Question 1

Motivation for DSMS

Not all problems can be solved with DBS. DSMS are used when DBS are too expensive. We have large amounts of raw data coming in from for instance satellites (measurements of earth geodetics). With DSMS we can query/analyse data in near real-time with SQL-like queries (which can be more complex or rather simple).

Differences

The main difference is that for DBS the data stays the same and queries change, while for DSMS the data changes constantly and queries stay the same (though they can be altered/added/removed). In DBS we have persistent storage of data (on disks) and queries that are used once, while in DSMS we can't afford to store all the data (possibly several GB of raw data arriving) and we don't need to. We don't know how and when data will arrive. We store the queries, there are no transactions, no concurrency problems (in DBS: transactions, concurrency control). We have to predefine schema for the data stream. Processing happens in main memory (it must happen fast, we have only a few nanoseconds before the data is gone). We analyse the data streams (there is a constant input) on-line. We have a set of continuous (DBS-like) queries, that filter out what we need, but what happens underneath is completely differnt in DBS and DSMS. Only the results are stored. Monitoring is an important part of DSMS.

Applications that make use of DSMS

Sensor networks, Network traffic analysis, Financial tickers, Online auctions

Question 2

(I used timestamp.format()because it makes the result much more readable. But that's only cosmetics. (Done so in all other queries that ask for dates)

Query:

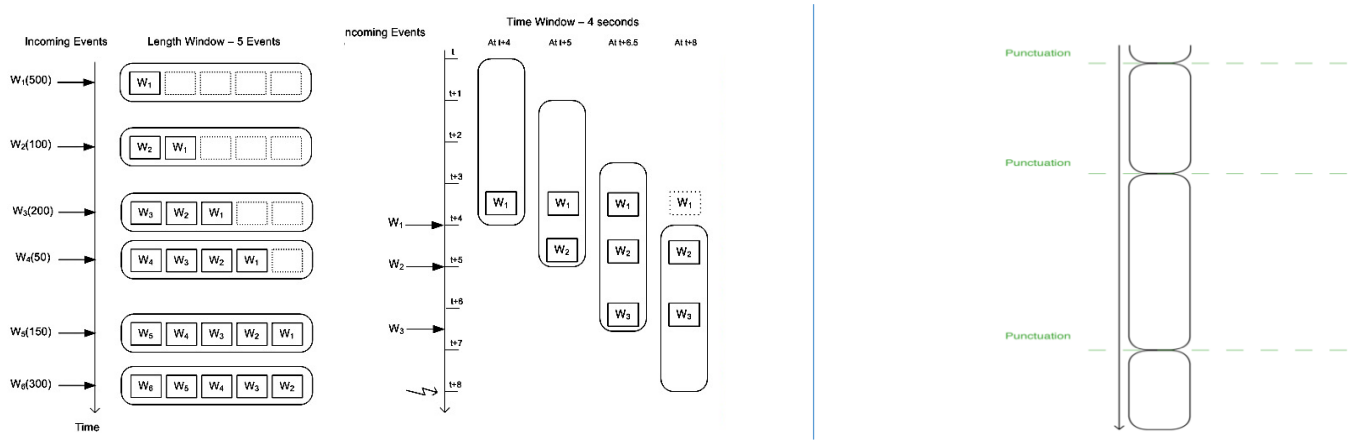
```
SELECT
        timestamp.format() as Date
FROM
        StockTick
WHERE
        close>17 and close<20
```

Output:

```
------------------
PERFORMING QUERY 1
------------------
EVENT! {Date=04.06.11 00:00}
EVENT! {Date=06.06.11 00:00}
EVENT! {Date=12.06.11 00:00}
EVENT! {Date=13.06.11 00:00}
EVENT! {Date=14.06.11 00:00}
EVENT! {Date=15.06.11 00:00}
EVENT! {Date=19.06.11 00:00}
EVENT! {Date=22.01.13 00:00}
EVENT! {Date=23.01.13 00:00}
EVENT! {Date=25.01.13 00:00}
EVENT! {Date=26.01.13 00:00}
EVENT! {Date=27.01.13 00:00}
EVENT! {Date=28.01.13 00:00}
EVENT! {Date=29.01.13 00:00}
EVENT! {Date=30.01.13 00:00}
EVENT! {Date=02.02.13 00:00}
```

Question 3

On data streams we work on unbounded data, rather than on static data sets. Windows in DSMS are a mechanism to get a relation from a stream. They make it possible to only look at parts of the stream. Only these parts will be considered during processing. Windows are sliding or tumbling windows.
There are several different types of windows. Windows can, for example, be tuple-count based or punctuation based. The latter creates boundaries withing a stream: a window buffer becomes ready for processing every time a punctuation is found and breaks up the input into substreams (a punctuation signals the end of a group, comprising the tuples that have been part of the flow so far, as well as the beginning of a new group, comprising all tuples up to the next punctuation). These windows can be of arbitray length.
Tuple-based windows, on the other hand, are of fixed length. They can be timed or count-based. Timed windows can for example look at the last 10 seconds of data, while count based windows consider, for instance, the last ten elements.



from chapter 3 of esper documentation (from the left: tuple-count based, timed window) & my own simplified interpretation of punctuation-based window. Inside my boxes are a random number of events (which I did not draw in).

Question 4

Thoughts:
To get 100 non-overlapping ticks, I used batch windows. It took me quite a while to figure out that it had to be length_batch(100), because the examples from chapter 3.6. only use time_batch(1 sec). Before it was enough to skim the pages, but not this time.
When I had that the rest was pretty straight forward. Use of min and max-functions in combination with the batch window.

Query:

```
SELECT
        max(weightedPrice) AS maxWeightedPrice,
        min(weightedPrice) AS minWeightedPrice
FROM
        StockTick.win:length_batch(100)
```

Output:

```
------------------
PERFORMING QUERY 2
------------------
EVENT! {maxWeightedPrice=0.36, minWeightedPrice=0.06}
EVENT! {maxWeightedPrice=1.06, minWeightedPrice=0.19}
EVENT! {maxWeightedPrice=8.67, minWeightedPrice=0.67}
EVENT! {maxWeightedPrice=29.58, minWeightedPrice=5.03}
EVENT! {maxWeightedPrice=6.11, minWeightedPrice=2.14}
EVENT! {maxWeightedPrice=7.05, minWeightedPrice=3.79}
EVENT! {maxWeightedPrice=6.71, minWeightedPrice=4.72}
EVENT! {maxWeightedPrice=13.26, minWeightedPrice=6.77}
EVENT! {maxWeightedPrice=17.15, minWeightedPrice=10.31}
EVENT! {maxWeightedPrice=214.67, minWeightedPrice=16.66}
EVENT! {maxWeightedPrice=134.02, minWeightedPrice=70.26}
```

Question 5

Thoughts:
This was relatively easy to accomplish. I simply added a (max – min) of weightedPrice which gives the difference. There might be a better way (since I am generating min and max twice).

Query:

```
SELECT
        (max(weightedPrice) - min(weightedPrice)) AS diff,
        max(weightedPrice) AS max,
        min(weightedPrice) as min
FROM
        StockTick.win:length_batch(100)
```

Output:

```
------------------
PERFORMING QUERY 3
------------------
EVENT! {min=0.06, max=0.36, diff=0.3}
EVENT! {min=0.19, max=1.06, diff=0.8700000000000001}
EVENT! {min=0.67, max=8.67, diff=8.0}
EVENT! {min=5.03, max=29.58, diff=24.549999999999997}
EVENT! {min=2.14, max=6.11, diff=3.97}
EVENT! {min=3.79, max=7.05, diff=3.26}
EVENT! {min=4.72, max=6.71, diff=1.9900000000000002}
EVENT! {min=6.77, max=13.26, diff=6.49}
EVENT! {min=10.31, max=17.15, diff=6.839999999999998}
EVENT! {min=16.66, max=214.67, diff=198.01}
EVENT! {min=70.26, max=134.02, diff=63.760000000000005}
```

Question 6

Thoughts:
For this question I generated the sum of all weighted prices per 100 ticks with the sum function. Same thought process as in question 4.

Query:
```
SELECT
        sum(weightedPrice) AS sum
FROM
        StockTick.win:length_batch(100)
```

Output:
```
------------------
PERFORMING QUERY 4
------------------
EVENT! {sum=9.17}
EVENT! {sum=39.289999999999864}
EVENT! {sum=230.47000000000017}
EVENT! {sum=1286.079999999999}
EVENT! {sum=358.8699999999993}
EVENT! {sum=523.9999999999995}
EVENT! {sum=540.7999999999997}
EVENT! {sum=1061.1}
EVENT! {sum=1265.1000000000008}
EVENT! {sum=6740.919999999999}
EVENT! {sum=10550.990000000007}
```

Question 7

Thoughts:
I used an inner SELECT-clause in the WHERE-clause to find the minimum and make sure that date and price correspond.

Query:
```
SELECT
        weightedPrice AS MinWP,
        timestamp.format() AS DATE
FROM
        StockTick.win:length_batch(50)
WHERE
        weightedPrice IN
        (SELECT
        min(weightedPrice)
FROM
        StockTick.win:length_batch(50))
```

Output:

PERFORMING QUERY 5

EVENT! {DATE=01.08.10 00:00, MinWP=0.06}
EVENT! {DATE=20.09.10 00:00, MinWP=0.06}
EVENT! {DATE=10.12.10 00:00, MinWP=0.19}
EVENT! {DATE=29.12.10 00:00, MinWP=0.29}
EVENT! {DATE=05.04.11 00:00, MinWP=0.67}
EVENT! {DATE=09.04.11 00:00, MinWP=0.74}
EVENT! {DATE=29.05.11 00:00, MinWP=8.3}
EVENT! {DATE=10.09.11 00:00, MinWP=5.03}
EVENT! {DATE=19.10.11 00:00, MinWP=2.28}
EVENT! {DATE=18.11.11 00:00, MinWP=2.14}
EVENT! {DATE=22.12.11 00:00, MinWP=3.79}
EVENT! {DATE=16.02.12 00:00, MinWP=4.19}
EVENT! {DATE=08.04.12 00:00, MinWP=4.72}
EVENT! {DATE=22.05.12 00:00, MinWP=5.08}
EVENT! {DATE=07.07.12 00:00, MinWP=6.77}
EVENT! {DATE=01.09.12 00:00, MinWP=10.0}
EVENT! {DATE=26.10.12 00:00, MinWP=10.31}
EVENT! {DATE=04.12.12 00:00, MinWP=13.03}
EVENT! {DATE=25.01.13 00:00, MinWP=16.66}
EVENT! {DATE=16.03.13 00:00, MinWP=46.86}
EVENT! {DATE=03.05.13 00:00, MinWP=91.3}
EVENT! {DATE=06.07.13 00:00, MinWP=70.26}

Question 8

Patterns are used to filter expressions. A pattern consists of pattern atoms (basic building blocks) and pattern operators (which control or combine atoms logically or temporally). There are four types of pattern operators: every, every-distinct, [num] & until. These control sub-expression repitition. The logical operators are and, or & not. The every and follow-by -> patterns are temporal operators and control event order. The every operator tells the pattern to restart a subexpression after trying to match. Without the keyword, it would simply stop. The followed-by operator checks first if the expression on the left of the arrow is true. If this is the case, the right side of the arrow will be evaluated for matches.
Last but not least we have pattern guards. They are where-conditions and control a subexpression's lifecycle.

(1) every A->B
This fires for every A event which is followed by a B event
(2) A->every B
This fires for an A event which is followed by every B event
(3) every A->every B
This fires for every A event followed by every B event
(4) every (A->B)
Looks for A events followed by B events and fires for all those combinations
Let's consider a stream: A1 B1 C1 B2 A2 D1 A3 B3 E1 A4 F1 B4

(1) would match for: (A1, B1), (A2, B3), (A3, B3), (A4, B4)
(2) would match for: (A1, B1), (A1, B2), (A1, B3), (A1, B4)
(3) would match for: (A1, B1), (A1, B2), (A1, B3), (A1, B4), (A2, B3), (A3, B3), (A2, B4), (A3, B4), (A4, B4)
(4) would match for: (A1, B1), (A3, B3), (A4, B4)

Question 9

Thoughts:
It was not easy to figure out how to use patterns. But with a little help from our TA I came up with the following: I am operating on streams (A and B). My pattern fires for every event A that is followed (not necessarily right after) by an event B with the condition. So if I find a date in A that is followed by a date in B where the weighted price in B is 100 times that of A and the volume of B is 40% bigger than the volume of A, I get a match.

Query:

```
SELECT
        A.timestamp.format() AS BEG,
        B.timestamp.format() AS STOP,
        A.volume AS VOL_B,              //volume at start date
        B.volume AS VOL_S               //volume at stop date
FROM
        PATTERN
        [every A=StockTick -> B=StockTick(weightedPrice >
        A.weightedPrice*100 AND volume > A.volume*1.40)]
```

Output:

```
------------------
PERFORMING QUERY 6
------------------
EVENT! {BEG=01.08.10 00:00, STOP=13.05.11 00:00, VOL_B=2601.0, VOL_S=64003.37}
EVENT! {BEG=12.10.10 00:00, STOP=01.06.11 00:00, VOL_B=25660.75, VOL_S=43479.96}
EVENT! {BEG=11.10.10 00:00, STOP=02.06.11 00:00, VOL_B=14092.91, VOL_S=41047.27}
EVENT! {BEG=10.10.10 00:00, STOP=03.06.11 00:00, VOL_B=50684.19, VOL_S=72980.24}
EVENT! {BEG=25.10.10 00:00, STOP=04.06.11 00:00, VOL_B=30299.64, VOL_S=44475.34}
EVENT! {BEG=28.10.10 00:00, STOP=07.06.11 00:00, VOL_B=21525.04, VOL_S=53715.04}
EVENT! {BEG=27.10.10 00:00, STOP=08.06.11 00:00, VOL_B=65605.87, VOL_S=104925.16}
EVENT! {BEG=06.11.10 00:00, STOP=09.06.11 00:00, VOL_B=32756.13, VOL_S=62933.66}
EVENT! {BEG=08.10.10 00:00, STOP=17.07.12 00:00, VOL_B=139287.29, VOL_S=210804.47}
EVENT! {BEG=30.12.10 00:00, STOP=22.02.13 00:00, VOL_B=2539.87, VOL_S=66951.69}
EVENT! {BEG=08.01.11 00:00, STOP=26.02.13 00:00, VOL_B=1629.44, VOL_S=42864.33}
EVENT! {BEG=07.01.11 00:00, STOP=28.02.13 00:00, VOL_B=42601.12, VOL_S=126517.41}
EVENT! {BEG=12.01.11 00:00, STOP=04.03.13 00:00, VOL_B=31359.77, VOL_S=46771.49}
EVENT! {BEG=13.01.11 00:00, STOP=05.03.13 00:00, VOL_B=20118.9, VOL_S=85425.52}
EVENT! {BEG=07.11.10 00:00, STOP=06.03.13 00:00, VOL_B=77218.72, VOL_S=126451.72}
EVENT! {BEG=30.01.11 00:00, STOP=10.03.13 00:00, VOL_B=10383.59, VOL_S=36363.25}
EVENT! {BEG=08.11.10 00:00, STOP=12.03.13 00:00, VOL_B=118203.83, VOL_S=183404.75}
EVENT! {BEG=31.01.11 00:00, STOP=19.03.13 00:00, VOL_B=63516.08, VOL_S=111627.22}
EVENT! {BEG=04.04.11 00:00, STOP=21.03.13 00:00, VOL_B=31155.46, VOL_S=94225.81}
EVENT! {BEG=01.02.11 00:00, STOP=25.03.13 00:00, VOL_B=31562.12, VOL_S=79730.06}
EVENT! {BEG=18.03.11 00:00, STOP=26.03.13 00:00, VOL_B=14226.22, VOL_S=56654.33}
EVENT! {BEG=04.02.11 00:00, STOP=27.03.13 00:00, VOL_B=42222.19, VOL_S=72182.81}
EVENT! {BEG=05.02.11 00:00, STOP=28.03.13 00:00, VOL_B=15786.81, VOL_S=140226.88}
EVENT! {BEG=22.02.11 00:00, STOP=29.03.13 00:00, VOL_B=12095.26, VOL_S=83152.75}
EVENT! {BEG=08.02.11 00:00, STOP=30.03.13 00:00, VOL_B=7021.29, VOL_S=37270.51}
EVENT! {BEG=02.03.11 00:00, STOP=31.03.13 00:00, VOL_B=2741.87, VOL_S=21092.38}
EVENT! {BEG=09.02.11 00:00, STOP=01.04.13 00:00, VOL_B=49632.5, VOL_S=90527.29}
EVENT! {BEG=12.02.11 00:00, STOP=02.04.13 00:00, VOL_B=4118.92, VOL_S=81260.8}
EVENT! {BEG=17.04.11 00:00, STOP=03.04.13 00:00, VOL_B=24225.7, VOL_S=152624.24}
EVENT! {BEG=22.04.11 00:00, STOP=04.04.13 00:00, VOL_B=39415.39, VOL_S=88124.78}
EVENT! {BEG=23.04.11 00:00, STOP=08.04.13 00:00, VOL_B=66620.36, VOL_S=114126.51}
EVENT! {BEG=27.04.11 00:00, STOP=09.04.13 00:00, VOL_B=33055.26, VOL_S=105724.64}
EVENT! {BEG=09.10.10 00:00, STOP=12.04.13 00:00, VOL_B=187846.95, VOL_S=556289.1}
```

Question 10

Distributed DBS consist of a database which is controled by a central DBS where storage devices are not sharing a common CPU, main memory or disks. They may be located in the same physical location or spread over networks (e.g. the Internet, intranets, ..). Distributed DBS are controled by
The big advantage of distribution is the improvement of database performance for the end user.
A centralized DBS is located at one place, but can be processed by several distributed computers across a network. Problems like bottle-necks can occur and the efficiency is limited in centralized DBS.

Advantages of distributed DBS

- sharing and accessing data is efficient and reliable
- each site is able to obtain a little control over data stored locally
- fault tolerance & high availability: if one site failes: the others may continue to operate (replication or duplication of data. data items may be available in several sites) -> failure does not imply shutdown
    - crucial for real-time applications
- possible to split queries into subqueries which can be processed parallelly -> speed,
    - workload can be balanced
- transparency on different levels, improved performance
- easy to modify the system (adding, removing without affecting other systems)
- transactions are reliable, (ACID properties for some systems)

Disadvantages of distributed DBS

- failure must be detected by the system and certain steps must take place to get it back up and running again, integration mechanisms can be complex
- complex server design results in more costly infrastructure, communication costs
- distributed transaction management (concurrency control): locks and timestamps
- security at remote sites must be ensured (and inbetween)

Architectural configurations

Homogeneous Distributed Database Systems: a network of two or more databases that reside on one or more machines. An application can simultaneously access or modify the data in several databases in a single distributed environment
Heterogeneous Distributed Database Systems: an automated system that integrates heterogeneous DBS to present the user with a single, unified query interface
Client/Server Database Architecture: a database server is the software that manages a database. Client is an application requestion information from server. Each node in a network can be a client, server or both

Question 11

Thoughts:
This was tricky. First I tried something like: fire for every A-event that is followed by a B-event (with condition 1 and condition2) *[Buying part]* or for every C-event that is followed by a D-event (condition1 and condition3) *[Selling part].* Trying it like this I got output, but it was clearly wrong since almost all BUY_DATE were null. So the TA pointed me to something like this: pattern [every a=A -> b=B( ) or c=C( ).
So I changed my pattern to: fire for every A-event that is followed by a B-event(cond1 and cond2) or C-event (cond1 and cond3) and it worked perfectly, I hope.

Query:

```
SELECT
        LastDate.timestamp.format() AS BUY,
        LastDate.close AS BUY_close,
        LastDate2.close as SELL_close,
        LastDate2.timestamp.format() AS SELL
FROM
        PATTERN
        [every FirstDate=StockTick -> LastDate=StockTick(volume>FirstDate.volume*1.70 AND
        weightedPrice<FirstDate.weightedPrice) OR
        LastDate2=StockTick(weightedPrice>FirstDate.weightedPrice*3 AND volume>FirstDate.volume*1.70)]
```

Question 12

Thoughts:
I was thinking like for question 11. I started again with the BUY-part. I had three dates to account for, so I was thinking something like:
pattern [every A->every B->C],
where A, B, C are FirstDate, SecondDate and ThirdDate. Fire for every A-event that is followed by every B-event which is followed by a C-event. It worked quite well, so I implemented the SELL-part accordingly:
pattern [every A->(B or C) -> (D or E)],
but I got null values for SELL. Which makes sense...
So, I tried:
pattern [every A->(every B->C) or (every D->E)].
It does make sense in my head. The parts in brackets fire for every event B/D followed by a C/E, and these fire only if they follow an (every) A-event.

Query:

```
SELECT
        ThirdDate.timestamp.format() AS BUYDATE,
        ThirdDate.close AS BUY_close,
        ThirdDate2.timestamp.format() AS SELLDATE,
        ThirdDate2.close AS SELL_close
FROM
        PATTERN
        [every FirstDate=StockTick->
        (every SecondDate=StockTick(weightedPrice>FirstDate.weightedPrice)->
        ThirdDate=StockTick(weightedPrice>SecondDate.weightedPrice AND close>FirstDate.close)) OR
        (every SecondDate2=StockTick(weightedPrice<FirstDate.weightedPrice)->
        ThirdDate2=StockTick(weightedPrice<SecondDate2.weightedPrice AND
        close>FirstDate.close))]
```

Question 13

What are pattern guards? Why cannot the timer:within pattern guard be used in simulated environments? What alternatives exists to control time and dates?
Pattern guards where conditions which control the lifecycle of subexpressions. An example is the timer:within guard. It works like a stopwatch. If the associated pattern expression does not turn true within the specified time period it is stopped and permanently false.
The timer:within cannot be used in simulated environments because they use system time as reference. This means that the simulated timestamp and the system time have to be in sync. Else it will go wrong. I actually tried to use timed queries and failed. So that's my theory.
An alternative is to use date-time methods. They work on date-time values and are independant from system time. With these it is possible to, for instance, to compare times and time periods, add or substract time periods,...
To get an interval which would be equivalent to a timer:within guard one could use a.finishedBy(b, 30 s) (two parameters, and interval 30s).

Question 14

Thoughts:
I had been skimming chapter 11 two or three times without getting any idea of how to get what I wanted. After reading about pattern guards, I was thinking "This is the way to go". I tried a couple of things but got too many lines, including the expected one. And I saw that the timer:within-function did not work as I wanted it to. Reading chapter 11 more thoroughly I discovered the Interval Algebra.
So I just put the condition for weightedPrice into my pattern and used a Where-clause after the pattern which contains the finishedBy date-time method with the 15 days parameter. It worked like a charm.

Query:

```
SELECT
        A.timestamp.format() AS StartDate,
        B.timestamp.format() AS StopDate,
        A.weightedPrice as StartWP,
        B.weightedPrice as StopWP
FROM
        PATTERN
        [every A=StockTick->B=StockTick(weightedPrice<A.weightedPrice/3)]
WHERE
        A.timestamp.finishedBy(B.timestamp, 15 days)
```

Output:

```
------------------
PERFORMING QUERY 9
------------------
EVENT! {StartDate=09.04.13, StopWP=65.33, StopDate=16.04.13, StartWP=214.67}
```

(formatted output ”StartDate=09.04.13 00:00” and ”StopDate=16.04.13 00:00” manually to fit into one line)

Question 15

Heterogeneous DBS are also called multi-databases or federated databases. Several databases appear to function as a single entity which have autonomy to a certain degree.
Even if we integrate existing DBS into a HDBS they won’t lose their stand-alone autonomy and keep working as before. Why do we use HDBS?: Old legacy DBS, time-consuming to migrate data, many different systems have to run with new data.
Which results in two problems: We want to keep working with our local applications, local transactions, but we still want to be able to use the DBS in a bigger context. We build data integration (logical). In contrast, data warehouses, store a copy of all the data.
Some transaction over many databases, while all is following Online Transaction Principle.
Data only in the local DBS, but create a layer that can access all the local data.
The local DBS will never be changed. They are running. In addition to local applications we will have a global integration layer which is a ”big tricky software bit” (Vera Goebel in lectrue) that has to be designed and built.
Very different local systems, depending on what we have, we might need compensating software to fill the gap to the global interface. The complexity of the integration layer depends on how many local DBS we have to integrate, how autonomous they are and how heavy the systems are. Stored data in the integration layer is metadata.
Tricky: whatever is running locally, globally we don’t know anything about it. What can one get from an autonomous system? Everything that the user interface is allowing you to access in the running system. So we might get statistics, information on what is happening locally. When we have a federation, we get export schema from local DBS up in the integration layer. They are willing to cooperate. Which is a great thing, because we get more information and can optimize globally.
Multi-database system: full autonomy means that we normally don’t export any schema and give it to a global layer for control and will not accept any cntrol from a layer on top. This has to be done artificially by pushing control with extra control software. Export schema will not get up to integration layer. All is more split up and harder to control.
The server has to run kind of a proxy around. It has to interact as a normal user and extract as best as possible to make it available afterwards.

Question 16

In the beginning it seemed so very overwhelming that we should learn to use Esper by ourselves. Streams are after all completely new to us. But after having started and reading the documentation parts that were hinted to, it became clearer how it works. Also, the first questions were kind of simple and straight forward to solve. I also was kind of scared because I did not know how this stream example would work. Would the data be lost if we didn't do it right at once? – Of course not.
With windows, I had to read more carefully. This is true for the rest of the query questions as well. I am used to skim through pages and look at examples and find what I am looking for. But most examples in the documentation used timed windows or were more complex than our test data. It happened two or three times that what I was looking for, was mentioned in an accessory sentence.
I liked that we could play around with our queries and try different approaches. I learned by failing many times, at least, I hope that is what I did.
I still find it quite hard to use the every operator correctly. For instance question 9. I used every A->B so that I get unique start dates, but can have the same stop date for different start dates. It would have made perfectly sense to use every (A->B) as well, with only unique combinations.
Another thing that was hard, was to explain functionality in my own words, when my only source was the documentation. Where things have been explained simple and to the point. And it was much more fun to work with the queries than the theory behind it.