Real-Time Pipelines with Flink and Kafka

Day 2 Lecture

Transcript:

for
1:30:23
day two I hope you enjoy it I wanted to talk a little bit today about what is
1:30:29
actually going on underneath the hood of like you saw how like uh on Tuesday
1:30:36
there was uh we have this Kafka thing that like you hit a website and then data just shows up in kofka right and so
1:30:45
I wanted to kind of explain like more in detail of like this right here is the
1:30:52
architecture of my entire website so let's start over here with like stick
1:30:57
figure man so this is you know this is you or a student or anyone who's visiting my website and then the first
1:31:04
thing that happens is you'll see that like if I want to go to exactly.com what happens is it gets that request gets
1:31:10
intercepted so there's this HTTP Interceptor and what that does is it just looks at the request and then it
1:31:16
logs it down to Kafka and then but it then it also just passes it forward so
1:31:22
then um exactly.com can then determine what to do with that request after uh it hits the server here hits hits Express
1:31:29
but then like okay when that request gets logged down it goes to K it goes to a kofka producer and then that kofka
1:31:35
producer puts it on Kafka and then we can then uh read from Kafka on the Flink
1:31:42
and then we can either dump it to another Kafka topic which is what one of those jobs did uh in uh day day two or
1:31:49
day one and there was the other job that dumped it to your local postcript right and then it did some of that like
1:31:55

you know geocoding IP address stuff as well so uh I want to talk about the other path here right so if it uh a lot
1:32:03
of times when it hits a server then uh what the the website will be like okay is this a web page request which means
1:32:10
it's like a normal like kind of like HTML request it's like are you asking for the about page or the login page or
1:32:16
the Boot Camp Page or like whatever page right if it is a webpage request then what it does is it renders its server
1:32:23
side cuz we use uh react right and it does serers side rendering and then it passes the rendered HTML to the client
1:32:30
and then that client is now ready for the the user to work with to click and
1:32:36
watch videos or whatever you want to do but if it's not then if it's not a
1:32:41
webpage request that means that it's an API request and if it's an API request that means that it's probably an update
1:32:48
or create or delete request so in this case uh it's going to hit postgress
1:32:53
remember how I was talking about uh in the Q&A how we use the per stack here so this is postgress right postgress over
1:32:58
here Express right here we have react and next next and react are kind of like coupled together and then all the the
1:33:05
whole thing kind of like runs on node.js as well so then uh postgress will run and this is like maybe you sign up and
1:33:11
then uh like if you sign up that's an API request and then uh it hits postgress and then it goes back to the
1:33:18
server and then it will redirect you because after you sign up usually it's going to go back to another page like the login page
1:33:24
or um if you log in which is also an API request it will then take you to the boot camp page right and then so this is
1:33:30
essentially the architecture behind exactly.com but my whole point here is
1:33:36

streaming needs to have some type of realtime architecture like this it has
1:33:43
to have a way to intercept realtime events so there's more to streaming than
1:33:48
just kofka right there has to be another layer there has to be the event generat
1:33:54
layer and in this case the event generation layer is this HTTP Interceptor and then that and that makes
1:34:00
it so we can see every time this is every time an action is taken by a user
1:34:05
that's what happens it gets dumped to kofka so this is essentially the architecture that we're kind of working
1:34:11
with uh in our boot camp so okay so so what do code does is it
1:34:18
sits between you and when you hit my server and you'll see what it has is uh
1:34:24
you see this like it's called a API event middleware and essentially what it does is you'll see has Rec res and next
1:34:30
so w is the request res is the response and next is how if I call next that
1:34:36
means it passes the request to the server so what happens here is uh first I have this this line here that says
1:34:41
okay should it be logged and essentially I'm like I don't want like if I'm in development and I'm doing like a Dev
1:34:48
work I don't I have my website's on Local Host and I don't want my events my development events to be logged to kofka
1:34:55
because I think that that's dumb because it's not real traffic so and then there's like uh there's also I don't want it to be a file request so like
1:35:02
every time that it requests like um an image like my logo or like uh any other
1:35:08
sort of like like the LinkedIn logo or all the different images that are on my website I also don't want to log that
1:35:14
because it's just like excessive and I only care about like the web page requests and the API requests so then we
1:35:20
have this event and this is the schema right that if uh if you remember like

1:35:25
when we were working with Kafka this is the schema that we work with right and you have the URL the referrer you have
1:35:31
the user agent this could be like if it's Google bot or Chrome or Android right and so then you have your schema
1:35:36
and then I have like this essentially if it should be logged or not and it's like okay if it passes the should be logged
1:35:42
then it sends the message to Kafka and then it calls next which means that then it passes the request to the server and
1:35:49
then the server does what it's supposed to do and then otherwise it just passes it to to the server anyway like um if
1:35:55
like it shouldn't be logged so that's essentially what this piece of code does this is a little tiny piece of code that
1:36:01
sits between you and my server that like uh essentially looks at every single request that comes
1:36:08
in okay so then we have the producer code so this is the producer code uh or
1:36:14
you saw you saw how like here we have that send message to kofka in the Interceptor well here is that send message to Kafka and then I create this
1:36:20
Kafka producer here which uh has a a similar sort of SSL connection like um
1:36:28
like we do in Flink where we have to connect with like all those like SSL credentials because we need to do that
1:36:34
on the producer side as well and then you'll see here I'm saying okay write to uh this porcupine boot camp events so
1:36:41
this is that topic or like I want to write this message to this table and then I have my message object which is
1:36:46
essentially just the object that I was I have over here this event object I'm going to be writing that and then we do
1:36:54
is we uh start up the producer and then we send the data to the producer and then this producer. send line will um
1:37:02

hopefully send the data to Kafka if it does then um it will say it we get a log that says message sent successfully and
1:37:09
then it will say result and then it will end otherwise it will um uh it will it
1:37:14
will send an airor message for us and but in and since it's just logging the Kafka I don't want this to throw an air
1:37:21
because it's just logs right so I essentially just say okay it's fine if it does if it logs or it doesn't log I
1:37:27
don't care I just want um uh because I I don't want that to stop the traffic to my website so anyways those are the two
1:37:35
pieces that happen Upstream of events Landing in Costco so now I want to talk
1:37:41
a little bit more about the future architecture of exactly.com and so this
1:37:47
is like you'll see is very similar except it has a couple more pieces to the puzzle here right so you'll see um
1:37:54
we have our Flink um job down here and then we have another kofka topic and
1:37:59
then you see how we have this like listens for events and then here's that kofka consumer that Doug was talking
1:38:05
about so this kofka consumer is just listening on this uh processed Kofa que
1:38:10
and then what will happen is this consumer will then push events back to exactly.com and then exactly.com will
1:38:17
push those events through a web socket and then the client will be updated in real time
1:38:24
so that's the idea I mean obviously like we're not quite there but this is what a DAT this is what a realtime data product
1:38:31
looks like is where you have a full loop where it has like uh you have data coming in and then it goes through and
1:38:38
then it goes back out back to the client and you have this like real time event stream and the client is very aware of
1:38:44
what's going on and so this and all this data would be updated as quickly as it possibly can and so the latency here
1:38:50

would be very low like on on the order of seconds for sure and uh and so that's kind of the idea of what uh the future

1:38:56

will look like and hopefully the future of this boot camp we will be able to implement more of these pieces as well

1:39:02

to show people how to build uh their own uh realtime updating UI so that they can

1:39:08

have their own kind of so they can see the real power of Flink and be able to build their own analytics dashboards and

1:39:13

stuff like that but that's kind of the idea of where the future is going to go with the architecture of the stuff but

1:39:20

um for uh today we're just going to be working more with um uh we're GNA kind of be still kind of

1:39:25

more just in this another Kafka topic and your local postgress so anyways one

1:39:31

of the big things that I think is going to be a massive massive thing in the future and going forward I've already

1:39:36

seen this happen at Netflix and at Airbnb is the whole idea of owning what's called a data product where you

1:39:43

have all this data that you're crunching but then that data somehow manages to get back into the app and that's what's

1:39:49

going to like that would be the idea here with these red things that are not quite built yet but that would be the

1:39:55

idea is that's how you would build a a closed loop cuz you see how now there's like a loop here right it's like you can

1:40:01

you can like there's like a you can go around in circles as many times as you want right and that's a that whole idea

1:40:07

of like a closed loop system is very powerful so um this is kind of the idea

1:40:13

behind uh data products I want to talk a little bit more about websockets because

1:40:18

I think websockets is something that uh not very many of you are probably

1:40:23

aware of like what a websocket even is so generally speaking when you are building a a website you have your

1:40:31
client and your server and the client always asks the server for more data and uh the client almost always initiates
1:40:38
the request and if you don't have if if the server needs to initiate the request like say the server gets new data and
1:40:45
then uh like the client usually has to ask for that new data even if the server got it and knows it's got it so the way
1:40:52
that and that's like Norm HTTP that's how normal HTTP Works um and so if you
1:40:57
want to have more of that two-way communication where the server can send stuff to the client as well then you
1:41:03
have to do this thing called a websocket and websocket is going to be how you um end up getting that like two-way
1:41:09
communication and that those real time updates so I want to talk a little bit more about uh competing architectures
1:41:16
here for um uh like how you process data so there's one called Lambda one called
1:41:23
Kappa architecture these are the two big like competing they're like the Pepsi and Coke or the you know uh kind of
1:41:30
competition uh different ways of doing things uh and let's go over each one I
1:41:36
have I like just to preface this like I have I've almost exclusively worked with Lambda architecture throughout my career
1:41:42
so Kappa architecture is mostly from Reading uh but I have a pretty good idea
1:41:48
about both of them Lambda architecture so imagine you have a pipeline that you want to
1:41:55
optimize and what it does is you are so you're creating a pipeline that is like
1:42:02
a batch pipeline but you want to optimize it for latency um and so you create a separate pipeline that's a
1:42:08
streaming pipeline that like lands earlier and is is the data is ready earlier because it's being processed in
1:42:14

real time and so how Lambda works is that you just have both you double the codebase you have a batch Pipeline and a
1:42:20
streaming pipeline that both write the same data and the main reason for that is the batch pipeline is there as kind
1:42:26
of like a backup if the streaming pipeline fails that's or the streaming pipeline has problems or issues or um
1:42:32
the correctness is a little bit weird right um so the pain here is uh double
1:42:39
code right that's the big pain of the Lambda architecture whereas uh you get a
1:42:44
little bit niceness from like the data quality perspective uh and stuff like
1:42:50
that but you the double code base is like one of the things that is a massive massive pain for when you're creating
1:42:56
this stuff um so that's essentially Lambda architecture like most companies Facebook Netflix Airbnb they like any of
1:43:02
these like Master data sets that they want to land as quickly as possible but they want to be as correct as possible
1:43:09
they use Lambda architecture to um solve that problem okay so let's just talk about
1:43:15
Kappa architecture real quick uh Kappa architecture is um where they say you
1:43:21
don't need both you just you can just use streaming so like Flink for example
1:43:27
you can actually flip Flink from streaming mode to batch mode so you can actually have Flink just read in data
1:43:33
and and do the flip itself so then you can have one code base and Flink can just handle the batch job as well uh and
1:43:40
that's good um it has some problems when uh like for example backfilling a lot of
1:43:45
history is hard because like Flink is still mostly geared towards reading from
1:43:51
Kafka and like if you're reading in like many many many days from Kafka like the
1:43:56
back fill is just super painful because you have to read in everything like in a row CU that's how kfka works is all the

1:44:02
data is in a line and so that is one of the things that is very painful about uh
1:44:07
Kappa architecture is those kind of back fills that's why Lambda architecture exists right is to kind of minimize that
1:44:13
problem uh one of the things that's nice though and this is something that Netflix is working on is they uh iceberg
1:44:19
is actually changing that right so it makes it so Flink uh can dump data to
1:44:25
Iceberg and then Iceberg you can have a nice partitioned table and so then Flink if you need a backfill with Flink you
1:44:31
can backfill days at a time right you don't have to backfill the whole line because kofka just has like one line of
1:44:37
data right whereas uh Iceberg can have partition data just like uh Hive and all
1:44:42
the other kind of data Lake kind of architectures so that's what's really cool about iceberg is Iceberg allows you
1:44:49
to do these appends so you can append and create and add more dat data to your
1:44:54
um pipelines as you go whereas Hive you couldn't do that and that's like a very big win for Iceberg and that's like what
1:45:01
probably my favorite thing about iceberg is that append because it allows the the streaming world and the batch world to
1:45:08
be mostly married and like at least from what I've been reading like Netflix is is starting to be more open-minded about
1:45:14
moving from Lambda architecture to Kappa architecture at least for a a certain
1:45:20
percentage of their jobs because of how iceberg is unlocking and minimizing some
1:45:26
of the pain that comes with uh like with Kappa architecture and like having to
1:45:31
Source all your data from Kafka so that's Kappa architecture so remember
1:45:37
Lambda architecture Capa architecture they're both kind of here like for me like the company the poster child of
1:45:42

Kappa architecture is Uber because they have uh like they're essentially a 100%
1:45:48
realtime company which makes sense right cuz they have to monitor where all their cars are at at all times and it's like
1:45:53
they can't be delayed so they're like uber is a very streaming first company and like it's very in their nature to be
1:45:59
streaming first and so uh definitely uh it's which architecture fits which
1:46:06
company does depend on like their business model and what they're working on I'm not saying one is better than the
1:46:12
other I just think that it should be good for you to be aware of both of these and like this question shows up in
1:46:18
data architecture interviews all the time I've been asked this question in in in data architecture interviews at like
1:46:24
every single company I've interviewed at since I became a staff engineer so uh yeah so definitely uh learn more about
1:46:32
these two architectures because they are very important so we're going to shift gears
1:46:37
here go back to uh so that was just a little aside about architectures that I think is important because it talks a
1:46:43
lot about streaming uh we're going to shift gears here to be talking more specifically about Flink
1:46:50
so Flink has udfs uh udfs are for like custom
1:46:57
Transformations Integrations we used a UDF on Tuesday and that UDF what it did
1:47:02
was it called an API and then it enriched our data set and we added some new columns that's great um so python
1:47:11
udfs are going to be not as performant as uh like J Java or Scola udfs and the
1:47:20
main reason for that is because they have to to have a separate process which
1:47:25
is like really gnarly so you have like the the regular Flink process that's running and crunching data and then uh
1:47:32

when you hit that UDF call it actually has to push it out of java and push it into python python executes it and then
1:47:40
it pushes it back into Java and so it has to I don't know if y'all ever used like a patchy arrow and it uses like
1:47:46
essentially it serializes things into a patchy Arrow moves it over into python calls a separate python process python
1:47:51
runs returns it its result moves it back into arrow and passes it back to Java whereas
1:47:58
if you are using pure Java then you don't have to do any of that right and
1:48:04
then you don't have like that because in that case you end up having two extra serialization steps that you or two
1:48:09
extra serialize and deserialize steps that you wouldn't normally have and so
1:48:15
that's where uh you want to be careful this is also uh this uh guidance we're
1:48:20
going to talk again about um next week this is generally speaking also uh good
1:48:26
practice and guidance uh when talking about Apache spark as well it has the
1:48:31
same issue especially with python udfs specifically because you have this like separate process so let's talk a little
1:48:38
bit more about window so this is going to get really uh kind of complicated and
1:48:44
messy so um there is essentially two types of Windows in Flink you have a
1:48:49
count window and then you have time driven Windows those are going to be the two different ways that you can define a
1:48:55
window and Flink and you can think of this as kind of like group by or like a
1:49:01
window function in SQL it's kind of like that but different
1:49:06
so we'll we'll go over a little bit more so when it's a datadriven window how that works is like you open a window and
1:49:11
then you say this window stays open until I see n number of events um and
1:49:17

then we have these time driven Windows tumbling sliding and session and we're going to go over each one of those like
1:49:24
in detail and so that y'all will be more aware of like what's going on
1:49:29
there so first off let's go with the count window count window is very straightforward so essentially how count
1:49:36
windows work is they open on the first event and then they close on when n
1:49:43
number of events happen keeping in mind that you can key here where like you can say like per user so it's like a window
1:49:49
per user kind of like um you know in the window functions in SQL you have
1:49:54
Partition by uh in Flink you get key by and uh that's going to be so you can
1:50:00
have your first event per user and one of the things that's another important part about these windows especially
1:50:06
these count Windows is that uh the number of events may never come right
1:50:12
say you're like I want 50 events to come and then someone does three and then they never do the other 47 and it's like
1:50:19
okay that window doesn't stay open forever so you also have to then uh you want to specify a timeout like might be
1:50:26
10 minutes or an hour or some sort of like whatever timeout makes sense for your use case and then that will close
1:50:33
the window even if the the number of events has not happened so these count
1:50:40
windows are very powerful for funnel analytics because funnel analytics often
1:50:46
times have a fixed number of events like for example like uh like the notific a
1:50:52
funnel at Facebook was generated sent delivered opened
1:50:59
clicked uh Downstream action so like I know seven something like that so then
1:51:04
like this can be this count can also be a distinct count so if they like click twice or open twice or whatever like
1:51:10

that's fine and they can do that and that is not going to be an issue so that's kind of the idea is how you um

1:51:16

how you can use these to kind of measure funnels because funnels usually have a fixed number of events and then you also

1:51:23

want to set a timeout of like well I don't want them to wait around forever it's like they click on a notification I'm not going to wait you know many days

1:51:30

to see if they actually end up following through so like you have to that's where the timeout is very useful when you're

1:51:36

kind of specifying windows in Flink okay so now let's talk about

1:51:41

tumbling Windows tumbling windows are really uh are very interesting these ones are very

1:51:48

uh like these tumbling windows are the closest comparison with uh data in batch so a

1:51:58

lot of times when you're in the batch world you have like hourly data or daily data and those have like a window and

1:52:06

that window is like from Midnight UTC to midnight UTC the next day right and it's

1:52:12

like 24 hours or it's from like 1 p.m. to 2 p.m and it's like fixed it's like from point A to point B and then

1:52:18

everyone in that window is going to be in that time right and that will be the

1:52:23

the window that you're looking for and so that is like this one is nice because

1:52:28

it has it's it's just very obviously the same as uh I say here it's very

1:52:34

obviously the same as like what you would expect from uh like batch so tumbling windows and batch are going to

1:52:41

be the most similar and you can obviously chunk data this way and a lot of times this is what people do is they

1:52:47

make these tumbling windows and then they dump it to hourly data and then you have data that's already processed

1:52:52

hourly and then you use batch uh at once it's at hourly and then batch picks up

1:52:59

everything else after that that that's a very common pattern at Facebook at least so um that's kind of the idea here this

1:53:06
is the most simple type of window we're going to cover this in the lab today as well this kind of tum tumbling window
1:53:12
it's uh one of the first types of Windows that I think is interesting in Flink let's go to the next one
1:53:20
so um then you have sliding windows so sliding Windows uh usually have a fixed
1:53:28
width but they can be like overlapping right so imagine if you have
1:53:35
um a fixed width window of an hour you could have one that's like from 1 to two but you could also have one from 130 to
1:53:43
230 and both of those windows are an hour and they're both valid right and so
1:53:49
and you'll see here how like you get these kind of like sliding overlapping windows and you get a lot of very
1:53:55
interesting patterns this way and you get uh obviously you get more duplicates this way as well like CU you see how
1:54:03
like uh like these records here are in window one and window 2 so you have to understand like how these sliding
1:54:09
Windows like how to manage them Downstream so that you're not like double counting or like what you're looking for so a lot of times that's the
1:54:15
whole point here though is that like you you're not looking you're not trying to aggregate all of these windows together
1:54:21
you're trying to find the window that has the most data and it's like like if
1:54:26
you pick one window there's no duplicates right because then you're fine you just don't want to look at all of it in aggregate because if you look
1:54:32
at all of it in aggregate you're going to have like twice as much data because all like pretty much there's going to be 50% overlap in each window so um one of
1:54:40
the things I think about here right is one of the things that this can do though is it can find your peak time
1:54:46
because if you have like a window that's an hour long but it's like shifted every 30 minutes then you can know like oh
1:54:52

maybe I actually my my Peak usage is like 1230 to 130 as opposed to like if
1:54:58
you looked at 12 to 1 and one to two those might be kind of flattened out because it's like the second half of
1:55:04
that first interval and the first half of that second interval are really high but then the other halves are kind of
1:55:10
low so then you can like there's like a spike that's like that crosses the windows and so that could be a it could
1:55:17
be very useful for finding like those types of peak uses right um it's also
1:55:22
good for like uh when you're crossing the mid the midnight boundary uh this is a this is a very interesting uh question
1:55:29
that we had at Facebook when we were talking about counting daily active users because it's like if you have a
1:55:35
user that starts their session at 11:58 at night like 11:58 U mid or 2 minutes
1:55:42
before midnight and then it goes to 2 minutes after midnight so it's like a 4minute session and uh do they count as
1:55:51
daily active on both both days like cuz it's one session right and
1:55:57
it's like it's only really one continuous use and it's like and it's not very long on either side so it's
1:56:03
like is that actually daily active for for both days and uh that's like one of those edge cases around midnight that
1:56:09
can happen and uh this sliding window was useful because one of the things it did was it showed us like just how many
1:56:16
people we were like essentially double counting um when we picked that time and
1:56:21
then we also did this crazy analysis where we were like okay what if we used a different time zone what if we didn't use UTC and we picked like a different
1:56:28
time zone that was more centered around a different place's midnight and we showed that like you can actually change
1:56:34
the number of daily active users by like an upward of like seven or 8% by just like changing the like the 24-hour

1:56:42
period that you pick because of and and that's because of this fact of people who have these like short sessions that
1:56:48
are overlapping with UTC midnight so so um that's one of the things that these
1:56:55
sliding windows can be very powerful at like finding is they can find those kind of uh situations for you and so um
1:57:03
anyways sliding windows are great uh I they're kind of Niche like I I I found
1:57:08
at least in my experience with Flink I um tumbling windows are going to be a lot more commonly used than sliding
1:57:16
Windows sliding Windows have like a very like specific uh use case like it's it's
1:57:21
not as much for like uh general purpose because of this weird overlap problem that's like why it's like why are we why
1:57:28
are we doing this to ourselves right it's more for like a very specific analytical use case as opposed to like building out Master data so um anyways
1:57:37
that's sliding Windows sliding windows are pretty cool all right then you have session
1:57:42
Windows session windows are um well this is the fixed length one oh oh I hold up
1:57:48
I messed up this is I need to update this slide this slides this slides off um but um so session based windows are
1:57:54
different so um how session based windows work is they are based like
1:58:01
they're user specific they're not like a fixed window so it's like you can imagine when you sign in uh that and you
1:58:08
have your first event that's the start of the window and then the window will last until there's a big enough Gap so
1:58:15
there's like a gap definition that you need to specify usually maybe it's five 10 20 minutes or something like that
1:58:21
where there's like a gap between when you have no data and like when you have data and so that is going to be the big
1:58:29

thing that you want to look at with sessions this is session sessionization is something that y'all are going to be
1:58:35
working on in the homework and it's one of the more powerful ones that I've noticed uh that will because this is the
1:58:41
one that can really help you determine like how users are behaving because you can see like when they first start using
1:58:47
the app and when they end using the app and so you can get a very clear picture of like how users are using your app in
1:58:54
real time because you can get the beginning and end of their session so that's essentially how sessions work and
1:59:01
uh that's configurable as well maybe it's one minute or however how however much of a gap You're Expecting from your
1:59:07
users when they're are um navigating the website but that's kind of the
1:59:12
idea okay we're going to cover Just One Last topic that I think is important uh and we'll we're going to do a little bit
1:59:18
of this in the lab uh it's kind of hard to emulate this stuff in real life uh because of the volumes and like I don't
1:59:24
have like the right uh setup for this yet but um so essentially you know you
1:59:30
have two ways to deal with out of ordered or late arriving events and the way it kind of works is like if it's a
1:59:36
little bit late then water marking is going to be good if it's late on the order of a couple seconds usually
1:59:42
Watermark is going to be great if it's late on the order of minutes then allowed lateness is going to be great uh
1:59:49
this essentially is like if you have a really it it all depends on like how late you are it's very similar to like
1:59:56
uh like you know when you have a zoom call with someone and like usually if you're like less than 5 minutes late you
2:00:01

just don't say anything and you go into the zoom call and like you act as if you're on time but if you're like more than five minutes late like you usually
2:00:08
want to text them like yo I'm a little bit late and then so that they aren't just sitting around like what the hell's going on so it's similar like Flink has
2:00:14
the same sort of idea where like you would think of like the watermark here would be like okay anything as long as
2:00:21
they show up in the next five minutes like we're going to count that as like in order but then if it's over five
2:00:26
minutes then we're going to count that as late and that's where you have this like allowed lateness which is like another way to do things but one of the
2:00:33
things that's rough about the allowed lateness is that if you end up getting that really late data uh it will
2:00:39
actually reprocess and it will it will either generate or merge uh data that
2:00:46
might have already been flushed because that data might have already the window might have already closed so it might
2:00:52
end up reading what was what was in the window that closed and then bringing in this new event and then closing it again
2:00:59
and you might get another window like you might get a second record uh or
2:01:04
you'll get an updated record like and that what that's like where this allow lightness can be really funky and like
2:01:09
how it like the actual Behavior behind what it does and like uh it usually does the merge but sometimes it does gener it
2:01:17
adds another rence so that's the whole thing that's where like for me at least when I've been working with Flink I'm all that watermarking and I usually just
2:01:23
don't let like if the record is 5 minutes late the way I I look at it is that like it's similar to like my zoom
2:01:29
call stuff where like most of the time if I have a one-on-one Zoom call with someone and they're more than five minutes late I assume that they're not
2:01:34
coming and uh and that's like why I have most of the time I have allowed lateness set to zero so I'm like whatever if it's

2:01:41
super late I don't care but if it's within the watermark that's fine that's uh my perspective on that but obviously
2:01:49
it depends on the use case because some use cases like you need to capture 100%
2:01:55
of the data even if it is really late so uh that is uh obviously not just like
2:02:01
don't take that advice for every single streaming pipeline but probably for most congrats on getting to the end of
2:02:06
streaming data pipelines day two lecture if you're taking this class for credit make sure to switch over to the next tab so that you can get credit for the lab