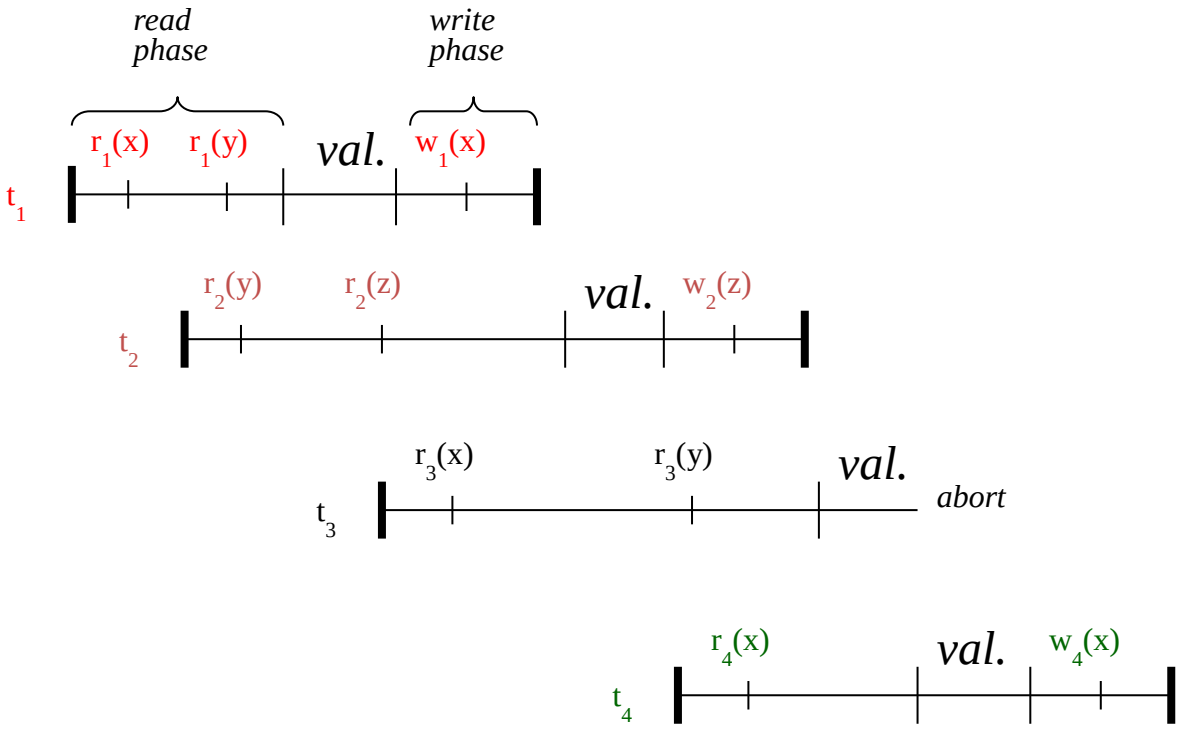
- **read phase:**
  execute transaction with writes into **private workspace**
- **validation phase (certifier):**
  upon t‘s commit request, test if schedule remains CSR if t is committed now based on t‘s read set RS(t) and write set WS(t)
- **write phase:**
  upon successful validation, transfer the workspace contents into the database (**deferred writes**),
  otherwise abort t (i.e., discard workspace)

Depending on how the validation phase happens, there are two types of optimistic concurrency control algorithms which are discussed in the following sections. The set of schedules generated by these two algorithms is included into CSR.

Backward-oriented Optimistic Concurrency Control (BOCC)

- Execute a transaction's validation and write phase together as a **critical section**: while t_i being in the **val-write phase**, no other t_k can enter its val-write phase
- Validation of transaction t_j: compare t_j to all previously committed t_i and accept t_j if one of the following holds:
  - t_i has ended before t_j has started
  - or $RS(t_j) \cap WS(t_i) = \varnothing$ and t_i has validated before t_j

Ex.1.

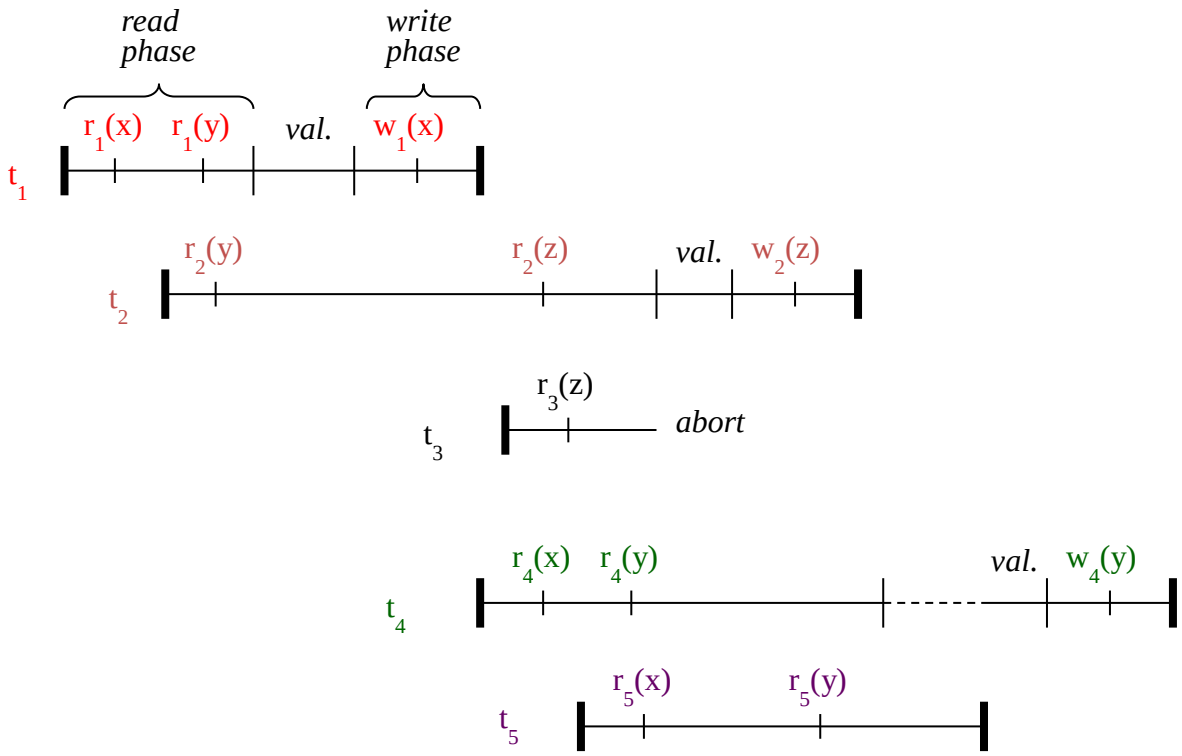


Forward-oriented Optimistic Concurrency Control (FOCC)

- Execute a transaction's val-write phase as a **strong critical section**: while t_i being in the **val-write phase**, no other t_k can perform any steps.
- **Validation** of transaction t_j: compare t_j to all concurrently active t_i (which must be in their read phase) and accept t_j if $WS(t_j) \cap RS^*(t_i) = \varnothing$ where $RS^*(t_i)$ is the current read set of t_i

Upon unsuccessful validation of t_j it has three options: abort t_j, abort one of the active t_i for which $RS^*(t_i)$ and $WS(t_j)$ intersect, and wait and retry the validation of t_j later (after the commit of the intersecting t_i). Read-only transactions do not need to validate at all.

Ex. 2.


read
phase
write
phase
r1(x)
r1(y)
val.
w1(x)
t1
r2(y)
r2(z)
val.
w2(z)
t2
r3(z)
t3
abort
r4(x)
r4(y)
val.
w4(y)
t4
r5(x)
r5(y)
t5

Multiversion Serialization and Concurrency Control

In all previous sections I have considered only mono version schedules, that is each data, whether x, y or z has only one specific value at a time. But now we will talk about multi-version concurrency control. We will assume that each data, x, y or z can have several versions that all exist at the same time and we will see that this gives us several advantages, several flexibility ideas which we can manipulate in order to produce better schedules. Stating this up front: if each data item has several versions at the same time, then of course, there is a time ordering between all those versions of the same data item (depending on the time of the version's creation). If we have this multi-version data, this would be very helpful at the end if we have to do rollbacks when we need to restore previous values of the data items that were modified by that transaction. So this would be the first advantage of having several versions (values) at the same time for each data item. But there's some other advantages that are brought by a multi-version storage. For example, let's consider the schedule:

$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_1(z)\ c_1\ c_2$

The above schedule clearly does not belong to CSR because there is a cycle between t1 and t2 in the conflict graph. Even though this schedule does not belong to CSR, this schedule would be tolerable if the operation $r_1(y)$ would be allowed to read the **old version** y_0 of y, not the version that is written by transaction t2. In that case it is like saying that we move this operation, $r_1(y)$, from its time point to a time point that's before $w_2(y)$. And this would give us a serializable schedule that's equivalent to the serial schedule t1 t2 which means it will give us a schedule that would belong to CSR. So all we need to do is just keep another y value with the previous value of y and make transaction t1 read that value when executing the operation $r_1(y)$. So nothing changes in the time of the execution of $r_1(y)$, but it's the logic of the execution that is changed. It's just like executing $r_1(y)$ before $w_2(y)$. A multiversion data set allows more flexible scheduling. This sequence of operations which would normally be excluded as not being CSR by a monoversion scheduler, would be CSR if it were executed by a multiversion scheduler. The method that we will follow throughout the course would be that each write operation creates a new version and we will identify the version that is newly created by indexing the data item with the number of transaction that writes/creates the new version. In the above example, $w_2(y)$ will create a new version of the data item y, so we write $w_2(y_2)$. Then each read step can choose which version it wants to read from and versions are transparent to the application and they are transient (i.e. they can be removed by the transaction system). We now introduce the version function and the multiversion schedule/history.

Definition (Version Function): Let s be a history with initial transaction t_0 and final transaction t_∞. A *version function* for s is a function h which associates with each read step of s a previous write step on the same data item, and the identity for writes (meaning the version function always associates the same write step to a specific write step).

Now we will introduce the concept of multiversion history and we will refer to the previous

histories that we've introduced so far as monoversion histories because they only have one version for each data item.

Definition (Multiversion Schedule)**:** A *multiversion (mv) history* for transactions $T = \{t_1, ..., t_n\}$ is a pair $m=(op(m), <_m)$ where $<_m$ is an order on op(m) and:

(1) $op(m) = \cup_{i=1..n} h(op(t_i))$ for some version function h,

(2) for all $t \in T$ and all $p, q \in op(t_i)$: $p <_t q \Rightarrow h(p) <_m h(q)$,

(3) if $h(r_j(x)) = w_i(x_i)$, $i \neq j$, then c_i is in m and $c_i <_m c_j$.

A *multiversion (mv) schedule* is a prefix of a multiversion history.

Example: Let's consider the multiversion schedule:
$s = r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(z_1)\ c_1\ c_2$

In this example $r_1(x)$ reads the x_0 version of data x and then writes a new version of data x the version, x_1, $r_2(x)$ reads the version x_1, then $w_2(y)$ writes a new version of data y which is version y_2. r1(y) does not read the most recent version of y, but it reads the previous version of y, i.e. y_0. So $r_1(y)$ reads the old version y_0 of y in order to be consistent with $r_1(x)$. So the above example would have the version function:
$h(r_1(x)) = w_0(x_0)$
$h(r_2(x)) = w_1(x_1)$
$h(r_1(y)) = w_0(y_0)$
and identity for all the write operations.

Definition(Monoversion Schedule)**:** A multiversion schedule is a *monoversion schedule* if its version function maps each read to the last preceding write on the same data item.

Ex. : The previous schedule example transformed into a monoversion schedule is:
s = $r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_2)\ w_1(z_1)\ c_1\ c_2$
Notice now that $h(r_1(y)) = w_2(y_2)$.

We will now go to serializability criteria for multiversion histories and then to scheduling algorithms or concurrency control algorithms just like we did for monoversion schedules. We will skip over the multi-version final stage serializability, we'll just talk about view serializability and conflict serializability. So similarly to monoversion schedules, we first define the read from relation and then define view equivalence for two multiversion histories and finally, we define view serializability.

Multiversion view serializability

Definition (Reads-from Relation)**:** For a multiversion schedule m, the reads-from relation of m is defined as **RF(m)** $= \{(t_i, x, t_j) \mid r_j(x_i) \in op(m)\}$.

Definition (View Equivalence)**:** Two multiversion histories m and m' having trans(m)=trans(m') are *view equivalent***,** denoted with $m \approx_v m'$, if RF(m) = RF(m').

Definition (Multiversion View Serializability)**:** The multiversion history m is *multiversion view serializable* if there is a serial monoversion history m‘ such that $m \approx_v m'$. We denote with **MVSR** the class of multiversion view serializable histories.

I’ve given you the equivalent definitions for multiversion histories and you've seen that these are very similar to the ones we've used for monoversion histories.

Ex.: The multiversion history does not belong to MVSR:
$m = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_0)\ r_2(y_1)\ c_2$

A history in order to be MVSR, we have to check whether the read from set of our history m is equal to the read from set of a serial schedule. The read-from set of history m is:
$RF(m) = \{(t_0, x, t_1), (t_0, y, t_1), (t_0, x, t_2), (t_1, y, t_2)\}$
Let’s now consider the monoversion serial history $m_{t1t2} = w_0(x_0)\ w_0(y_0)\ c_0\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_1)\ r_2(y_1)\ c_2$.
$RF(m_{t1t2}) = \{(t_0, x, t_1), (t_0, y, t_1), (t_1, x, t_2), (t_1, y, t_2)\}$
Let’s now consider the monoversion serial history $m_{t2t1} = w_0(x_0)\ w_0(y_0)\ c_0\ r_2(x_0)\ r_2(y_0)\ c_2\ r_1(x_0)\ r_1(y_0)\ w_1(x_1)\ w_1(y_1)\ c_1$.
$RF(m_{t2t1}) = \{(t_0, x, t_1), (t_0, y, t_1), (t_0, x, t_2), (t_0, y, t_2)\}$

Because $RF(m) \neq RF(m_{t1t2})$ and $RF(m) \neq RF(m_{t2t1})$ we can say that the multiversion history m is not MVSR.

Ex.2. Show that multiversion history $m = w_0(x_0)\ w_0(y_0)\ c_0\ w_1(x_1)\ c_1\ r_2(x_1)\ r_3(x_0)\ w_3(x_3)\ c_3\ w_2(y_2)\ c_2$ is view equivalent with the monoversion serial history $t_0t_3t_1t_2$.

We first compute the read-from set of our multiversion history m:
$RF(m) = \{(t_1, x, t_2), (t_0, x, t_3)\}$

We then consider the serial monoversion schedule $t_0t_3t_1t_2$:
$t_0t_3t_1t_2 = w_0(x_0)\ w_0(y_0)\ c_0\ \ r_3(x_0)\ w_3(x_3)\ c_3\ \ w_1(x_1)\ c_1\ \ \ r_2(x_1)\ w_2(y_2)\ c_2$
The read-from set of this serial schedule is computed the usual way:
$RF(t_0t_3t_1t_2) = \{(t_1, x, t_2), (t_0, x, t_3)\}$
He have $RF(t_0t_3t_1t_2) = RF(m)$.
So we can say that m is view equivalent with the serial monoversion schedule $t_0t_3t_1t_2$ which means that m is MVSR (i.e. belongs to the class of multi-version serializable schedules).

So this is multi-version view serializability and is very similar to the way view serializability is defined for monoversion schedules/histories with a little bit of changes when defining the read from set. MVSR is more flexible than VSR meaning it it says about schedules that many of them are correct according to a multi-version view serializability.

Theorem**:** $VSR \subset MVSR$

But of course, the same problem that we had for monoversion view serializable schedules remains: deciding whether a multiversion history belongs to MVSR is an NP-complete problem.

Theorem: The conflict graph of an multiversion schedule m is a directed graph G(m) with transactions as nodes and an edge from t_i to t_j if $r_j(x_i) \in op(m)$. For all multiversion schedules m, m‘, if $m \approx_v m$‘ this implies that the conflict graphs are equal: G(m) = G(m‘).

Something notable here is that this is just a unidirection relation meaning that if we have two multiversion schedules m and m’ which have the equal conflict graphs, it doesn't necessarily mean that those two multi-version schedules are view equivalent which was true about view serializability for monoversion schedules. We have an example below of two multiversion histories that have equal conflict graphs, but they are not multiversion view equivalent:
$m = w_1(x_1)\ r_2(x_0)\ w_1(y_1)\ r_2(y_1)\ c_1\ c_2$
$m‘ = w_1(x_1)\ w_1(y_1)\ c_1\ r_2(x_1)\ r_2(y_0)\ c_2$

But we can build some other type of graphs, not the conflict graph as defined previously for a multiversion schedule and if that graph is a acyclic then we can say that we have a schedule or history that is multiversion view serializable. This graph is introduced in the following definition.

Definition (Multiversion Serialization Graph (MVSG)):
A version order for data item x, denoted $<<_x$, is a total order among all versions of x.
A version order for multiversion schedule m is the union of version orders for items written in m.
The *multiversion serialization graph* for m and a given version order <<, **MVSG (m, <<),** is a graph with transactions as nodes and the following edges:

a) all edges of G(m) are in MVSG(m, <<)
   (i.e., for $r_k(x_j)$ in op(m) there is an edge from t_j to t_k)
b) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_i << x_j$ then there is an edge from t_i to t_j
c) for $r_k(x_j)$, $w_i(x_i)$ in op(m): if $x_j << x_i$ then there is an edge from t_k to t_i

The multiversion schedule m is in MVSR if and only if there exists a version order << such that the graph MVSG(m, <<) is acyclic.

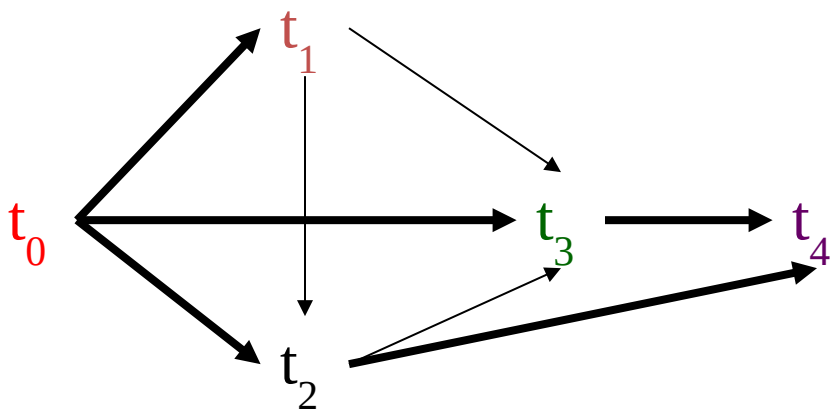
Example:
$m = w_0(x_0)\ w_0(y_0)\ w_0(z_0)\ c_0\ \ r_1(x_0)\ r_2(x_0)\ r_2(z_0)\ r_3(z_0)\ w_1(y_1)\ w_2(x_2)\ w_3(y_3)\ w_3(z_3)\ c_1\ c_2\ c_3\ r_4(x_2)\ r_4(y_3)\ r_4(z_3)\ c_4$
having the version order <<:

$x_0 << x_2$
$y_0 << y_1 << y_3$
$z_0 << z_3$

Let's build the multiversion serialization graph. For each read $r_k(x_j)$ operation, there is an edge from the from t_j to the t_k, so from transaction that writes to the transaction that reads. Let's find read operations. $r_1(x_0)$ reads x from transaction t_0 so there's a link from t_0 to t_1. There's another link from t_0 to t_2 because of this operation $r_2(x_0)$. There's another link from t_0

to t_3 because of this operation $r_3(z_0)$. There's another link from t_2 to t_4 because of $r_4(x_2)$. There's another link from t_3 to t_4 because of $r_4(y_3)$ (and also $r_4(z_3)$). And then for each read and write operation if x_i is before x_j then we have an edge from t_i to t_j. So we need to find an edge from t_1 to t_2. We apply rule c) from the definition; we have in this rule i=2, j=0, k=1. So we have $r_1(x_0)$, $w_2(x_2)$ in op(m) and we know that $x_0 << x_2$ which means that there is an edge from t_1 to t_2. And that's why we have this edge t_1 t_2. And the same goes for the remaining edges which are t_1 t_3 that we didn't talk about and t_2 t_3 and then we get this graph which is the multiversion serialization graph.

we have the MVSG(m, <<):



This multiversion serialization graph doesn't have a cycle and that's why the multiversion schedule m is multi-version view serializable.

Of course, the problem remains that testing whether there is actually such a version of the relation for which this graph is acyclic is still an NP-complete problem.

Multiversion conflict serializability

Let's move to conflict serializability for multiversion histories/schedules. A multi-version conflict is defined a little bit different than for monoversion schedules now. So for monoversion schedules we've defined a conflict to be two operations operating on the same data item but at least one of them should be a write. So we have a conflict between a read and a write operation or between two write operations. In the case of multiversion schedules we only have one type of conflict and which is defined between a read and a write operation and this read operation should be before the write operation. We don't really care about conflicts between two write operations even if they operate on the same data item because in multiversion histories they don't conflict anymore because the two write operations on the same data item will create different versions of the same data. If there are several versions for a data, the read operation is free to choose whatever version it wants to read. But if we have a read and a write operation on the same data item and the read is before the write and if we switch them, so the read becomes after the write, then there's a new version that this read might be able to read. So that's why a conflict is defined like this: between a read and a write operation and the read should always happen before the write. We formalize the

definitions of conflict pair, conflict serializability and conflict graph in the context of multiversion schedules, in the lines below.

Definition (Multiversion Conflict): A **multiversion conflict** in m is a pair $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

Definition (Multiversion Reducibility)**:** A multiversion history is **multiversion reducible** if it can be transformed into a serial monoversion history by exchanging the order of adjacent steps other than multiversion conflict pairs.

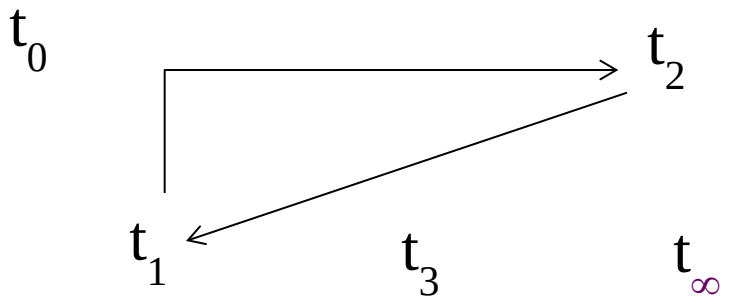
Definition (Multiversion Conflict Serializability)**:** A multiversion history is **multiversion conflict serializable** if there is a serial monoversion history with the same transactions and the same (ordering of) multiversion conflict pairs. **MCSR** denotes the class of all multiversion conflict serializable histories.

Definition (Multiversion Conflict Graph)**:** For a multiversion schedule m the **multiversion conflict graph** is a graph with transactions as nodes and an edge from t_i to t_k if there are steps $r_i(x_j)$ and $w_k(x_k)$ such that $r_i(x_j) <_m w_k(x_k)$.

The following theorem gives us all the possible ways to prove that a multivesion history is conflict serializable, using the conflict pairs or the conflict graph or the reductibility based on commutativity.
Theorem: The multiversion history m is in MCSR if and only if m is multiversion reducible. And m is multiversion reducible if and only if m‘s multiversion conflict graph is acyclic.

Multi-version conflict serializability is included in multiversion view serializability similar as for the monoversion context. Here we have an example which shows that the reverse is not true. So we have here a multi-version schedule that belongs to MVSR, but not to MCSR.
Theorem: MCSR $\subset$ MVSR

Example: The following multiversion history does not belong to MCSR, but it belongs to MVSR:
$m = w_0(x_0)\ w_0(y_0)\ w_0(z_0)\ c_0\ r_2(y_0)\ r_3(z_0)\ w_3(x_3)\ c_3\ r_1(x_3)\ w_1(y_1)\ c_1\ w_2(x_2)\ c_2\ r_\infty(x_2)\ r_\infty(y_1)\ r_\infty(z_0)\ c_\infty$
$m \approx_v t_0\ t_3\ t_2\ t_1\ t_\infty$



Moving on to concurrency control protocols, that is scheduling algorithms that work with multiversion data, we have the multi-version equivalent of two-phase locking, we have the multi-version equivalent of timestamp ordering protocol and the serialization graph testing.

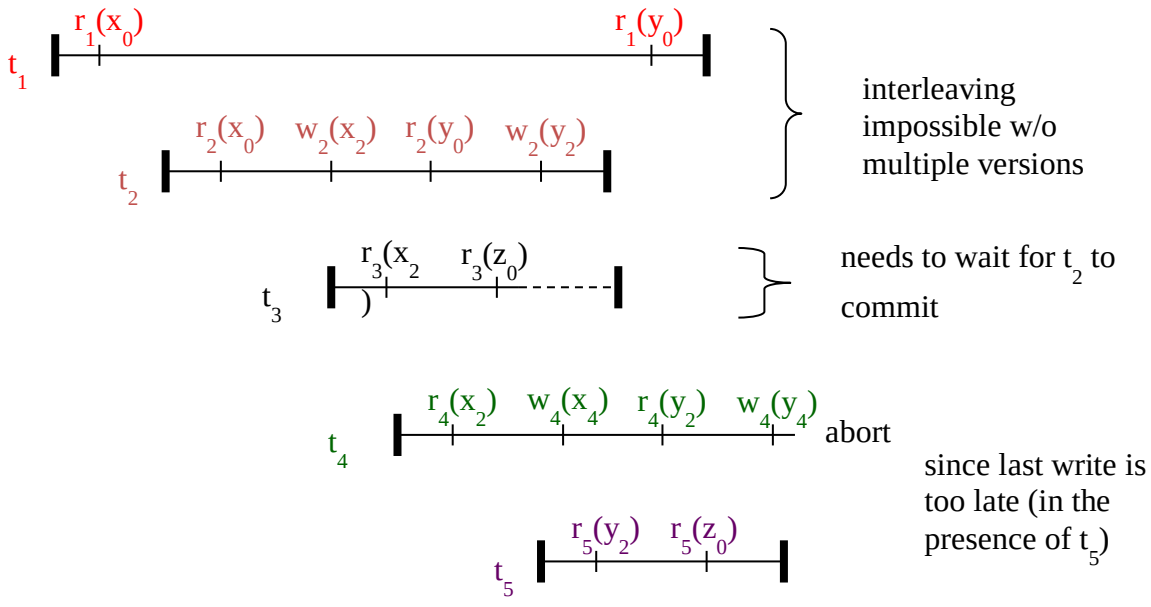
Multiversion Timestamp Ordering (MVTO)

In Multiversion Timestamp Ordering we still have timestamps assigned to each transaction and a read operation is mapped to a write operation that carries the largest timestamp that is still smaller than the timestamp of the current transaction. A write operation is accepted and creates a new version of the data or is rejected if it arrived too late. Each transaction t_i is assigned a unique timestamp $ts(t_i)$. $r_i(x)$ is mapped to $r_i(x_k)$ where x_k is the version that carries the largest timestamp from all the timestamps that are smaller than $ts(t_i)$. So the a read operation $r_i(x)$ always reads the version from the most recent write belonging to a transaction that started before transaction t_i. A write operation $w_i(x)$ is either:

- rejected if there is $r_j(x_k)$ with $ts(t_k) < ts(t_i) < ts(t_j)$ because $r_j(x)$ should read from the transaction with the largest timestamp from all timestamps smaller than $ts(t_j)$ and this would be transaction t_i, but it did not read from transaction t_i because $w_i(x_i)$ only now arrives at the scheduler and $r_j(x_k)$ already happened in the past
- or mapped into $w_i(x_i)$ otherwise

The c_i commit is delayed until c_j of all transactions t_j that have written versions read by t_i. This is done so that we can not have a situation where a transaction has read a value that is not committed (this is problematic if the transaction that wrote this value does a rollback later on). And of course the schedules that are generated by a multi-version timestamp ordering protocol are included into the multiversion view serializable set (MVSR) similar as for mono version schedules.

Example: a schedule where we have operations that arrive too late and are rejected.



Multiversion 2PL (MV2PL) Protocol

Now we introduce two-phase locking but for multiversion schedules. There are two versions for each data, a commited version and an uncommitted version. The idea is to use write locking to ensure that at each time there is at most one uncommitted version. For t_i that is not yet issuing its final step:

- $r_i(x)$ is mapped to "current version" (i.e., the most recent committed version) or the uncommitted version
- $w_i(x)$ is executed only if x is not write-locked, otherwise it is blocked

t_i's final step is delayed until after the commit of:

- all t_j that have read from a current version of a data item that t_i has written
- all t_j from which t_i has read

So all the transactions that depend one on the other either because tj has read versions written by ti or ti has read data from tj, they are all committed together.

Ex.: For input schedule:
$s = r_1(x)\ w_1(x)\ r_2(x)\ w_2(y)\ r_1(y)\ w_2(x)\ c_2\ w_1(y)\ c_1$
MV2PL produces the output schedule:
$r_1(x_0)\ w_1(x_1)\ r_2(x_1)\ w_2(y_2)\ r_1(y_0)\ w_1(y_1)\ c_1\ w_2(x_2)\ c_2$

The MV2PL schedule shifts $w_2(x)$ at the end because x is already locked by t1 for writing. The x_1 version is the uncommitted version of x. And in order to write it, t_1 must lock it for write. And t_1 does not release the lock until it executes all its operations.

The 2V2PL Protocol

The 2V2PL is just a specialization of MV2PL because it stores only two versions for each data item: a temporary version and a commited vesion. Transaction t_i will request write lock $wl_i(x)$ for writing a new uncommitted version and ensuring that at most one such version exists at any time. Transaction t_i will request read lock $rl_i(x)$ for reading the current version (i.e., most recent committed version). All transactions t_i need to request certify lock $cl_i(x)$ for final step of t_i on all data items in t_i's write set. The lock compatibility table is the following:

lock holder \ lock requestor	$rl_j(x)$	$wl_j(x)$	$cl_j(x)$
$rl_i(x)$	+	+	–
$wl_i(x)$	+	–	–
$cl_i(x)$	–	–	–

Example: For the following sequence of operations
$r_1(x)\ w_2(y)\ r_1(y)\ w_1(x)\ c_1\ r_3(y)\ r_3(z)\ w_3(z)\ w_2(x)\ c_2\ w_4(z)\ c_4\ c_3$
2V2PL will generate the following schedule:
$s = rl_1(x)\ r_1(x_0)\ wl_2(y)\ w_2(y_2)\ rl_1(y)\ r_1(y_0)\ wl_1(x)\ w_1(x_1)\ cl_1(x)\ u_1\ c_1\ rl_3(y)\ r_3(y_0)\ rl_3(z)\ r_3(z_0)\ wl_3(z_3)$
$w_3(z_3)\ wl_2(x)\ w_2(x_2)\ cl_2(x)\ cl_3(z)\ u_3\ c_3\ cl_2(y)\ u_2\ c_2\ wl_4(z)\ w_4(z_4)\ cl_4(z)\ u_4\ c_4$